M1 Exploration - v 0.70

This remains a preliminary version.
Everything up to L1 cache discussion I am fairly confident of, having had some time to perform experiments, read the patents carefully, and edit the writing.
The L1 cache discussion and everything subsequent is a lot more tentative. Poorly/not yet edited, no experiments performed (or I'm unhappy with some of the experiments because I keep thinking of new ways to interpret the results so they're not definitive). Many of the patents I refer to I think I have the essential ideas correct, but that's the result of a quick scan and analysis, not a thorough reading or a tracking down of all the related patents.

Even so, pretty soon the A15 and its companion chips (M2?) will be out, and people will want to start exploring those. Hopefully with this as vade mecum, they can spend more time getting to fundamental analysis and truly understanding what's new, and less time re-inventing the wheel.

We're all in this together, trying to understand. However I'm exhausted and others probably also have good ideas. I hope others can contribute, so we can continue the great work.
I will keep updating this and maybe publishing new updates every month or so. Who knows when it will be done? When I started I had no idea it would become such a deep dive, or that so much could be (somewhat reliably) established.

This document was written for myself, to remind me of, and to organize, my investigations into the M1. These investigations took the form of experiments, and reading many Apple patents, all tied together by a reasonable knowledge of the academic literature. The audience is anyone who is interested in technical details of the M1. The level is somewhat choppy, but assume a substantially higher degree of knowledge than ye average internet opinionator on CPU's. I've included a huge number of references to papers and patents -- if you want to understand this stuff, read them. Yes it takes works. Yes, you have to do that work; no-one else can do it for you.
I expect the presentation will be to pretty much no-one's tastes. What can I say – skip over the parts that don't appeal to you, whether that's how an experiment was designed, how it was interpreted, a description of the literature surrounding a point, or a patent dump.
There is some repetition, in part because some material naturally fits into more than one place, and because reading the same thing in different contexts helps with understanding.

Obviously I've done my best to make this accurate. Even so, there are probably multiple errors, whether of experimental design, implementation, analysis, my understanding of a patent, or anything else. Technical corrections are welcomed.

Experimental Setup

To run the experiments you will need a test setup. I used the one created by Dougall Johnson here https://gist.github.com/dougallj/5bafb113492047c865c0c8cfbc930155, and https://gist.github.com/dougallj/c9976a52d592af24960ea7989cf652b1.
Dougall is the true hero of all this M1 investigation, doing the hard work of creating a useful test harness, especially all the low-level OS nonsense required to create JIT'able pages, set up the CPU counters, and so on, along with a python script to convert lines of ARMv8 assembly into machine code (required to make any interesting modifications to the tests).
Dougall also created the M1 instruction cycle counts web page at https://dougallj.github.io/applecpu/firestorm-int.html, and it's worth reading his investigations into the M1 at https://dougallj.wordpress.com/author/dougallj/.
Having said all that, much of my technique deviates substantially from both what Dougall did and what (Travis Downs, and, earlier, Henry Wong did). My technique is probably less numerically precise than Henry Wong's technique, but I found it easier to modify and change, something essential for this sort of open-ended research where one has no idea quite what to expect.

Note something ***extremely*** important and not at all obvious.
In Dougall's code there is a fragment of code called add_prep() that zero's out as many of the integer or FP/SIMD registers as feasible. This seems like an optional flourish that might be important for certain specialty measurements (like the size of the physical register file). Not so!
What is important is not the zero'ing of these registers (any value will do), it is marking them as "being used by this app". If this is not done then any code that reads these registers will run substantially slower than expected. For example, while the throughput of basic integer `ADD` is 6-wide, if your code is

reading a register like x5 (eg `ADD x0, x5, x5`) that has not been "claimed" it will run at something more like 4-wide, with constant weirdness and results that don't make sense.

I mention this because, of course, I hit this very issue, occasionally switching this off (in my version of the code), forgetting to switch it on, and then wanting to cry as nothing I did from that point on made any sense or matched my earlier results.

(Note that little of this will make sense, especially if you're not yet an OoO CPU expert, till you have read the entire report.)
What is actually going on here?!? I have no idea, but much much later, after we understand many more features of the M1, we will revisit the issue. My *guess* is that we are colliding with a security feature that is supposed to prevent access to "unauthorized" registers.

Recall the SPECTRE exploits some years ago in all their various forms; the common theme was a fear that, under the right circumstances, generally speculation beyond an inappropriate point, a piece of code could read or perhaps even "influence the value of" machine structures that were not appropriate to it.
Apple has a specific patented solution for this for the branch predictor; essentially every entry in the branch predictor is tagged with a value that incorporates at least some bits from every sort of indicator that might be violated by an exploit, so the tag includes an exception level (hypervisor, OS, user), a processID, even some high address bits (to catch JIT'd code spying on the rest of the process). We will discuss this much later when we cover branches.

However this same scheme is very general, and is almost certainly being applied to registers. The idea would be have a security tag for each mapping in the architectural to physical address mapping. Each time the mapping is referenced, the tag is compared with the current security tag and, in the event of a mismatch, various things happen which presumably include the CPU
- providing some value (zero, or random) that is not the architected value (and the value in the physical register), meaning that an app cannot, eg, read the registers used by the OS across a system call
- noting the mismatch somewhere, incrementing some register
- providing, probably, some debug mode (maybe only available within Apple) that would force an exception at this point.

Normal code should never find itself in a position where it is reading a register that it, itself, did not previously write (at which point thee security tag was set). It's only either malicious code, buggy code, or weird code (like ours, which cares only by timing `ADD`s, not the fact that the `ADD` is reading a random register value!) that would ever read a register with a mismatching security tag.

So, put it bluntly, every time my code tries to read x5, the core will notice that x5 is currently "owned" by some other thread (probably the OS or an interrupt), and this incurs some substantial additional cost. By overwriting every register at the start of our code, we “claim ownership” of that register and avoid this security violation and, in particular, its overhead.

We can probe this further. What if we use something like `MOV xn, xn`, rather than the current `MOV xn, #0`, to force the overwrite? That does *not* work! Either that instructions is mapped to `NOP` earlier in the pipeline, or

the zero-cycle rename stuff kicks in, without ever validating the thread ownership of the source register.
(My bet is on `MOV xn, xn`, being mapped to `NOP`.)
It's worth noting that I can't generate the same sort of disaster by not forcing ownership of the FP registers. If the "loss of ownership" of the registers is occurring at an OS-call or interrupt boundary, and the OS/interrupt does not touch FP registers, this makes sense.

There is an alternative possibility, that we are seeing a mechanism for hardware context switching, to be discussed much later. However the facts seem a better fit with the security violation explanation.

As an aside, at this point might I point out how much I absolutely *LOATHE* XCode. Everyone associated with the Debug side of the product should be deeply ashamed of themselves. It is beyond pathetic that this multi-gigabyte behemoth provides a worse debugging experience than freaking Macsbug from 1981, let alone Think C or Metrowerks. In particular, the inability to *IMMEDIATELY* display a pointer as an array of that time is beyond incomprehensible.
If I never engage in any more of this work for further Apple SoCs, a desire never again to repeat the XCode experience will be a large part of the reason.

Mathematica Setup (not relevant if you're reading the PDF)

Need to use a conditional on "printing to PDF" to hide this!

This writeup was all done in Mathematica. If you have access to Mathematica, you can download the companion notebook and look at the actual numbers, draw your own graphs from those numbers etc. But most people don't have Mathematica, so for you I've printed the notebook to a PDF.

If you use Mathematica, we need a way to paste results data from the command line apps into Mathematica.
Easiest solution appears to be
http://szhorvat.net/pelican/pasting-tabular-data-from-the-web.html

Note the "Show Input" button at the top of this notebook; toggle it if you want to see the (sometimes copious!) input data for any particular graph.
The Mathematica code below adds that functionality to this notebook (not shown when "Show Input" is untoggled.

Also ctrl - clicking on a graph brings up a contextual menu, one of whose items, "Get Coordinates" is useful in getting a quick, reasonably accurate, feel for the coordinates of a point .

Theory of a modern OoO machine

Introduction

Around the 2000's up to the 2010's a number of nice articles were published giving overviews of how current OoO CPUs worked. However 20, even 10, years is a long long time in CPU design (think what you would do if you had 2^5 or 2^10 x resources for a project, how differently you would proceed!), and ideas that were state of the art back then are baseline today.
You should be able to read something like David Kanter' s Nehalem overview, (2008) https://www.real-worldtech.com/nehalem/ and understand all the terms introduced, know what a ROB is, know what instruction scheduling is, know why register renaming exists, and so on. For a very simple overview of some aspects of a modern design, you should read (2008) https://carrv.github.io/2020/papers/CAR-RV2020_paper_15_Zhao.pdf *SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine.*
Our goal is to move to a level substantially beyond that.
You will encounter a lot that is unfamiliar, but I will try at every stage to explain the reasons why things are done, or not done, as they are.

For obvious reasons, some of this article has to be speculation. But it is informed speculation. We have three types of sources available.

- There are academic papers, and I will frequently reference these, which explain that something can be done, at least one way of doing it, and how well it works. They don' t prove that anything is implemented, in the M1 or anywhere else, anywhere, but they do explain the details of a technique, and that it is feasible.

- There are Apple patents, which explain in detail the precise innovation that is to be patented, and that often, as part of their explanation, include other interesting details of the design. It is always possible to claim that a patent, and the details that it contains, tell us nothing about how Apple actually does things; that the patent was simply filed as a good idea that Apple may eventually want to use but not yet. That certainly has happened – Apple has a large collection of patents filed around an architecture called Macroscalar
(2004) https://patents.google.com/patent/US8412914B2 *Macroscalar processor architecture,* based on very flexible indefinite length vectors, and which have not yet turned into a product. (It' s unclear whether SVE was in part inspired by Macroscalar; many of the ideas seem similar…)
However when a patent is narrowly defined, makes clear sense, and fits in with everything else we know, it' s sheer stubbornness to insist that it does not "prove" anything; our goal here is understanding, not impossible standards of certainty that will never be attained until perhaps the relevant designers write their memoirs .

To put the patents in context, remember a few dates:

- Sept 2012 is the release of Swift/A6.	3-wide OoO, first (visible...) Apple core	1st gen
- Sept 2013 is the release of Swift/A7.	6-wide OoO, add 64-bit	2nd gen
- Sept 2017 is the release of A11.	8-wide OoO, drop 32-bit, clusters (P and E) as the basic unit	3rd gen
- Sept 2021 [reckless speculation!]	10-wide?, ARMv9?, virtual registers?	4th gen
- Sept 2025 [doubleplus speculation!]	12-wide? drop x86 support?	5th gen

- Finally there are code experiments. One can write carefully designed programs, measure their timing, perhaps augment that information with readings from Performance Monitor Counters, and try to figure out from the results what' s going on. This is the only way to establish quantitative information – but one has to be careful.
Ultimately all the program tells you is how long it took; it does not tell you why. Benchmarks without understanding gives you Phoronix, but we subscribe to the Richard Hamming viewpoint: *the goal of computing is insight, not numbers.*
Much of the early information about the M1 is not so much incorrect as extremely incomplete, because people have been running these programs without a good model of the machine, interpreting it as much like a standard x86 machine. Our goal here is to go beyond that, to explain how these programs work, what they measure, and how to interpret what they measure.

I would hardly expect you to be able to read the papers or patents referenced right now. But hopefully, if you get to this end of this document and want to learn more, you' ll be in a position to work your way through them. Be patient! The first few times reading a paper or a patent is very difficult. The trick is to accept that you don' t need to understand everything.
Read the parts that you understand, skip the parts that don' t interest you.
But take time to struggle through the parts that do interest you, but are unfamiliar – that' s where the value is!

You will find, after repeating the exercise five or ten times, you have begun to understand the structure of patents and papers, at which point you can be a lot faster, knowing what can be skipped over and immediately heading for the good stuff. I tend to skim the diagrams, looking for those that represent the part I care about, then looking for the explanation of the diagram in the DETAILED DESCRIPTION OF EMBODIMENTS section (this is most easily done by search for one of the numbers that appears on the diagram). Lawyers care especially about the Claims section, which is the legally binding part, but I find there tends to be nothing interesting there; Whereas the Detailed Description part tends to be fully of statements like "in one embodiment of", which is almost certainly going to be the way Apple actually do it in their implementation. Sometimes what' s described in the patent may seem petty, or obvious, but in part becoming "one skilled in the art" is being able to look past the petty, obvious, parts to pick out the one gem, the one part that' s new or interesting.

The Basic Speculative Superscalar OoO Machine

Consider the basic out of order superscalar machine, as of say around 2000. Once we understand the general idea of this machine, we can consider all the many dimensions along which to improve it.

The CPU pipeline starts with a mechanism for deciding the address from which to load the next few instructions. This is non-trivial!
Normal'ish code contains about a branch every six or so instructions. The numbers vary depending on the type of code, but we can assume around a half of them are taken (meaning that the instruction pointer changes in some discontinuous way after the branch). So we are talking a change in PC around every ten instructions or so. If you're trying to run at around 8 instructions per cycle, you need to be able to handle a new PC discontiguous with the previous run of instructions, every cycle.
Clearly you cannot wait for the last branch instruction *executed* to tell you what the new PC is. Even under the best of conditions, that would result in a delay of maybe five or so pipeline stages between the fetch of a branch and when it is executed – five cycles while your fetch stage and most of your CPU is waiting, until it can jump to the next run of instructions. Clearly unacceptable!

And so we use branch prediction. The CPU guesses (based on past history) what the new PC will be at this point in execution (ie if the previous 64 times we encountered this branch, it jumped to address X, it's a good bet that it will do the same thing this time). For now let's ignore the details of how branch predictors work beyond accepting that they continually inform fetch of a (generally very good guess) as the address of the next few instructions in the execution stream.
The consequence of this prediction is that almost every instruction on the machine is executed in a speculative state, meaning that the machine needs to hold every result generated (including all stores) in temporary storage until the point at which it's clear that all the branches that affect a given instruction were correct, and so that instruction can Commit its state (that is, convert it from something temporary into something permanent).

We also perform instructions out of order. It's a somewhat surprising fact of real world programs that there's a lot of independence between successive instructions. This takes two forms. There is "immediate" parallelism, where successive stages of a chain of execution each require two or three independent operations.
If we bundle those together as a single "macroinstruction", we then tend to have chains of sequentially dependent macroinstructions, each about two to three instructions wide.
Your intuition might be that this means most code can only run about two to three wide, but that's not the case!

What most code looks like is that it consists of short chains of sequentially dependent macroinstructions (say 5 to 7 macroinstructions, 10 to 20 instructions long in total) which store their result to memory or a register, and that memory or register is not accessed until many (hundreds) of cycles later.
This means that while each sequentially dependent macroinstruction has to execute one after the other, you can execute many of the chains in parallel...
That sounds good but you need a variety of machinery to track which instructions are independent of previous instructions, and to track the program order of instructions so that as branches are resolved as correct, you know which of the instructions in program order now resolve as correct.
(This fact is why so many people's intuition about the value of superscalarity is so flawed. Most people

hone their assembly optimization skills on long stretches of sequentially dependent instructions; but such code is actually unrepresentative of most of what runs on a CPU.
This fact is also why OoO superscalarity works so well, whereas most attempts to create static wide machines have been problematic. All the pieces -- out of order, prediction, and superscalarity -- work synergistically. In particular most of these chains that are running in parallel come from different basic blocks [ie are separated by some sort of if() statement that the compiler can't see past] and so are impossible to aggregate statically.)

So the basic machine fetches instruction in a guessed instruction order, allocates resources to each instruction in order, throws the instructions into a large pool from which they Execute as soon as they can (ie once their Dependencies are satisfied), and then Retires the instructions in complete program order.
Retirement is the point at which all the guesses along the way are tested for correctness (and if they fail, we flush everything after the wrong guess and start again). Retire (because it happens in order) also undoes all the confusion caused by the OoO execution.

So let's think about the consequences of all this. What sort of state needs to be held as tentative until it can be Committed?
One obvious set of state is stores.
There are less obvious issues surrounding loads.
There are possible exceptions that were raised (eg by loads or stores that had invalid addresses).
And there are register values that need to be restored to whatever the programmer view of the registers was at the point of restoration.
(Even less obvious is the state that is used to inform predictors. You can update your branch predictors, or your prefetchers, as soon as the relevant instructions are executed. But you may be updating them with flawed data...
It turns out, another non-intuitive result, that on the data side, eg for data prefetch, this is mostly not a big problem, whereas on the instruction side, if you want quality accuracy for both your branch predictors and your instruction prefetchers, you need to ensure that they are not polluted by incorrectly speculated paths. [You also need to ensure that they are not polluted by interrupts, which throw in behavior that doesn't actually represent the flow of control you are trying to model.])

So we need a variety of structures to keep track of all this. Traditionally the largest of these structures is known as the ROB. This stands for Reorder Buffer which is not an especially helpful name. A better way to think of it is as Retirement Buffer, and to think of retirement as the point where *two* tasks are performed
- every instruction is given its last chance to either Commit its results to programer state (ie we agree that the instruction is not speculative at this point, and has raised no problematic issues like an exception)
- all resources that were allocated to the instruction can be returned to the machine for reuse.
(We can ignore the second point for now, but will return to it in time.)

So the basic flow of instructions (and remember, under ideal circumstances, each of these stages is dealing with around 8 instructions during the same cycle, and all stages are happening in parallel on a

different 8 instructions!) is:
• Fetch (all the branch prediction machinery)
• Decode
• Map
• Rename
• Coarse Scheduling
• Fine-grained Scheduling
• Execution
• Retirement
• Commit

What do each of these do?

Decode transforms instructions from their representation in memory (ie 32 bits) into something that's more convenient for the machine. Of interest to us, at this stage pairs of instructions may be fused, or meaningless instructions (mainly NOP) can be thrown away as irrelevant to the rest of the pipeline. NOP might seem a special case, not worth special treatment. Why bother with an instruction that does nothing, and why bother to treat it efficiently?

In the first place, much of the low-level machinery of modern operating systems is based on dynamic linkers and position independent code. This code is created in multiple pieces (separate compilation) and joined together by a linker. This joining process (the details of creating code that is position independent and can act as a library that can be called simultaneously by multiple programs) requires that all the calls in the code (app or library) that look like they might cross from one piece of code to another need to have a particular structure that might consist of two or three specific instructions. But when the linker actually stitches the code together, many of the calls that seemed like they might need to be "international" are in fact only local. Hence the two or three instruction slots that were reserved for an "international call" to a separately maintained piece of code, can be filled in with a local call (single instruction) and one or two NOPs.
Thus real world code contains a surprising number of NOPs, so why not make them as cheap as possible?

ARM also defines a family of HINT instructions (basically NOPs with different bits set in the instruction) might act as various hints for prefetching, branch prediction or other such control. A machine may understand a particular hint, in which case it will be treated as a real instruction, or it may not understand this particular hint, in which case it will be treated as a NOP and just ignored.

Map handles register renaming. (Yes the naming seems wrong, be patient.)
Recall that part of the machinery of speculative OoO is that the logical registers of the machine (the programmer visible registers, let's say 32 for ARM integer registers) are mapped onto a much larger pool (hundreds) of physical registers.
The idea is that at any particular time in the CPU, there is a map saying that logical register rN is

mapped to physical register pM.
So consider an instruction like

```
ADD r2, r1, r0
```

This instruction has two input registers (r0 and r1) so we need to consult the mapping table to figure out that r0 is currently mapped to physical register p7, and r1 is mapped to p45.
The add also takes a destination register, r2, so we need to create a new mapping for r2 (a new mapping is created every time a register is overwritten). This "single-write" rule means that the physical register can holds temporary state as long as the instruction is speculative; no other instruction is allowed to reuse a physical register until we can be absolutely certain that the temporary result it stores won't ever be needed again.

This concept of renaming registers should be familiar, and you should take some time to think about exactly how it operates, how the mapping table would be updated, when it is safe to reuse a register, and so on. But the most difficult part of the problem is one you probably didn't think of!
Remember, the machine may have to remap (source registers, plus allocation of new physical registers for all destination registers) up to eight instructions in a cycle. The tricky part of this is that often the same registers are used in many of these instructions. For a trivial case consider just the two instructions

```
ADD rA, rB, rC
MUL rD, rA, rF
```

The rA physical register must be allocated for the Add, before the mul is handled because the physical register looked up in the mapping table for rA (and rF) must reflect the mappings *after* the add instruction.
So you can't just remap all eight instructions independently! You have to figure out the successive name dependencies between all the instructions and treat that appropriately. That's the most difficult job of Map, and why it has a separate pipeline stage. It's an interesting fact that Apple is very clear, in multiple patents, that they use this separate Map stage, whereas most other companies do not mention such a stage. Being willing to move the most difficult part of the job to a separate stage may be part of why Apple has been able to maintain such spectacular CPU widths?

Rename is the traditional name given to a stage that should really be called Allocate. This is the stage where each instruction is given the resources it requires to perform its job as part of a speculative OoO machine.
What sort of resources need to be allocated?

The most basic is a slot in the ROB. The ROB is a queue of instruction in program order (this order is speculative, as best can be guessed by the branch predictor, but assuming that's correct, instruction are stored in the ROB in program order with no OoO weirdness). Each cycle the instruction that have just arrived in Rename are allocated a slot at the end of the ROB, while (as a separate stage) the instructions at the head of the ROB are tested to see if they have completed, and completed correctly.
If the instructions at the head of the ROB have completed correctly their resources are returned to the system and the head of the ROB queue moves down a few slots.

If they have completed incorrectly, corrective action is taken (maybe flush for a branch misprediction, maybe call into the OS for an invalid load or store address).
If the instruction at the head of the ROB has not completed, it remains at the head of the ROB until it completes.

The consequence of this is that suppose an instruction begins to be executed that can take a long time (say a square root). This instruction may stay at the head of the ROB for many cycles, while other easy instructions after it get their slots in the ROB marked as complete. By the time the square root completes, there may be 80 instructions directly after it in the ROB that have all been marked complete. The CPU will only then start retiring instructions from the ROB, as fast as it can. And it will keep retiring as fast as it can until either the queue runs dry or it hits another instruction that isn't yet marked as complete, at which point it will again wait until the head instruction is marked complete.

The most extreme version of an instruction that takes a longtime to complete is a load that misses all the caches and has to go to DRAM. This can take hundreds of cycles. Since the load cannot exit the ROB until it completes, instructions pile up behind it. At some point every slot in the ROB is full (or some other resources has been used up), and the machine halts until the load completes.
Thus the primary significance of the ROB's size is that it represents how well the machine can cope with a load that misses to DRAM. In time we will explore exact numbers, but assume a CPU that is 8-wide with a ROB that can hold 640 instructions. That would mean that if a load blocked the head of the ROB, the OoO part of the CPU could keep processing instructions for at least 80 cycles (assuming a full 8 instructions can be executed every cycle which is probably a little optimistic). That's good enough that the machine will not have to halt on a miss to L2 cache (around 15 cycles for M1) and will usually cover the delay to the System Cache (around 90 cycles for M1).
But when this 640-entry ROB machine misses to DRAM, (taking about 100ns, so about 300 cycles), eventually all the slots in the ROB will be filled (waiting for the head of the queue, the load from DRAM, to retire). Rename will not be able to allocate a slot to its instruction so they will not move down the pipeline. So the instructions in the Map stage will not be able to move into Rename, those in Decode will not be able to move into Map, and so on. At this point no new instruction can enter the machine until the load completes.

In addition to ROB slots, there are other resources required by different instruction types.
- Any instruction that generates a result needs a destination physical register to hold the result, and traditionally that would be allocated here (Map decided on an appropriate ID for the physical register, but the actual allocation of the register could be delayed till this stage.)
- Load and Store require slots in the Load and Store queues (for reasons we will describe).

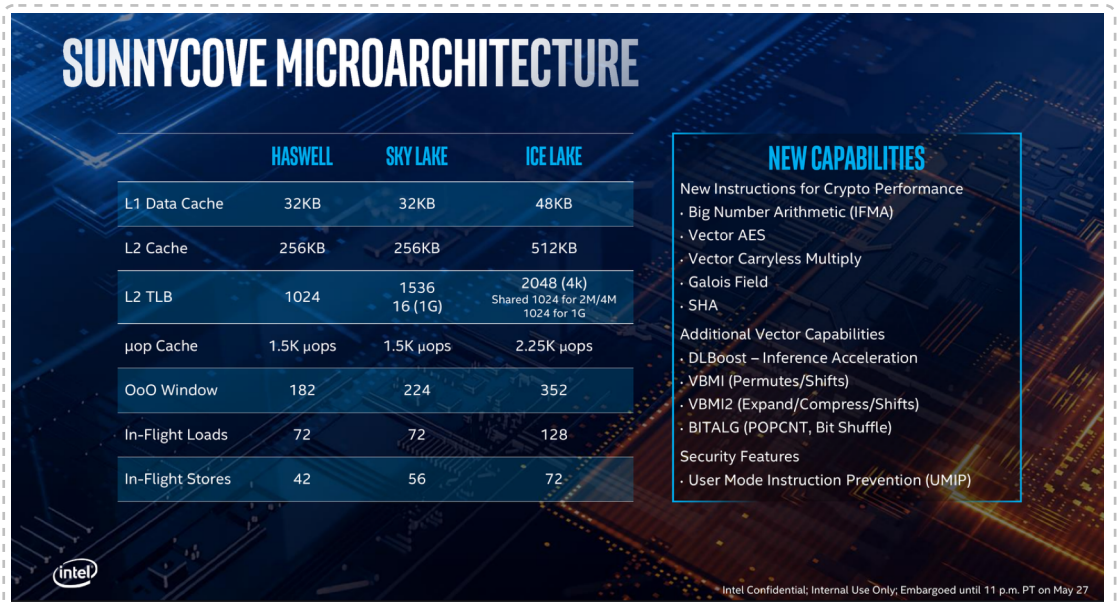
If any of these resources (physical register, load slot, store slot) are unavailable, again the the flow of instructions halts at this point, until some instructions in the ROB retire, and release the appropriate resource.

The extent to which execution can keep going after a load misses to DRAM depends on the size of the ROB. But the ROB is essentially just a queue, a low power structure that can easily be made larger. So why not make the ROB thousands of entries in size, so that we can sustain load misses all the way to DRAM?
Because a large ROB is no help if we run out of other resources along the way and so our machine grinds to a halt when only a few hundred ROB slots are occupied...

The problematic resources as far as Rename is concerned are the physical register file and the load and store queues (usually referred to as a single object, the LSQ). Both of these are large in area, power hungry, and difficult to grow .(Even if you are willing to pay the area and power costs, as they grow larger they grow slower, and if they become slower than can be accessed in a single cycle, performance falls off a cliff).
So you generally grow the physical register files and the LSQ as large as you can (given your area, power, and clock budget) and then scale the ROB to a size that seems to make sense given these resource limits.

You want LSQ to be as large as possible because, while the head of your ROB is blocked behind a load that is missing out to System Cache or DRAM, the instructions behind that load might consist of a large number of other loads (most of which hopefully hit in L1), or stores, or instructions that write a result to a register, and you'd like to do as many of these as possible (hundreds if necessary). Given that you could have (as we said, say 640 instructions piling up in the ROB this suggests that you might want to have access to many hundreds of physical registers, and perhaps a few hundred load or store queue slots. Those are large numbers!
Compare with the competition:

In[ ]:=



We see that Intel are running at a ROB of 352, with a load queue size of 128, and a store queue size of 72, also 180 integer physical registers and 168 fp physical registers.

(More details here if you want: https://www.hardwaretimes.com/intel-sunny-cove-vs-amd-zen-2-core-architectures-10th-gen-ice-lake-vs-ryzen-3000/)

The above, as I have described it, is that traditional role of the ROB, along with the constraints imposed by various resources. As we will see, one reason Apple can do so much better is that they substantially rethink this traditional design.

Scheduling. Instructions are processed in-order up through Rename (ie Allocate). After this they are placed in a scheduling queue.
The point of the scheduling queue is to provide a buffer until the “resources” required for instruction execution are available.
Every instruction describes what it needs to execute. Some of these resources (eg ROB slot, or destination register) have already been discussed, but to execute the instruction also requires its data inputs, and an execution unit. Consider, for example,

```
ADD rA, rB, rC
```

This requires an execution unit that can perform ADD, and the data value for rB and rC, which may not yet have been calculated.
So the instruction sits in the scheduling queue and, essentially, every cycle the scheduler checks "has rB become valid? has rC become valid? is an adder free?" Once all three are true, then the instruction gets moved on to the execution unit.

A scheduling queue is another very area intensive and power hungry structure. Instructions are moved into and out of it at random places, and testing (for whether the instruction can now execute) has to be redone every cycle for every instruction in the queue.

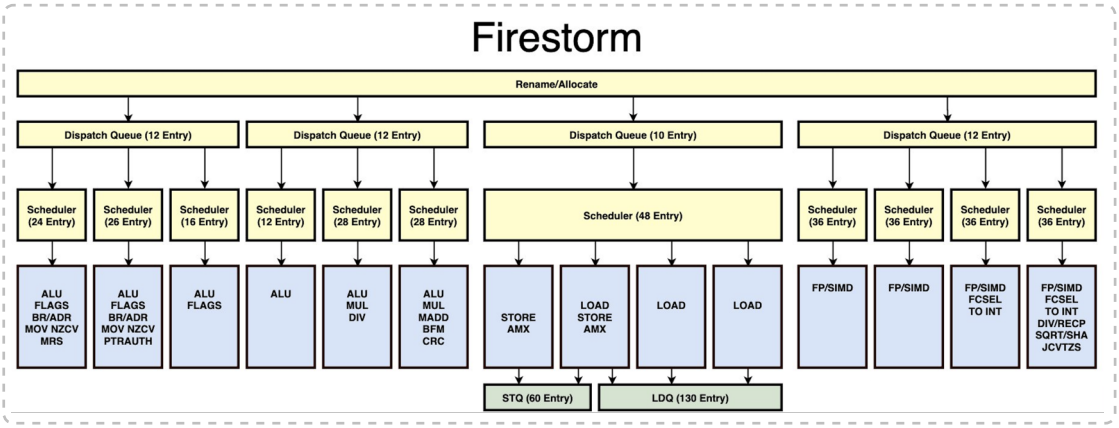
The scheduling queue is yet another constrained resource in the CPU. In any cycle, an instruction may not be able to execute because its dependencies have not yet been calculated, or an execution unit may not be available. It is possible for more and more instructions to pile up in the scheduling queue until it is full, at which point, once again, yes, everything grinds to a halt until one of the instructions in the queue has all its requirements satisfied and can be fed to an execution unit, freeing up its slot in the queue.

There are a variety of technical details of how exactly one might design a scheduler queue to try to reduce the power and area. But we are interested in a higher level design issue:
Intel has traditionally used a single large scheduling queue, which is expensive for the reasons given, but can be used by any instructions.
Almost everyone else uses multiple scheduling queue (for example maybe one for integer operations, one for load/store, and one for FP). This allows each queue to be shorter (lower area, lower power, easier to make it fit cycle time constraints), but it can mean that your integer queue has filled up, while your FP queue is sitting empty, and there's no way to use that FP queue space.

One can then go to the opposite extreme of, instead of one queue for integer operations, one has say a

separate queue in front of every integer unit; and this is what Apple does, they have multiple functional units (eg six integer units, four FP/SIMD units) with separate queues in front of each. Each scheduling queue may sound small (36 entries compared to say Intel's 97) but when you sum them all up, Apple has many more scheduling queue entries. Here's a diagram (edited slightly, from https://twitter.com/-dougallj/status/1373973478731255812)

In[ ]:=



You can work around the queue imbalance problem to some extent if you have a two level scheduling system. The way this works is you have a "coarse buffer" that accepts instructions out of Rename, and then, as slots open up, moves them to an appropriate scheduling queue in a load-balancing fashion. In the diagram above, this is called the Dispatch Queue (though that's incorrect; a better name would be Dispatch Pool, because these pools do not attempt to preserve instruction order, unlike queues).

How many scheduler slots does one want? Lost, sure, but how many exactly? What do more slots get you?
Remember the goal of an 8-wide OoO machine is to execute 8 instructions every cycle. What sort of things prevent that?
Issues that we will cover later include
- ensuring that 8 instructions are available every cycle,
- that branch prediction is rarely incorrect, and
- that loads can usually be found in cache.
Let's ignore the fetch and branch issues for now, and consider a given stream of instructions.

What most instruction streams look like is, first, at a rough level, about 15% branches, 10% stores, 25% loads, leaving 50% integer operations.
FP tend to pull stores and branches down some, and integer quite a bit, leaving space for fp operations.
You can see the sorts of analyses people do here: (2018) https://tosiron.com/papers/2018/SPEC2017_IS-PASS18.pdf *A Workload Characterization of the SPEC CPU2017 Benchmark Suite.*
So a standard instruction stream will consist of branches (we ignore, that's Fetch's job) stores (we ignore because they are fire and forget) and a whole lot of loads and integer instructions.

Mostly these instructions occur in small clumps (they are separated by a branch every five or six cycles,

and there are limits to how much a compiler can structure code across branches), with maybe two or three independent instructions, followed by another two or three instructions that each dependent on the previous instructions. To make this run fast we need to be able to queue up instructions that cannot yet execute because they depend on results from prior instructions (which may be executing, or in turn waiting for earlier instructions). And we need to be able to look in this pool of instruction to find every cycle at least eight that are ready to execute. The larger your pool of instructions waiting to be scheduled, the more instructions you can have look over every cycle, hoping to find eight or more that are executable.
People frequently confuse ROB size with scheduling queue size.
- As a practical matter, ROB size determines *how many instructions the machine can continue to execute after a load misses to DRAM*.
At any given time, many to most of the instruction the ROB have completed, they are simply waiting their turn to be Retired because Retirement happens in order.
- As a practical matter, Scheduling Queue size determines *how far ahead in the instruction stream the CPU can look for instructions to execute that are independent of the instructions currently executing*. At any given time every instruction in the scheduling queue has not yet executed, and it is waiting to execute as soon as everything it requires (dependency data and execution unit) becomes available.
So if we assume as rough rule that a dependency chain is about ten instructions long, and we'd like to sustain eight instructions per cycle, we'd like our Scheduling queue to be at least 80 instructions in size; and of course, like all resources, unevenly balanced situations may arise (much longer than usual dependency chains, for example) so we'd like as much larger than 80 as our design, power, area, and timing allow. Apple's techniques (many, shorter, Scheduling Queues, kept balanced by Dispatch Pools) allow them to do extremely well, achieving a lot better than Intel (looking further ahead in the instruction stream, at lower power, while issuing many more instructions) than Intel.

Be aware that the language related to scheduling is somewhat confused.
IBM (and Apple) use the terminology that moving instructions into the Scheduling queue(s) is called Dispatch, and moving them out of the queues (to an execution unit) is called Issue. But Intel tends to use those words with the reverse meaning. So generally don't get too obsessed about which is used for what, look at the context to understand exactly what is meant.
Also, Apple and IBM use the term Reservation Station for what I have been calling a Scheduling Queue, because this is the Intel, and thus the internet default, terminology.

An interesting point here is how are dependencies tracked. I have described this as the instruction
`ADD rD, rA, rB`
depends on rA and rB; rA is mapped to physical register pM, rB is mapped to physical register pN; and the instruction cannot be Issued to Execute until pM and pN are marked valid.
This seems so obvious and natural (and appears to be how Intel do it) that most people think it's the only way to do things. But there are alternatives.
In particular rather than tracking the dependency on rA through physical register pM, what if you track a dependency on the instruction, call it iR that will calculate pM? So we say that the ADD depends on

instructions iR and iQ, and won't execute until those have both completed. Logically this is the same as a dependence on registers, but how would you implement it, and why?
Implementing is easy – every pending instruction has a unique, unchanging number, namely the which slot in the ROB it occupies, and which is allocated early in the pipeline. So we can use that as an instruction identifier.
Why do things in this way that seems unnatural? Well, where do instructions get their operand values from? From the register file, sure, that's what you think, but how exactly is this done?
There have been many possibilities but let me just describe two.
One possibility is to copy the value from the register file into scheduling queue when the instruction is placed in the queue. This assumes the value in the register file is already valid, and requires extra storage in the scheduling queue. It seems (IMHO) kinda pointless except in light of the historical evolution of the technology where it was one of these easy extensions of the even earlier method.
Another possibility is to read the value from the register file as the instruction is issued for execution. That gives a little more time for the value in the physical register file to be valid, and doesn't waste space replicating the value in the Scheduling Queue.
But both these suffer from the question of: what if the previous cycle created one or both of the register values required by this instruction? Do I have to wait a cycle for the result(s) to be written to the register file before I can read them back out of that same register file? That's clearly not ideal, and so we have the concept of the bypass bus: the busses on which results flow from each execution unit back to the register file can be read by each execution unit, so that the operand is immediately available without having to read it from the register file. And it's another of these facts about real code that most instructions feed their results into an immediate successor, and most results have a very short lifetime before their register is overwritten.

All this means that, in fact, when it comes to considering implementation rather than considering the programmer's viewpoint, it's closer to optimal to say "my two inputs are instruction iR and instruction iQ" than to say they are "register pM and register pN". Of course there are still long-lived inputs that come from registers, but the common case (and the desirable case, for latency and power) is to grab as many inputs as possible directly off the bypass bus.

Apple patents explicitly say that registers are read at Issue time, not early; and they strongly suggest that scheduling is based on instruction numbers (called SCH#'s); examples of the latter are https://-patents.google.com/patent/US9940262B2 or https://patents.google.com/patent/US8555040B2.

We've described the obvious instruction dependencies on previously calculated registers. But there are more subtle instruction dependencies. For example consider a load instruction. The obvious dependencies are the registers used to calculate the load address. But the next stage of execution is to load the result from the L1D cache. What if the data is not in the cache? This gives rise to a situation where you can't clear the instruction (it hasn't fully executed!) but you don't want it hanging around, blocking the execution unit until the value is available. This gives rise to a concept known as *replay*. Every CPU vendor does this differently, but the common themes are that

- you need to be able to keep the instruction in the Scheduling Queue for a few more cycles (so you can't automatically just move instructions out of a Scheduling Queue, not until their execution sets a flag saying "OK, we're done, it's all over")
- you also need to be able to mark the instruction in some way as "try again later". In the dumbest sort of Replay, you just try again a fixed time later, say every 5 cycles. More ideal is to add a new type of dependency of some sort to the instruction, so that it doesn't try again until the additional dependency is satisfied. Now that sounds good in theory – but given what I have said about dependencies being either based on physical register being marked valid, or instructions in the ROB marked as completed, how exactly will you implement these additional new types of dependency (like "cache line has been deposited in the L1D")? We will eventually see one possible answer.

Execution: There's actually not much to say about execution. Execution resources tend to be clustered into a "unit" which can perform a variety (but not the full range) of operations.
Our current best knowledge of the M1 is that it contains 14 units as described here: https://-dougallj.github.io/applecpu/firestorm.html.
In one sense execution units are the point of the CPU, and the numbers that are associated with execution units (what's the latency of operation X, how many of operation Y can I perform per cycle, what operations share what units [because if divide and multiply share a unit, I can't do both operations in the same cycle) are something that has traditionally been obsessed over by a certain type of enthusiast.
But in another sense, execution units are just no longer where the action is. Yes, you want as many execution units as possible, and you want each operation as fast as possible; if FP multiply is reduced from 5 cycles to 4, that's a good thing. But on a CPU like the M1, very little of the performance of most code can be understood by thinking about the execution units. The M1 is fast because it is extremely wide, and can run extremely out of order, not because each execution unit behaves in some way that is exceptional compared to other ARM or to x86.

One amusing way to capture this fact is to ask what is meant by the "width"of a machine.
- Amateur enthusiasts will reach for the number that looks most impressive, which is generally the Issue number, ie the number of instructions that can be fed to an execution unit. For the case of the M1, there are fourteen units, each has an independent scheduling queue, so under ideal circumstances, the M1 could fire off fourteen instructions, one to each instruction unit, in the same cycle.
- Serious enthusiasts will answer this as the maximum sustained possible throughput. In the case of the M1, this is 8, not 14.

Why this difference?
In the old days a machine that was, say, 4 wide, was built with each stage 4 wide, so fetch would deliver four instructions, decode would decode 4, there would probably be a single scheduler queue that could make a maximum of 4 scheduling decisions (even though there might be 5 or 6 execution units), and retire could likewise remove 4 instructions from the head of the ROB per cycle.
This sort of design is still based in the original microprocessor days where a single instruction moved

from one execution phase to the next; with the implicit assumption that the machine works by moving blocks of four instructions from one stage to the next per cycle. But this has become an ever less appropriate model as CPUs have ever more transistors available.
What you should think of is that a CPU consists of a number of jobs to be done, with queues connecting each job to the next. So Fetch does its job and dumps fetched instructions into the Instruction Queue. Decode/Map/Rename do their job, then dump instructions into (multi-level) Scheduling queues. After execution instructions proceed to Retirement via the ROB queue.
Under ideal circumstances, each of these queues should be extremely dynamic (they should constantly be growing very full, then shrinking to empty). The point of a queue is to buffer between stages, so that if any stage is temporarily slowed down (eg Fetch misses in the I-fetch cache) the next stage can keep going for a while just by draining its feeder queue.

With this sort of design philosophy, the width you make each stage is no longer constant because you have no vision of a single block of four instructions proceeding together through each stage of the pipeline. Rather you make each stage as wide as it can be, given power budget and timing. A wider stage will drain its feeder queue faster, ensuring that that particular stage is never the bottleneck.
This is most obvious in the case of the Execute stage. It is fairly easy to run this extremely wide (14-wide for the M1, as we see) because the Scheduling queue and execution units are essentially independent. (There are technical details, like the result buses between units, that mean we can't go absolutely wild, but we can go fairly wild).
Another stage that can easily be made wider than the sustained width of 8 is Fetch. As we'll see when we discuss Fetch, Fetch can probably pull in a maximum width of 16 instructions (one cache line) in one cycle. But in most cycles that's not possible because the basic block [distance to the next branch point] is shorter than 16, or because the basic block runs over into the next cache line, and only one I-cache line is accessed per cycle. Thus to sustain an *average* of 8 in the face of these frequent problems means you want to be able to pull in a lot more than 8 when you have the chance.
Likewise it is ideal to be able to retire wider than 8 so that once an instruction that has been blocking the head of the ROB retires, you release any resources being held by successor instructions as rapidly as possible.

Another way to look at this is that, in the older, resource-constrained days, a reasonable way to design a CPU was to target the mean properties of code, to design around the 15% branches, 10% stores, 25% loads, 50% integer operations already mentioned. But once you have the resources to do better, you should try to deal not just with the average behavior of code but also with extremes. On average, yes, 25% of the instructions are stores, but this is not the same thing as saying every fourth instruction is a store; in fact you may encounter long runs of stores (or load+stores) followed by almost no stores. So a better design target tries to hit not the averages but a metric that captures this variation. Once again this is where design as a sequence of connected queues really pays off. By providing deep queues between every conceptually different stage, Apple can rapidly shunt instructions into the appropriate queue, where they can wait to eventually execute while not blocking other instructions. The goal should not be to flow the same N instruction instructions from the beginning of the machine to the end;

it should be to have enough reserve capacity in *every* queue that all reasonable surges (a run of 50 successive FP instructions?) and dips (no instructions fetched for 8 cycles while we wait for a new cache line from L2?) can be handled without pausing.

Retire: We've pretty much covered everything related to Retire in the earlier discussion. Retire as a general concept refers to three different sort of ideas.

- Completion. This means that the sum has been added, the load has been delivered from cache, whatever it is, the result is available for someone else to use. Completion mainly means that a flag gets set in the ROB entry (saying this instruction has generated a result), but under the traditional design all the resources acquired by the instruction at Rename are still held onto.
- Retire is the point at which the instruction is checked in-order, to make sure that everything went correctly; whatever speculation occurred was correct, no exceptions or faults occurred, etc.
- Commitment refers to somehow moving the (now non-speculative and absolutely correct) state from speculative storage to non-speculative storage.

The exact order in which these tasks are done has tended to be very variable, and mostly undisclosed because most of what gets written up about CPUs tends to be to help developers optimize code, and pretty much nothing on this backend has any relevance to code optimization. It has tended to be treated as boring cleanup work that has to be done after the main event of the instruction execution. This is shortsighted, and the M1 shows why.

Note that there are distinct tasks to be done here. And whenever you have distinct tasks to be done, you can disaggregate them into distinct data structures with distinct timings. The way Apple has done this with the ROB and the structures surrounding Retire and the release of resources is, as we will see, substantially different from anyone else (though has some similarities to IBM).
- Ideally you'd like to release resources as soon as an instruction completes. But that may not be possible, especially if the resource is the speculative storage that holds the instruction result, because we can't go non-speculative until we pass through Retire. Commitment may be feasible in the same cycle as Retire, or it may be an additional cycle (usually invisible because almost nothing depends on it).
- Moving to non-speculative storage (ie Commit) for store instructions takes the form of moving instructions from the Store Queue to the L1D cache, but while (for x86) there are good reasons to do this as fast as possible after Retire, whereas for ARM (and POWER, in general for weakly ordered memory models) there are good reasons to delay this write out for some time.
- For instructions that write to registers, in principle Commit could be copying the result from the physical (speculative, out-of-order) register file to the architected (non-speculative) register file; but this model seems to have fallen out of favor, Instead now Commit means only flipping some bits associated with the physical destination register of the instruction; so that there is no single register file that represents the architected state of the machine at a given time, but you can recover this state by going through the physical register file and the logical to physical mapping indexes.

In particular, it is *possible* (if you're willing to make the effort) to
- Commit stores to L1D (thus giving early release of LSQ entries, and early informing other CPUs for multi-threaded code) as soon as a store becomes non-speculative. Even if the head of ROB is blocked by a long-executing instruction, if there are no pending branches (branches not known to be successfully predicted) between the head of ROB and the store, then there is nothing preventing the store from being committed. Apple does this with the M1.
- Early release registers. If a physical register's architected value has been overwritten, and there are no dependencies, and the register is no longer speculative (same as above, no pending branches) you could reuse the physical register. Apple does *not* do this (nor, as far as I know, does anyone else). Like every innovation, it requires some additional book-keeping, and I expect that book-keeping machinery is not present in the M1 -- but is an obvious extension to future CPUs.

So the basic flow of instructions after decode is
- [in-order] Rename (which should be considered more generally as "Resource allocation")
- [likely mixed order] Dispatch (move the instruction together with identifiers for all its resources into a scheduling queue)
- [OoO] Execute the instruction (which will involve lots of waiting around in various queues while predecessors execute)
- [in-order] Retire the instruction

Design Principles

Once you are committed to designing an OoO CPU and want to make it go faster, what are you are options? You can understand what Apple has done better if you appreciate some general design principles.

Faster

Obviously three trivial mechanisms to go faster are "run at higher frequency", the not -exactly-equivalent "generate less power per operation", and the easy "use more cores".
These are valid mechanisms, but are primarily the province of the silicon process (TSMC) and specific circuit design techniques. Apple has plenty of patents in these areas, specific ways to run some low-level design element at lower power, or at higher frequency; but these are not the level which we are going to explore.
An important qualification is that these are valid *within reason*. In particular the optimal frequency is a contentious point. Both Intel and IBM overshot the point of sensible frequency (Intel with the Pentium 4, IBM with the POWER6). It's unclear that either fully learned their less. Higher GHz is always easy to justify -- it sounds cool, it sounds heroic, it's easy to market. But it's a bad path to go down for many reasons.
- Higher GHz requires physically larger (substantially larger) transistors and standard cells. This means substantially lower (like half to a third lower) transistor density than if your design was based primarily on the lowest power transistors. And designing with only half the transistors otherwise available is a

severe handicap. Are you sure the additional frequency (say 5GHz rather than 3GHz) is worth the IPC cost?
- Of course higher GHz uses more power – a lot more power. Once again, is this a sensible tradeoff, if you could hit the same performance at lower GHz but higher IPC?
- Higher GHz requires much more human intervention in the circuit design. It's like choosing to write your code in assembly rather than in Swift. With the same sort of consequences. Working at a very low level makes you focus on very local optimizations, but it's hard to see the big picture to try for more powerful optimizations. Doing anything takes far far longer, which less help from tools. And it becomes terrifying to contemplate a total restructuring of your design.

More

Only slightly less trivial is "do more of what you're already doing". Use more cache! Provide more physical registers! Make all queues deeper! Go wider!
Even apart from the practical issues associated with this (using "more stuff" always costs more power and increase clock cycle length) it's not generally appreciated how impotent this design strategy is when pursued in isolation.
This Intel paper, (2019) https://arxiv.org/pdf/1906.08170.pdf *Branch Prediction is not a solved problem*, is concerned with a different topic, but the very first graph shows the important point. Even truly massive (32x) scaling of OoO resources delivers a substantially less than 2x performance increase.

In[ ]:=

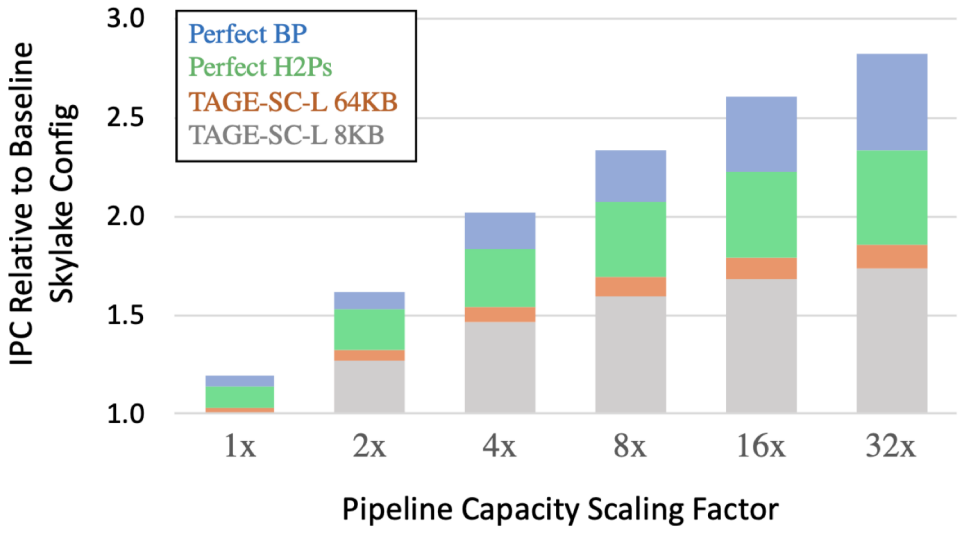


Fig. 1: Without better branch prediction, scaling the pipeline capacity of an Intel Skylake configuration will produce diminishing returns in single-threaded IPC for SPECint 2017.

The point is not that more resources are bad, it's that blindly increasing resources is not enough, not even close. You need to understand everything in your processor that is holding you back, you need to

attack *every single one* of the pain points, and you need to keep redoing it every few years as many more design options open up with more transistors.

This is, in a way, very depressing news for CPU designers. As you can imagine, designing a CPU from scratch is not easy! What you would prefer to do every year is make some tweaks, provide more resources. But not rewrite from scratch!
Samsung have given us a rare look into the sorts of annual evolutionary updates to a CPU in (2020) https://conferences.computer.org/isca/pdfs/ISCA2020-4QlDegUf3fKiwUXfV0KdCm/466100a040/466100a040.pdf *Evolution of Samsung Exynos,* but it's clear if one looks at any CPU family that this sort of unambitious evolution is the norm.
Apple had the good fortune (or good sense) to begin with a design that looks like it's from scratch as of the mid-2000s (when many of these good ideas were already known), and to design with the expectation that what future processes would give them would be many more (but not substantially faster) transistors; so their initial design incorporated a variety of flexible "communication channels" from one part of the pipeline to another, which allowed them in turn to insert various good ideas with each new design. Other machines that have not been designed with these communication channels in mind can see the value in many of Apple's ideas, but cannot easily retrofit them without substantial redesign. Or to put it differently, "more, but without new algorithms, doesn't get you much".
In other words while it is interesting to see, for every successive CPU in a family tree, how many more resources were provided of each class, you can learn this mainly because it's what's easiest to probe, not because it's what's most important. Far more important is any sort of modification in how this resource class is managed, and that most of what we will be discussing.

Guess Smarter

Much more interesting is more, and better, predictors. You should know (and should understand how it is implemented) that modern CPUs engage in aggressive branch speculation, and doing this ever better remains an active area of research. But many many other things are or can be speculated in a modern CPU:
- there is load-store address speculation (which guesses as to whether a load can be executed early, before pending stores),
- there is scheduling speculation (which guesses as to how long an operation, usually a load, will take, and schedules dependent instructions based on this guess), (2015) https://hal.inria.fr/hal-01193233/document *Cost-Effective Speculative Scheduling in High Performance Processors*.
- there are way prediction and drowsy caches (which are techniques for saving power rather than increasing performance).
- in the past Apple had predictors for whether loads might partially overlap with recent stores, or whether loads might be misaligned, though now both these issues are solved by more powerful techniques. They also have a patent (maybe still relevant) that’s essentially predicting how congested the NoC will be (and thus how long it will take for data being transferred from DRAM to reach the CPU).
In every case what you’re trying to do is discover a pattern that is common in real execution, and try to make that common pattern faster or lower power; at the expense of making uncommon patterns more

expensive.

Along with predictors one should track the *confidence* of a predictor. Suppose that recovering from a mispredict is cheap; in that case go ahead with the prediction even if it is low confidence. But if recovery is expensive, you may want to delay an operation until it is no longer being speculated, until you know for certain.

A new concept (which appears occasionally in the literature of the past fifteen years, but which does not yet seem to have been productized, including by Apple, is criticality, or the similar idea of urgency:
(2009) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.157.2426&rep=rep1&type=pdf
Criticality-Based Optimizations for Efficient Load Processing.
(Apple does use urgency in some recent NoC patents, so who knows, maybe we'll see it in the CPU soon?)

Of course closely related to predictors are the concepts of *prefetching* and *cache management*, both huge subjects.

Work Smarter

Most interesting of all, and very important is two, somewhat related design principles, which I will call *resource amplification* and *task disaggregation*. (It's interesting to note, as an aside, that while resource amplification is the common design pattern in almost every decent micro-architecture design paper, task disaggregation is far far more rare. It's something one sees all over the place in the Apple design – and almost nowhere else... Not in the literature, not in other designs.)

Resource amplification is about making existing resources (which are always in short supply!) go further.
For example you have access to a fixed size L1D cache. You may think that's the end of the story, but not even close! For example -- what algorithm are you using to replace lines in your cache? A better algorithm (which holds onto useful lines longer) will make your cache effectively larger. This is amplification – using better algorithms to make a small resource provide the value of a large resource (or, in a slight twist, providing a large resource that mostly costs the power of a small resource).
We will an astonishing level of this in Apple's load/store/TLB/cache pipeline, where Apple gets most of the performance of a 4-wide pipeline while paying the costs of a 1..2-wide pipeline.

Resource Lifetimes

It has been traditional in CPUs to Allocate resources in one place (in a pipeline stage called Rename) and to Deallocate them in one place (in a pipeline stage called Retire or Commit or something similar). Like many things, this was an obvious solution, and probably optimal at the time it was invented, but far from optimal today.
This pattern means that any resources that is allocated (think for example of a destination register) is reserved, and locked up unavailable for use, from the Rename stage forward. This is so even if the

register is the target of a very long latency operation (maybe a load that missed to DRAM), and even though the register is only required at the very end of this operation, at the point where the result needs to be saved in a register.
A similar situation holds, for example, with slots in the load and store queues.

Why was it done this way? The reason is ordering/dependency requirements. The pattern of register allocation and renaming establishes the true data dependencies that allow an OoO machine to reorder operations for maximum speed while still being correct; the ordering of load/store slots allows loads and stores to operate out of order while still ensuring that if a load tries to read data from a store address that is pending, but not yet executed, the load will delay until the store provides the data.

These seem like non-negotiable requirements, but not so! The trick is to realize that the allocated resource is performing two tasks.
One of these is expensive (storage, of an execution result, or of a load/store address), the other task is cheap (providing an ID that's ordered relative to other, equally cheap, IDs).

Turning this rough insight into an implementation is done via Virtual Registers, first explained here: (1999) https://upcommons.upc.edu/bitstream/handle/2117/101362/00809456.pdf *Delaying Physical Register Allocation Through Virtual-Physical Registers*
or Virtual Load Store Tags, described here (2007) http://citeseerx.ist.psu.edu/viewdoc/download?-doi=10.1.1.140.3533&rep=rep1&type=pdf *Late-Binding: Enabling Unordered Load-Store Queues.*
Apple and IBM both use Virtual Load Store Tags. No-one (not yet, as far as I can tell) uses the full virtual register idea.

So one can delay the allocation of resources, thus having them reserved for a shorter time.
An alternative is to tackle the other side, to try to free resources sooner. Once again, it has been traditional to deallocate at Retire because that's the point at which everything related to the instruction is over, but it is frequently possible to retire a physical register much earlier. (2004) http://pages.cs.wisc.edu/~rajwar/papers/taco04.pdf *An Analysis of a Resource Efficient Checkpoint Architecture* talks about many interesting things (including why one wants to keep increasing resources, especially via amplification mechanisms), but among other things, in section 4.3 it discusses ways to more rapidly reclaim physical registers.
Apple are using one (limited) form of early register reclamation: (2017) https://patents.google.com/patent/US10691457B1 *Register allocation using physical register file bypass.*

One slight variant of this idea (less resource amplification, more energy minimization) is how Apple manage both their registers files and their L2. Both of these are nominally large structures that could take lots of power and be slow. But in both cases they are in fact split into multiple independently managed smaller pieces. So, as much as possible, the machine runs using eg one quarter of the physical register file, or one third of the L2, with the rest powered down and taking no energy; the extra resources are only powered up and used once certain metrics indicate that they would provide real

Fusion

The most obvious form of resource aggregation is instructions that do more than one thing. So you pay the cost of a single instruction, at least for some stages of the pipeline, while achieving more than one result.

Much of the ARM64 instruction set provides *pre-fused* instructions that do this – a single instruction that performs one task (add, select, move, ...) with a small additional tweak (shift an input by a few bits, increment or negate the output, …). The trick, of course, is to ensure that the tweak is indeed so lightweight that it can be performed as part of the single instruction without compromising the design of the entire CPU.

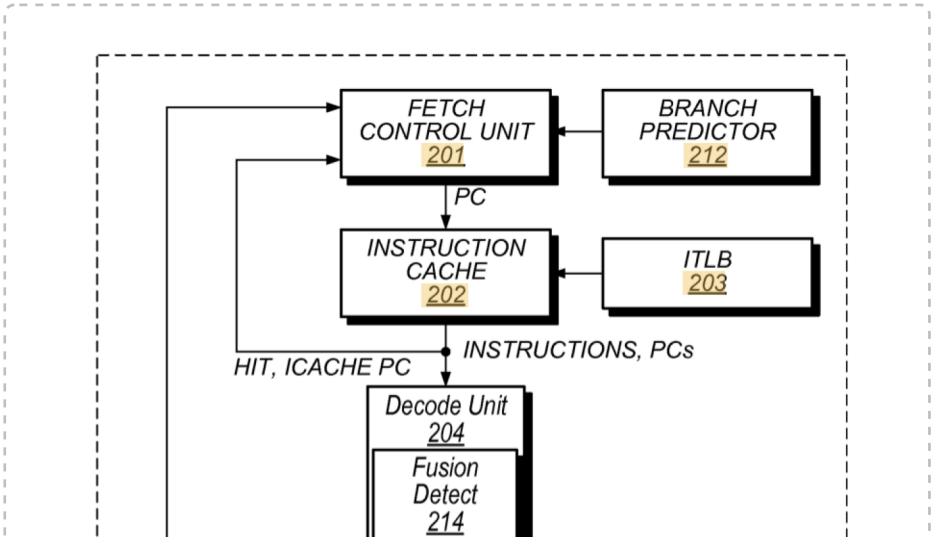
Alternatively, the decode stage of the CPU can fuse together common lightweight instruction pairs. x86 does this for compare+branch instruction pairs, and so does every high end ARM CPU, but many additional instruction pairs can be fused when it makes sense.
The idea of fusing instructions has been around in various forms for years: (1996) ftp://ftp.cs.wisc.edu/-sohi/papers/1996/micro.collapse.pdf *The Performance Potential of Data Dependence Speculation & Collapsing* is worthless for its performance projections, but is interesting in its categorization of the most common feasible fusion sequences. You'll see that the most common patterns (and the ones that have been implemented by everyone) fuse with a branch, and so only require a standard ALU; most of the patterns require a 3-input ALU, which we know how to design and which is known to be feasible in current cycle times, but which has (as far as I know) still not yet been implemented in generic form.
Right now what Apple does, though more aggressive than any other vendor, is still limited to the obvious pairs, as listed here: https://dougallj.github.io/applecpu/firestorm.html.
(For people who are curious, the reason some non-fused pairs are given is that they are, in fact, obvious fusion candidates at some later point:
`ADRP+ADD` is used to construct large offsets for loading global data, strings, floating point constants, and suchlike
`MOV+MOVK` is used to construct large immediate values (ie large integer constants)
`MUL+UMULH` (or `SMULH`) is used to multiply two 64-bit values to create a 128-bit value
`UDIV+MSUB` is used to calculate A mod B)

Industrially, instruction fusion is still somewhat unexplored territory.
On the public side, we mostly only know the fusion patterns that have been published by vendors (eg in ARM or x86 per-CPU optimization guides), or that have been guessed at and tested. It's possible there are more fusion pairs on the M1 not yet discovered.
Cases that are still open for exploitation (even by Apple) include
- A fused instruction pair may allow the second instruction to reuse some work that was done by the first instruction. There are a few ARMv8 pairs of crypto instructions that do take advantage of this, but it would be possible in theory for 128-bit multiply (`UMULH` giving the upper 64 bits, along with `MUL` giving the lower 64), or for the pair (`DIV+MSUB`) giving the quotient and remainder of an integer division, to likewise share work across the two instructions, which is why they are obvious test candidates.

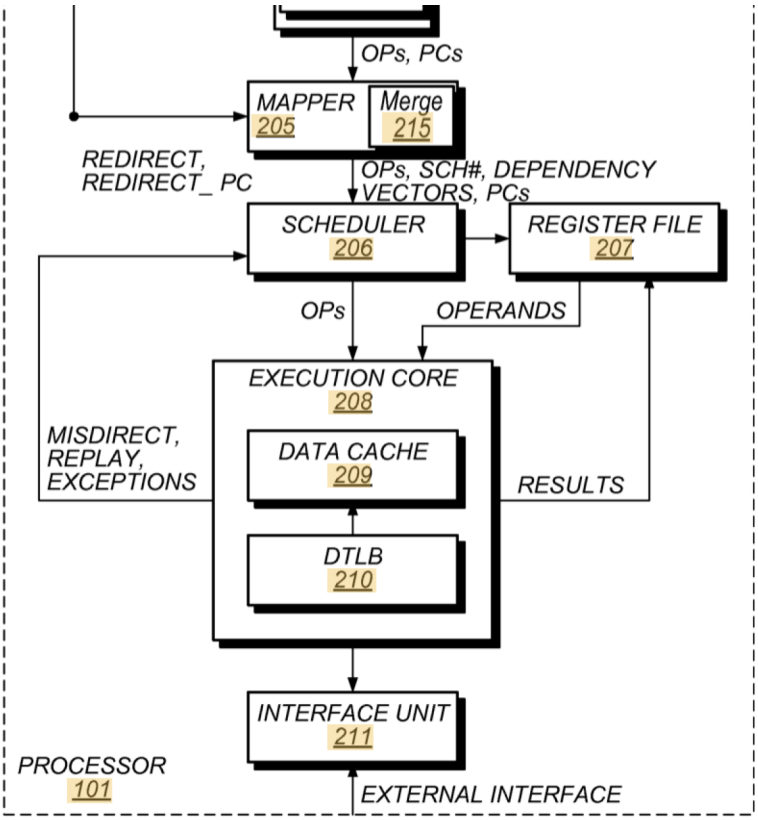
However it's not always easy! There is a tricky element to the latter two cases, namely that they return two results. So you're kinda asking for a new type of instruction that overwrites not one but two registers, and so requires the allocation of two physical registers.
When we get to register allocation, we will see why this is a non-trivial issue, but Apple is in a position to deal with it (they already have to allocate a register pair for Load Pair instructions). But if you look at the various cases that have been fused so far, they either involve branches (no second register) or involve overwriting an intermediate register, so avoid this issue.

- A different type of case is if the next instruction immediately overwrites a register, only a single register allocation may need to be performed. Consider for example
`add rD, rA, rB`
`add rD, rD, rC`; which could in principle fuse to
`add3 rD, rA, rB, rC`
This would require only one destination register allocation, and, for ARMv8, which already has integer instructions that take three source registers, would not require a substantial redesign of the internal data flows. Would it be worthwhile? The pattern is probably common, and if the two additions could be performed in a single cycle...? Of course multiply-add is already a pre-fused version of the same *sort* of pattern, where we remove the use of an intermediate register.

Instruction fusion is not a panacea; it only amplifies resources if there is in fact some sort of common resource whose utilization can be removed by the fusion.
And for the most part, Apple actually seems to consider fusion more as a way to remove one cycle of latency than as resource reduction. We have one Apple patent that covers fusion, namely (2013) https://patents.google.com/patent/US9672037B2 *Arithmetic branch fusion*, which discusses the pair arithmetic op+compare result. (The obvious cases of interest are `SUB+CBZ` and something like `OR+TBZ`). The patent includes these diagrams:
need to check the TBZ case -- dougall does not list it (neither does the patent) but it's an possible next step



In[ ]:=



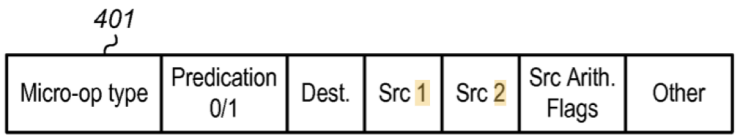
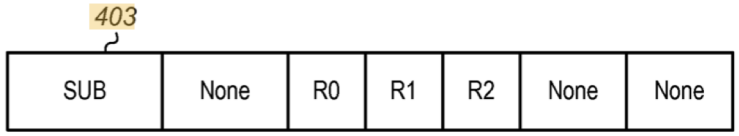


FIG. 4A

FIG. 4B



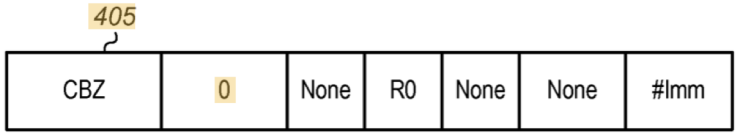
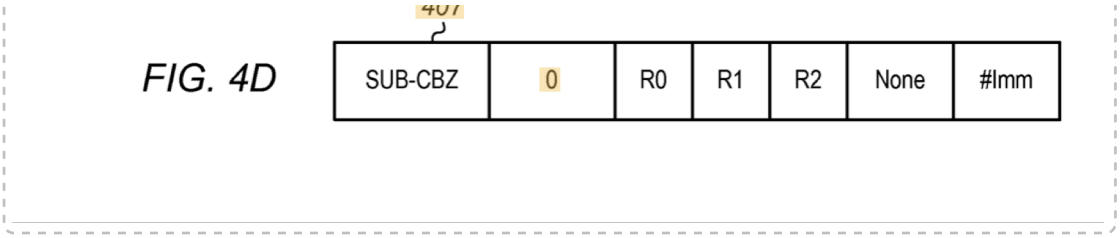


FIG. 4C



Of particular interest is that
- fusion detection occurs at Decode and is implemented at Map (essentially the Rename/Resource Allocation stage)
- it is implemented by changing one of the two instructions in the instruction stream as seen in the second diagram (from an ARM `CBZ` to an Apple-invented `SUB–CBZ` instruction) while the other instruction is discarded.
Consequence is that some resources (in particular a ROB slot) are already allocated; we see something similar in the way that `NOP` and NOP-equivalents also take up ROB slots though they execute at full 8/per cycle, presumably limited by Decode. However only one Scheduling slot is required, not two, and the result is available in once cycle, not two.

This implementation is not ideal, but still seems to be the implementation structure today. More ideal would be earlier fusion, something like
- mark instructions of possible fusion classes in pre-decode (ie when the instruction is pulled into the I1-cache from the L2)
- perform the actual fusion (ie tie the two instructions together, and delete NOPs, and other cleanup) before Decode, while the instructions are in the Instruction Queue.
Among other things, this would allow higher Decode throughput (a `SUB–CBZ` would count as a single Decode instruction, so in many cases Decode would be performing the work of nine or ten "current-style" Decodes in a cycle); and we'd save a few ROB slots.

Note also the power implications: that is, the fusion is repeated every time it re-Decodes and re-Maps. Power can be saved slightly with a loop buffer, and one could imagine a CPU that, at the point where it decides to utilize the loop buffer, makes a second pass over the instruction stream looking for a wider collection of possible fusion pairs (or the similar sort of exercise for CPUs that utilize a micro-op cache). However, in addition to energy savings, what I suggested above about doing this work as early as possible in pre-Decode and in the Instruction Queue also allows the fusion to be made more complex (spend more time looking for candidate pairs) while it benefits both straight line code and can amortized over a wider range of loops.

We do know that Apple performs pre-decode. Examples include:
- an (interesting, but obsolete) case here, related to 32-bit ARM THUMB processing: (2013) https://-patents.google.com/patent/US9626185B2 *IT instruction pre-decode*
- (2014) https://patents.google.com/patent/US20160011875A1 *Undefined instruction recoding* detects all the possible undefined instruction encodings in an instruction, and replaces them with a single "undefined instruction" value

- (2014) https://patents.google.com/patent/US20160085550A1 *Immediate branch recode that handles aliasing* precalculates a portion of the target address of a branch (adds the branch offset to the lowest 14 bits of the PC)
- (2016) https://patents.google.com/patent/US10747539B1 *Scan-on-fill next fetch target prediction,* and a few other branch/fetch patents mention how cache lines already have the branch instructions in a cache line marked by branch type

Task Disaggregation

This refers to splitting a task that' s traditionally considered unitary into multiple pieces.
For example suppose you have a load in the load store queue that is behind a memory synchronization of some sort. In other words, the program semantics are that no memory operations are allowed to happen until the memory barrier is complete. That sounds like it is game over, no optimization possible.
But a load consists of multiple pieces! One piece is the actual "load value into register" part; but another piece is getting the actual value into the cache. What' s to stop you generating a prefetch that optimistically starts that loading into the cache now, maybe even before the synchronization begins to execute? (Of course you have to be careful about the precise semantics of each synchronization primitive, but the idea is possible.)
You can do similar things with stores. Again, of course, you cannot write a store to the cache too early. But you can preload the target line of the store into the cache, and simultaneously inform other CPUs of your Exclusive capture of the line, even before the store instruction becomes non-speculative.

Once you appreciate this point, you can do this kind of thing in multiple places. For example Apple disaggregates Instruction Retire into one part that handles freeing up the ROB (and can be extremely aggressive, freeing up to 56 ROB slots in a single cycle), and a second part that frees registers, a more difficult task that can only run at 16 registers per cycle.

Another place is: consider that instructions are sometimes split into two for the purposes of resource allocation (for example the implementation of an instruction may require allocating an implicit second destination register along with the obvious destination register). This sort of splitting an instruction into two, called cracking, is common; everyone does it.
What' s not common is that Apple then sometimes joins the two pieces back together. The resource allocation task was one step – best handled by two instructions – but Scheduling and Execution are different steps, best handed by one instruction! Again, disaggregation of the different parts of "performing the instruction".

Theory of the experiments

Now that we have an idea of some of the complexity in a modern CPU, how do we go about investigating what's there?
It's an imperfect science! One has to maintain in one's head a number of simultaneous possibilities, till the correct option becomes clear (and sometimes it never does). It requires patience and creativity!

One direction of attack is obvious.
You can learn the latency (how many cycles before you can use the result of an instruction) of an instruction by creating a chain of instructions that each feed their result to the next element in the chain, something like

```
op r2, r2, r0
op r2, r2, r0
op r2, r2, r0
op r2, r2, r0
```

...
Create 1000 successive instances of this assembly, loop it 10 times, use the CPU's cycle counter to tell you how long it took. and you have your result.

Alternatively you can learn the throughput (how much independent instructions of this type you can execute per cycle) via something like

```
op r3, r2, r0
op r4, r2, r0
op r5, r2, r0
op r6, r2, r0
```

...
Create 1000 successive instances of this assembly, loop it 10 times, use the CPU's cycle counter to tell you how long it took. and again you have your result.
These are, obviously, the essence of what Dougall used to create his latency/throughput website.

This is fine as it goes; important first step information. But note what it doesn't tell you (even in the big picture). It won't tell you about instructions that fuse together. Or clever tricks that may allow some instructions to exit the pipeline without requiring a step in each stage. Or how functionality is grouped together into different units. Or resource limits like the size of the ROB of scheduler queues. or other aspects of the OoO design.

The basic pattern of an OoO machine is that
- at the front of the machine, IN-ORDER, instructions are Decoded, Mapped (establish dependencies, establish the remapped registers) and Renamed (allocate additional resources, like a destination register).
- One of these stages (I assume Decode) also allocates an In-Order ROB slot.

- Between Rename and Retire the OoO machinery takes over
- At Retire, and after all speculation has been tested and found correct, resources are deallocated as each instruction retires.

If a resource cannot be allocated, then the machine will stall (ie no instructions will move past the in-order stage that cannot allocate). The OoO part of the machine will continue to execute, and eventually the back end will make enough progress to retire some instructions and free some resources, at which point the in order front end will resume execution.

The consequences of this pattern are that if you create a delay block (so that the head of ROB cannot clear until the last instruction of the delay block execute), then you can prevent the deallocation of resources until that head of ROB clears. Which means that you can stall the machine at the front-end in-order stage until the head of ROB clears. It takes a few cycles for instructions to then proceed from the stall down the pipeline to execution; and we hope to be able to see this glitch, this transition from one performance regime to another.

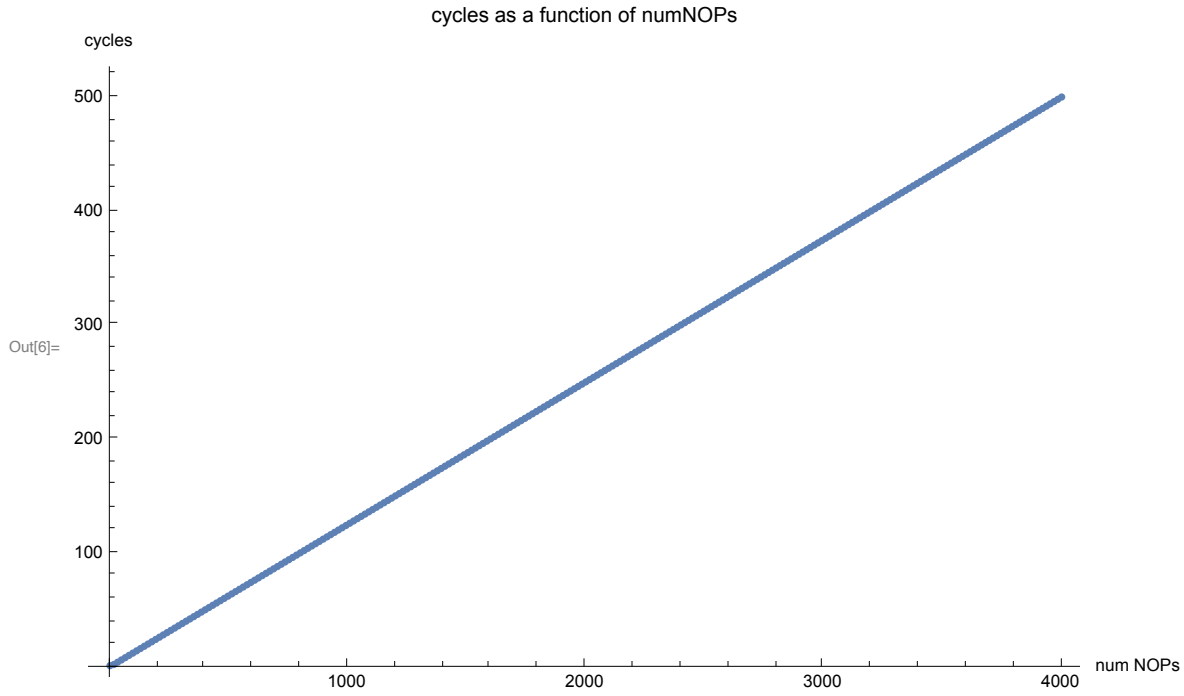
Life is complicated (as we will see going forward!) by the fact that, to make a machine more performant, you can, if not break the above rules, at least substantially bend them. So we will find that some resources are in fact allocated beyond the in-order front end, or that some resources that we imagine as unitary are in fact composed of multiple facets.

Let's see a simple example of the above idea:

ROB size (NOPs)

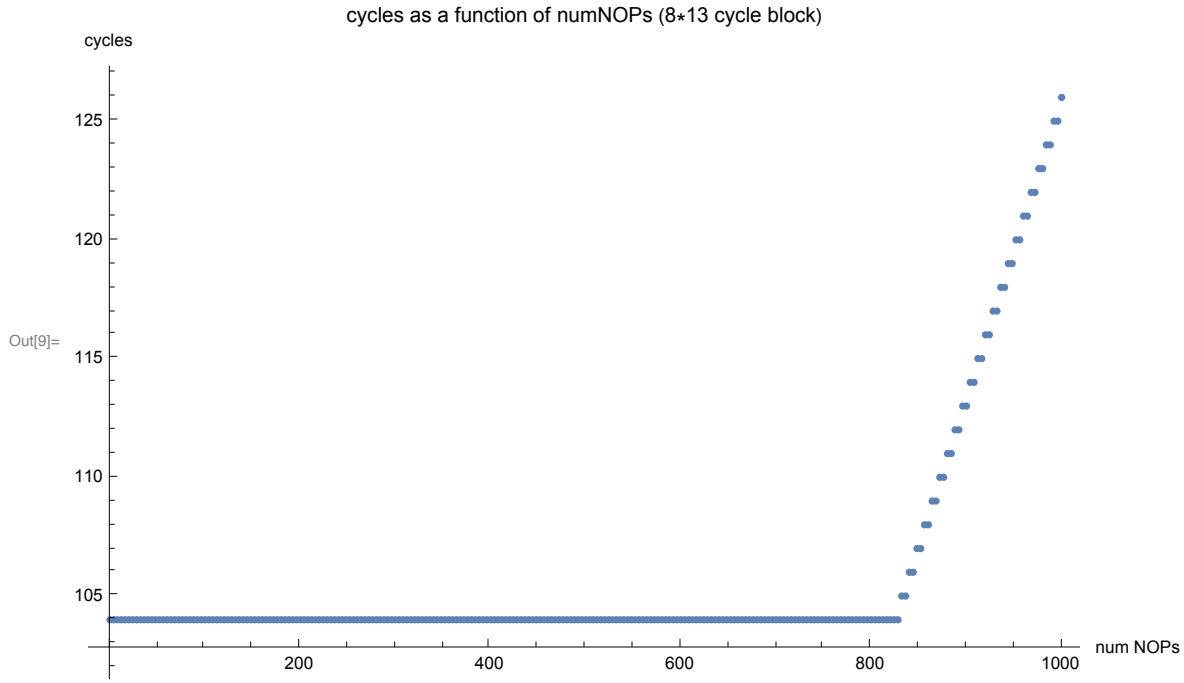
To get calibrated, our first code is nothing but a series of NOPs. We would expect that this should run at 8 NOPs/cycle, and that's exactly what we see.

Out[6]=



Now add a delay block of 8 chained FSQRT.D, each 13 cycles long.

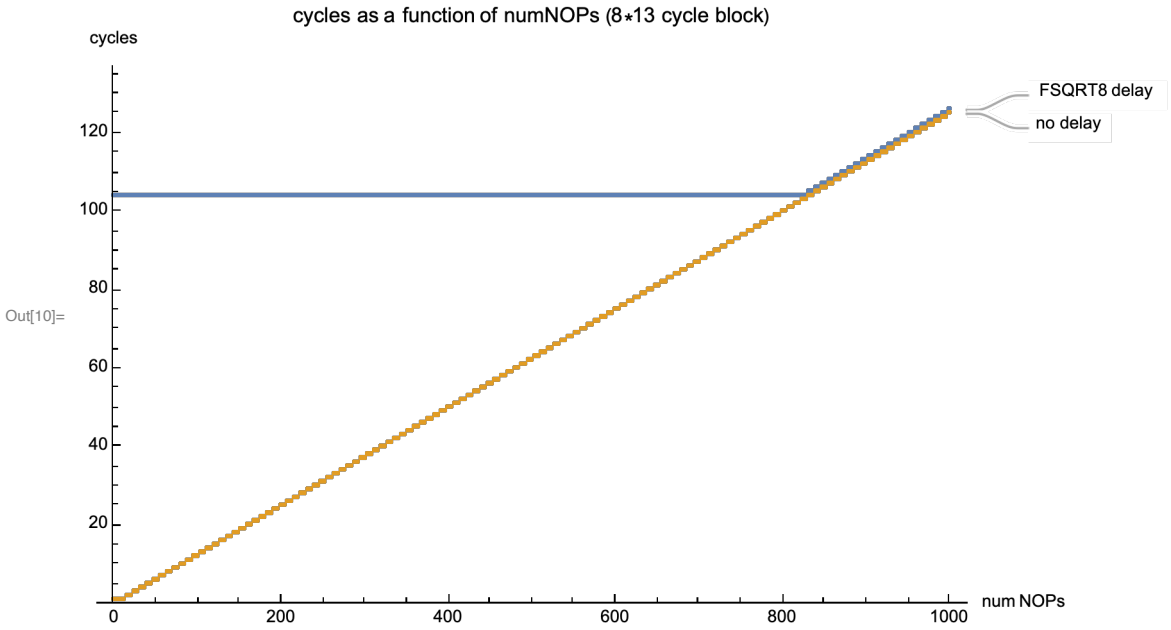
Out[9]=



Nothing too difficult to understand here; for fewer than (8*13*8=832 NOPs) the loop time is defined by the FSQRTs, for more than 832 NOPs the loop time is defined by the NOPs.

1. Delay block timing (blue) vs non-delayed timing (gold)

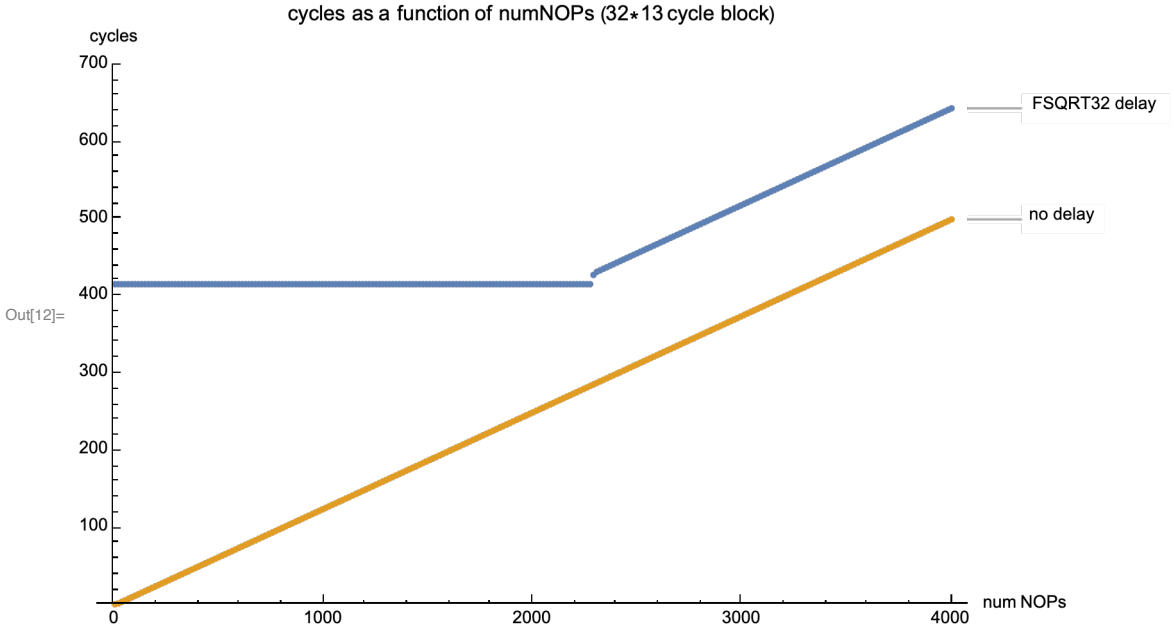
Out[10]=



We can force the FSQRTs to delay for a lot longer by chaining 32 of them together. Now we see something interesting, a glitch in the graph!

2. Delay block timing (blue) vs non-delayed timing (gold), with delay block and number of operations large enough to force a glitch.

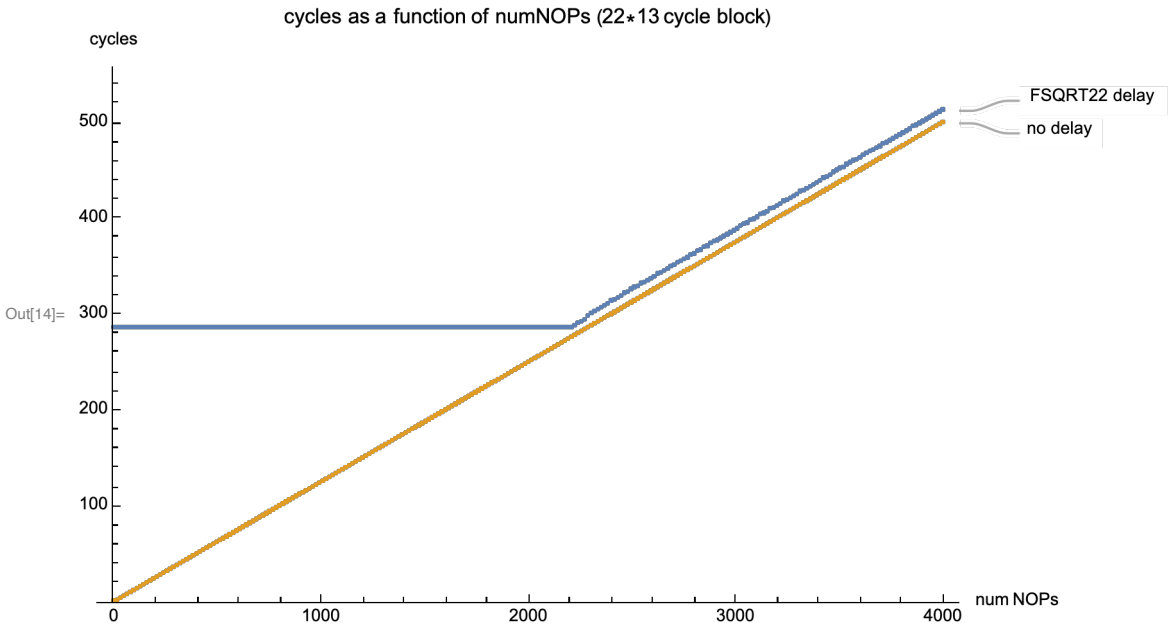
Out[12]=



What happens if we try to shrink the delay to the bare minimum?

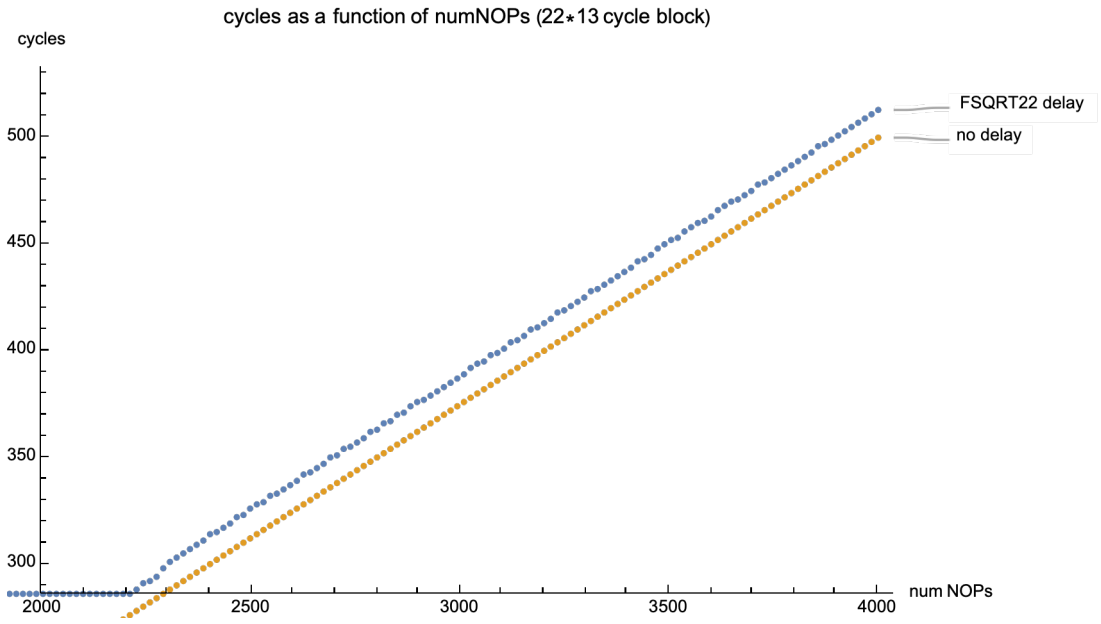
3. Delay block timing (blue) vs non-delayed timing (gold), with delay block and number of operations optimized for visible (but minimal) glitch.

Out[14]=

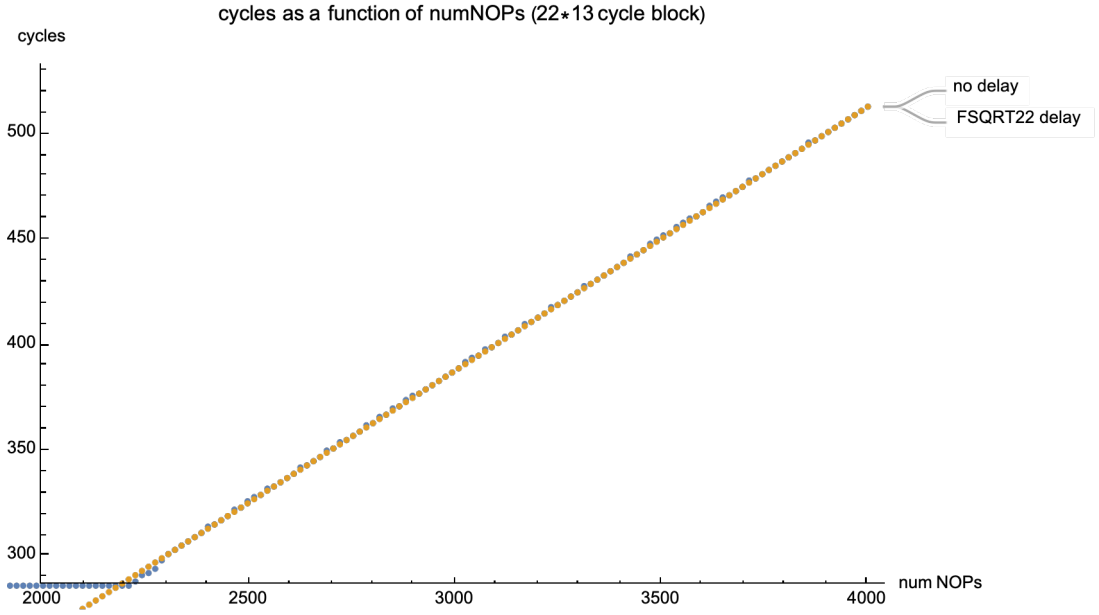


And zooming in

Out[15]=



Out[16]=



There are clearly two interesting numbers.
- At 4000 NOPs, the time taken is 500+13. ie one fsqrt was delayed each iteration.
- The delays starts at 2228 cycles.
How do we explain these facts?

In the case of NOP, the only resource that can possibly be relevant is the ROB. In other words, as a rough approximation, what's happening is:
- the fsqrt's block the ROB for long enough that around ~2200 NOPs can pile up in the ROB, unable to retire because the fsqrts are still going.
- at that point the next fsqrt (placed by looping around back to the start of the loop, is not able to even

enter the machine because one of the requirements to pass Decode to the next stages is a ROB slot.
- so that fsqrt is delayed, and our timing switches from MIN(22*13, N/8) to something like MIN(21*13, N/8+13).

That gives the intuition (and approximates the numbers) but is off by a few cycles because it does not include the overhead of the fsqrts themselves. Can we do a little better?
We will assume that any type of operation can be decoded at 8/cycle and placed into the ROB at 8/cycle.
This assumption may need to be revised, but let's assume that for now.
At t=-1 the ROB holds nothing.
At t=0 the ROB holds 8 successive fsqrts.
At t=2 the ROB hold 22 successive fsqrts and 2 NOPs.
At t=13 the ROB holds 21 successive fsqrts and 2+ (13-2)*8 NOPs
until we loop to the second iteration:
At t=T the ROB holds (22-T/13) fsqrts and 2+(T-2)*8 NOPs

For T=21*13=274, the number of enqueued NOPs will be 2178
For T=22*13=286, the number of enqueued NOPs will be 2274
We see trouble at 2228 between these two, so our ROB size is somewhere between these two.
Essentially at T=~280, we reach a situation where
- 21 fsqrts have been processed
- the last fsqrt is still executing, with a few cycles to go (22*13-280=6)
- The ROB holds 1 fsqrt, ~2270 NOPs, and perhaps two instructions from the branch test+loopback
BUT it can no longer accept the first fsqrt from the next loop iteration. So we get a stutter at this point, a delay in the smooth flow by 6 cycles before that fsqrt exits head of the ROB, and Decode is no longer stalled.

Once one has a more accurate idea of the machine, one can attempt to perform precise measurements. But this entire document will be about trying to understand the overall machine, thus we'll be satisfied with approximate numbers, leaving precision for later workers.

So we've established a few initial facts here:
- The ROB is ~2274 instructions in size. Over time we'll see that I believe it's best treated as ~324 rows, each 7 slots in size, giving a total of 324*7=2268 entries.
These "rows" are the units of Retire.

Retire (and the ROB) have to handle the following tasks
- deallocate resources (physical registers, load/store slots, ...) as instructions are completed BUT
- maintain machine integrity in the face of misspeculation (eg branch prediction).
This means that temporary state (in physical registers) can only be deallocated once it is certain there's no possibility that unwinding speculation may require the use of that state.

This leads to a delicate balancing act where one has to hold onto resources as long as necessary (for correctness) but wants to relinquish them (for reuse) at the absolute earliest moment that that is possible. Much of Apple's somewhat unusual ROB machinery is an attempt to be more aggressive in this reuse goal.

More Experimentation Theory

The ROB+NOP analysis gives some of the flavor of the enterprise going forward. The usual pattern is
- a delay block
- running in parallel with other operations
To void confusion, let's consider the delay block (which can be of variable length) to take duration D, and the Ops block to be comprised of N Ops that would be expected, under ideal circumstances, to take duration O.

(a) Once the "other operations" take longer than the delay block we get a diagram like diagram 1 above. The diagram to keep in mind is

```
| Delay<---------D---------->|| Delay<---------->|
|Ops<----------O--->|          |Ops<---------->|          |
```

transitioning to

```
| Delay<---------D---------->|     | Delay<---------->|
|Ops<----------O---------->||Ops<---------->|
```

(b) But once we make the ops block long enough, lack of some resource will stall the earlier stages of the machine (Decode, Map, Rename)and prevents all the operations from executing right away because all resources are locked up until the head of the ROB (the Delay block, retires .and so the Ops block encounters an additional delay, a delay caused by its inability to fully execute). This gives us a diagram like 2. with a notable glitch.

```
| Delay<---------->|          | Delay<---------->|
|Ops<----------xxxxxxxx---------->||Ops<----------xxxxxxxx---------->|
```

In other words once the resource count used by Ops is exceeded, then the delay O_measured(N ops) is larger than O_expected(N ops) by the delay xxxxxxxx that occurred while the Ops were sitting idle, unable to proceed through the machine until the head of the ROB retired (at the end of the delay block) and resources were freed.

Note that for this condition to occur (presence of the xxxx) we require
- D must be large enough to block N ops partway through their execution
- N must be large enough that all resources are fully consumed.
However once we see a glitch, it's sometimes desirable to try to shrink D and N down to about the minimum values that will reproduce the phenomenon, the glitch in timing, as in diagram 3.
Other times it makes sense to push D rather larger than the minimum because you'll then catch an

additional unexpected event!

In this case the number of interest was the minimal number of ops that had to be enqueued to generate a glitch. As our analyses become more sophisticated, we will be considering other features of these sort of graphs. For example the slope of the NOPs line tells us how many NOPs/cycle can be processed by the machine. We may structure things so that, after a glitch, the slope of the post-glitch line tells us something about how rapidly resources are freed (as opposed to what we have learned so far, namely how many resources can be used up).

Register Files

More OoO Theory

Remember why we distinguish between logical and physical registers: https://en.wikipedia.org/wiki/Register_renaming.
We use physical registers (the logical physical register perhaps mapped to different physical registers as execution proceeds) for two reasons
- to avoid having to pause execution because of false dependencies
- to maintain state in a temporary form for speculative execution. If we want to be able to execute further down a speculative path, we need to be able to hold more temporary state, ie more physical registers

Most machines distinguish three classes of registers: int, FP/SIMD, and flags; and all three are renamed and live in separate physical register files.

So consider a stream of instruction that each generate an integer result. Every such result requires a destination register which will be a newly allocated physical register. If the pool of such physical registers runs out, integer instructions cannot proceed until more physical registers become available. In fact *no* instructions can proceed, because register allocation occurs in the front end of the machine, where instruction flow is still in-order.

The flow of instructions is essentially
Fetch
Decode,
Map (mainly figure out the dependencies between the current group of eight newly decoded instructions)
Rename (allocate resources required by each instruction, eg a destination register
Dispatch

Schedule
Execute

Retire
Complete
Writeback

The initial set of stages all occur in-order – meaning that any block in one of those stages blocks all subsequent instructions.
Schedule and Execute occur out of order (which is where the performance win is!) The win has two

sources:

- some instructions will be delayed by random events the compiler can't predict, like loads that miss in cache. Other, independent instructions can occur while those instruction wait for whatever is they're waiting for.

- instructions tend to cluster as sequentially dependent instructions (C depends on B depends on A), but these dependency chains are usually surprisingly short (about 10 to 20 instructions) before they are terminated (usually by storing the result to memory), after which an independent chain starts. Thus if we can queue enough of these independent dependency chains in the machine, we can run many of them in parallel.

In *theory* the compiler could do this for us on an in-order machine. In practice this fails because
(a) most languages don't provide enough information about the non-overlapping of storage for compilers to be able to disambiguate that the store from the last dependency chain doesn't overlap with the loads that starts the next dependency chain
(b) most of these dependency chains are separated by branches of some sort, and the compiler can't be sure which way the branches will go
(c) trying to optimally schedule hundreds of instructions in a compiler is a combinatorial explosion problem, one that takes a lot of time, usually too much to be practical.

Retire, Complete, and Writeback tend to be bundled together in modern cores, and (mostly) happen in order.
Retire is easy to understand. It's the final point at which instructions have done (almost) everything they are supposed to do, and are no longer speculative. Their resources can be released, and reused by new instructions.
Writeback used to mean that results that were held in physical registers were written back to "architected registers". This is a term you will see if you read really old papers (say pre-2000 or so.) The evolution of out of order technology went essentially something like

(0) an in-order machine, with let's say 32 architected integer registers x0..x31 has a register file of these 32 registers. Every instruction affects its destination architected register.

(1) very early out-of-order has each ROB entry providing a slot that acts as the destination register of each instruction. So Register allocation happened automatically as allocation of the ROB slot. Writeback meant copying from that ROB slot register to the architected register file.
This is clearly built on the in-order model, with the idea that when anything fails, you fall back to the architected register file as the true state. But it means the number of physical registers equals the number of ROB entries -- and what to do about FP/SIMD vs int registers? And you waste a register slot for instructions (like branches) that don’t use a register. And all that writeback copying uses power.

(2) the physical register file was detached from the ROB. This solves the FP/SIMD vs int problem, and allows each of int register file, fp register file, and ROB to be independently optimally sized. But you're still paying writeback costs

(3) various alternatives were adopted for how to recover from misprediction using maps that describe how physical registers map to logical registers. Done correctly, this avoids the need to have a separate architected register file along with writeback; the state of the machine exists only in the physical register file together with the map between the physical and architected registers.
I refer frequently to the paper (2004) http://pages.cs.wisc.edu/~rajwar/papers/taco04.pdf *An Analysis of a Resource Efficient Checkpoint Architecture* by Haithim Akkary. This was written just at the point of this transition, so it describes multiple options for how this can recovery be performed.

(4) the usual state of the art for industry today is much (not all) of what Akkary is proposing in that paper.
Apple is the only company I know of that's using the next step in the evolution which is to use a History File to record the changes made to the mapping tables as execution proceeds. This change decouples the ROB itself (a queue of instructions) from the History File (a record of changes to register mappings). This is as opposed to recording these mapping changes as part of the ROB.
The first advantage of this is that once more the two can be separately sized, rather than tying the number of instructions in the ROB to the number of changes made to registers.
The second is that the two are deallocated separately at Retire – the ROB can free up to 56 instruction in a single cycle; the History File, operating independently and with a more difficult task, can free up to 16 registers in a single cycle.

(5) The next step beyond 4 would be the use of Virtual Registers (to be discussed below). Apple doesn't do this yet, but probably will soon.

Finally the Complete stage is mostly meaningless (by the time an instruction is considered eligible to Retire it has executed and so completed its job)… except for the case of stores.
In the old days, at the the point of Retiring a store, the store would be copied from the Store Queue to the L1D cache. (This could involved pulling in a line that was the target for the store.) And it was only when that line had been returned from L2 or DRAM, and had the store written to it, that the store could be considered Complete.)
Nowadays, for a store:
- Execution consists of calculating the store address, and placing the store address and store data in a store queue.
- At a later point (when the store is non-speculative) the store will move from the store queue to a write buffer (sitting between the store queue and the L1D cache) at which point the entry in the store queue can be released.
- But the store is still, (in some sense) not complete! At some random later point the store will move

from this write buffer into the L1D. And at some random even later point, it will be made visible to other cores.
It's something of a question of exactly what you're trying to achieve as to when you define a store as Completing, so once again this has become a somewhat obsolete stage.

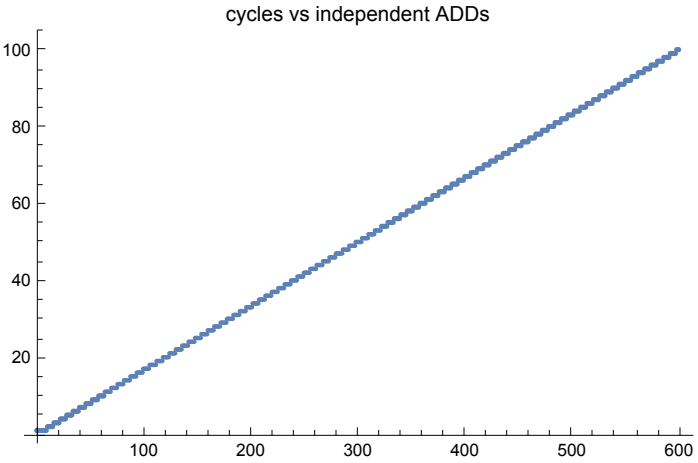
Essentially Retire now means Deallocate (resources) in the same way that Rename means Allocate (resources).

Back to our stream of integer instructions, and to experiments.
Each of these allocates a physical register in Rename. That register can be freed once the instruction Retires (but not earlier). Which means that if the integer instructions are blocked by a delay block at the head of the ROB, eventually all physical registers will be allocated, the machine will stall, our timing will see a glitch. Let’s try it!
Remember as we perform these experiments below, all the details I gave above (eg Apple’s use of a history file decoupled from the size of both the ROB and the physical register files) was not told to us by Apple! It’s all the result of experiment, and that’s what this long run of experiments below is designed to figure out.

Integer Physical Register File Size (ADDs)

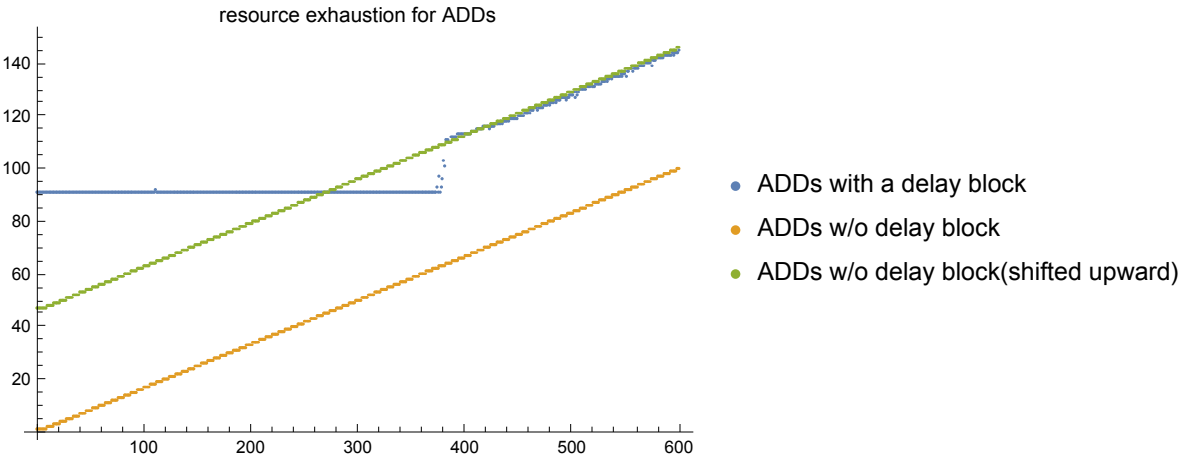
Before going further let's validate our foundational ideas about the physical register file.
We start with the simplest model, then see if it breaks as we add tweaks.
Start with a run of instructions (`ADD x0, x5, x5`) that each require a physical register to hold the generated result.

Out[18]=



No surprises, the primary notable fact is the slope (6 independent integer adds/cycle).
Now let's add in a delay.

Out[20]=



In[•]:=
```
{377,91},{378,93},{379,96},{380,103},{381,101},{382,111}
```

Here are the notable points:
- Clearly at ~378 ADDs, there is a notable jump in the cycle time.
- The slope of the delayed ADDs is unchanged, just shifted up by 46 cycles.
- So at ~378(=63*6) physical registers, at 63+2 cycles in, ADD progress stops, ie at INT_STOP_-TIME=(num phys reg/6 + 2 [two cycles for branch and enqueing the sqrts])

For how long? Until the ROB clears, so
WAIT-TIME=DELAY TIME (13*numSqrt) - INT_STOP_TIME.

I *think* the transition occurs over ~5 cycles because of 8-wide Rename.

```
| Delay<---------------------------------->|                | Delay<---------------------------------->|
|Ops<---------------------------xxxxxxxx-------------->||Ops<---------------------------xxxxxxxx-------------->|
```

Let's assume there's a few cycles of jitter in xxxx (just assume that for now). And that the actual physical register count is say 380. This would mean that for N very near the resource limit (say 379 or 380) those N's will occasionally hit the resource limit (and execute the slow path, where xxx force a delay and everyone has to wait for ROB to clear) and will occasionally not hit (meaning the fast path, no XXX delay, no requirement for the next block of fsqrts to delay till the ROB clears before they can begin execution).
If such a jitter existed then we'd see something like maybe 1/6 of the time the 378 case is slow, 5/6 of the time it is fast, and the average cycle count is 5/6 to 1/6 weighted. Likewise the 379 case is 2:6 to 4:6 weighted and so on, and we'd get the sort of staircase we see. Depending on exactly how we aligned all the instructions, and the source of jitter, we might be able to get a perfect jitter free-case, to a perfect linear ramp.
I assume the jitter reflects differential widths in different parts of the machine (eg Decode can generate 8 instructions per cycle, but integer execution can use up only 6 per cycle) meaning that slightly different numbers of instructions could be enqueued (and blocking the transit of fsqrt from the in-order path to the OoO path) in different places on different loop iterations. But trying to pin that down further using this particular harness seems inefficient.
Going forward we should GENERALLY expect a slightly noisy transition region (over 6? 8?) cycles because of our jitter explanation, and not waste time obsessing over its existence.

So what is definitely established is that

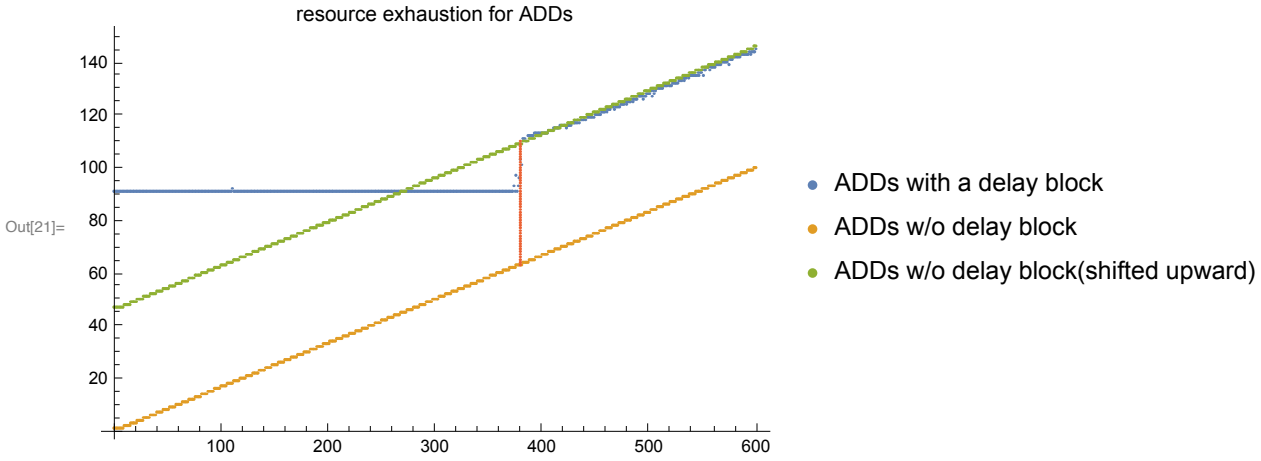
• there's some sort of limitation at around 380 ADDs (seems like physical registers)

• assuming the above point, we've also learned that registers are deallocated at Retire (deallocation is delayed by the Head of ROB blocking) rather than some more ambitious schemes that de-allocate a register when all *users* of that register have completed. We'll discuss this later.

It's important to have a correct understanding of the analysis, which is somewhat more abstract (less mechanistic) than Henry Wong's method. In particular it's very tempting to view the x-axis (the N axis, where N is the number of filler instructions between delay block) as representing time, and doing that will drive you crazy!

To be sure you understand, consider the graph below.

Out[21]=



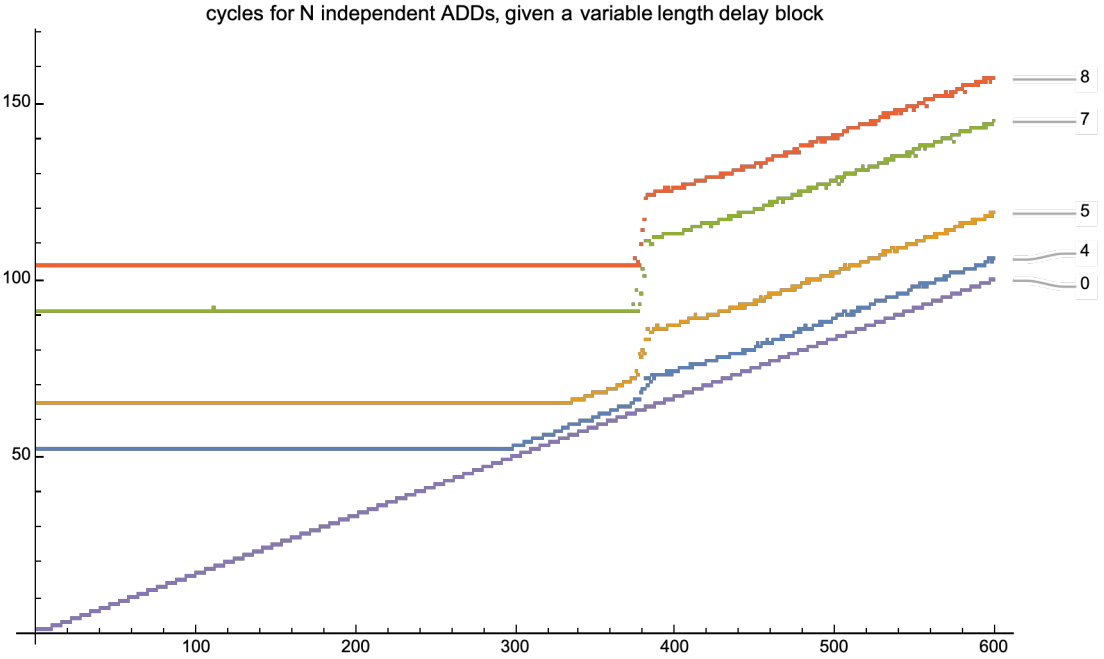
Essentially we have two independent blocks (call them chains) of sequential execution.
The first chain is the delay block, the block of 5 (or 10, or 20) chained `FSQRT`s. The amount of time this block would take to execute is independent of the number of filler instructions (ADDs), and can be viewed as the flat part of the blue curve, extended to arbitrarily high values of N.
The second chain is the `ADD`s. For small N, they take an amount of time linear in N, following the left hand side of the gold curve. For large enough N the execution of the ADDs is forced to block (the xxx time) while we wait for a resource to be freed.
Thus for N<~380 the time taken by the `ADD`s follows the gold line,
for N>~380 it follows the green line,
with the two lines linked by a jump occurring at N=~380.

SO: Loops with filler of less than ~380 execute on the fast path; loops with filler of more than ~380 execute on the slow path.
The curves show what happens for loops with differently sized filler, they *do not* show what happens in time as successive `ADD`s of the filler are executed!

The whole blue curve (the one we measure) is the time taken to execute (in parallel) the filler block (following the curve gold/red/green) and the delay block (always taking a constant 7*13=84 cycles). The time for a loop iteration is always the larger of these two possibilities.

To get a clean curve, it's ideal (though sometimes not possible) to have the delay block last substantially longer than the filler block takes to reach resource exhaustion.
Observe what happens when we don't do that.

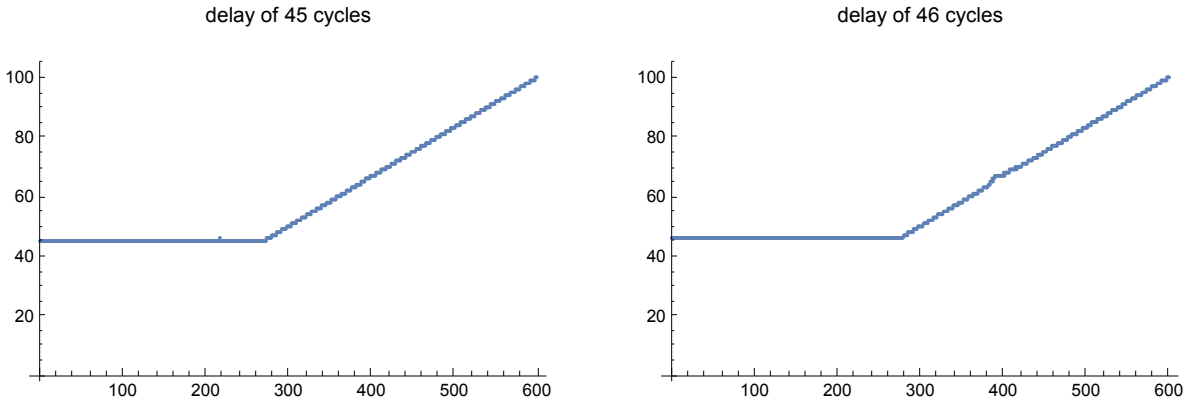
Out[29]=



We point out these salient facts. Note that 5*13 takes 65 cycles, which is time to perform 390 adds, so a delay of 5 is right on the cusp, able to show the phenomenon but not a clean jump.
• For a delay that is too short (4 `FSQRT`) we clearly don't cleanly hit the phenomenon of interest (ie `ADD`s delayed because of inability to allocate a resource, blocked behind the head of ROB).
• For a delay that is just on the money, (5 `FSQRT`) the crossover region (the jump at ~380 `ADD`s) is not smoothly isolated and it's easy to get distracted in the hopeless task of trying to figure out the exact structure of the crossover region.

an aside for experts

Out[32]=



It's interesting to consider why we still get a (small) bump for the case of delay of 4*13 (52) cycles.
The first parts of the curve, up to N=~380 are clear. What's unclear is why there should be some delay for, say, a filler of 390 ADDs.
After about 312 ADDs are executed, the head of ROB clears, the physical registers used by the first few ADDs behind the head of ROB will start to be released (in fact at a rate of 16/cycle as we will see) and there is no obvious reason for the ADD chain ever to be blocked or delayed, no reason for any xxx time.

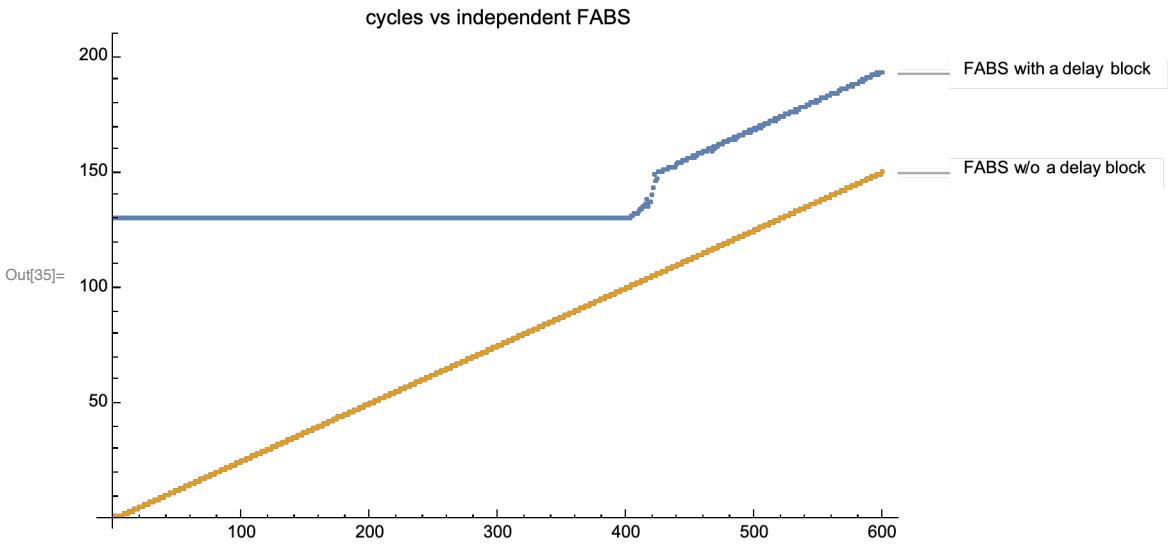
This understanding is (kinda sorta) confirmed by comparing a delay
of 45 cycles (4 `FSQRT s1, s1, s1 FADD s1, s1 FABS s1, s1`)
with 46 cycles (`4 FSQRT s1, s1, s1 FADD s1, s1 FADD s1, s1`).
45 cycles behaves as expected, 46 cycles shows a slight but definite bump at N=~380.

I believe this effect is a consequence of a low-level implementation detail in Apple's register file. I'll explain this below, in power aspects of the register file, once we have explained higher level aspects of the register file.

FP Physical Register File Size (FABS)

Let's try the same strategy for fp registers. The code is as before except we replace the `ADD` with `FABS d2, d0`

Out[35]=

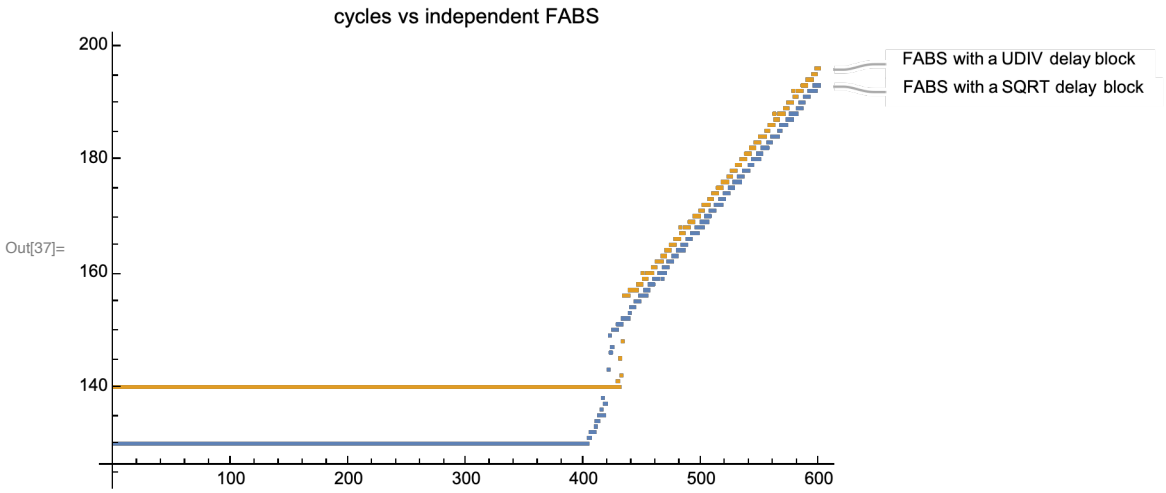


In[•]:=
```
{403,130},{404,131},{405,131},{406,132},{407,132},{408,132},{409,132},{410,133},{411,134},
{414,135},{415,136},{416,138},{417,135},{418,137},{419,137},{420,140},{421,143},{422,149},
```

So we see the structure we expect by now. The slope of both lines is 4 independent fp instructions/cycle, as expected.
The jump happens over the range 403..425, so over a range of about 25 instructions, centered at about 415 instructions.
The jitter range is a little messier than before, I assume at least in part because the delay block and the `FABS` are sharing registers and execution paths.

We can use an integer delay block (chained `UDIV`, 7 cycles/iteration) to get a slightly cleaner result

Out[37]=



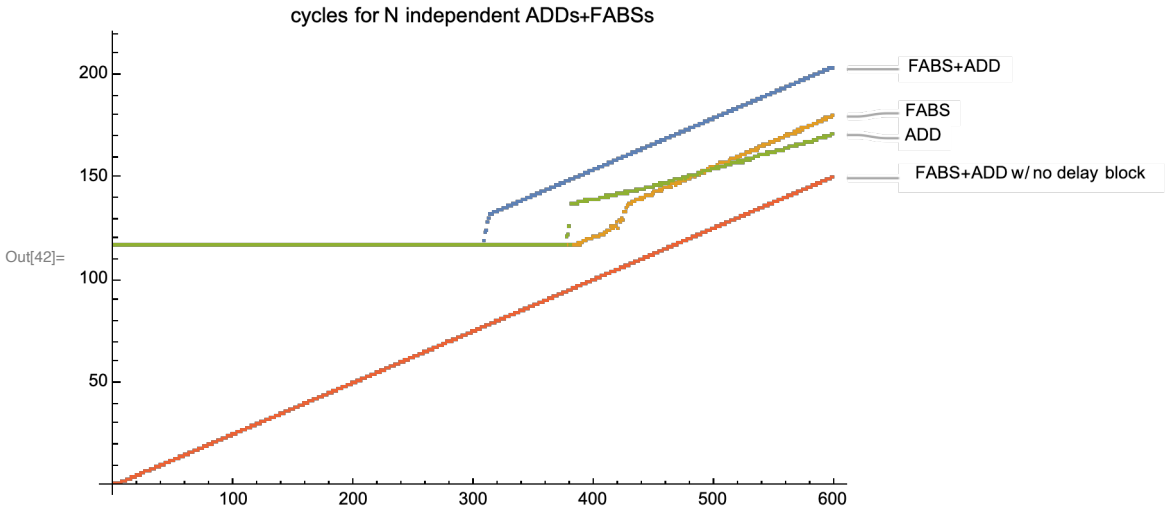
In[•]:=
```
{428,140},{429,141},{430,140},{431,145},{432,142},{433,148},{434,156},{435,156}
```

We see a narrower transition region, and without the delay block using up some of the fp registers, we see that the fp register file is perhaps closer to 432 or so in size.

Register File Size (All Registers)

The apparent number of physical registers for int and fp are fairly similar.
One can entertain a number of possibilities (is there a common register pool?), so lets examine what happens when we try to allocate both integer and fp registers. Let's try a probe consisting of an `ADD` and a `FABS`.

Out[42]=




```
{308,117},{309,119},{310,123},{311,124},{312,128},{313,130},{314,132},{315,132}
```

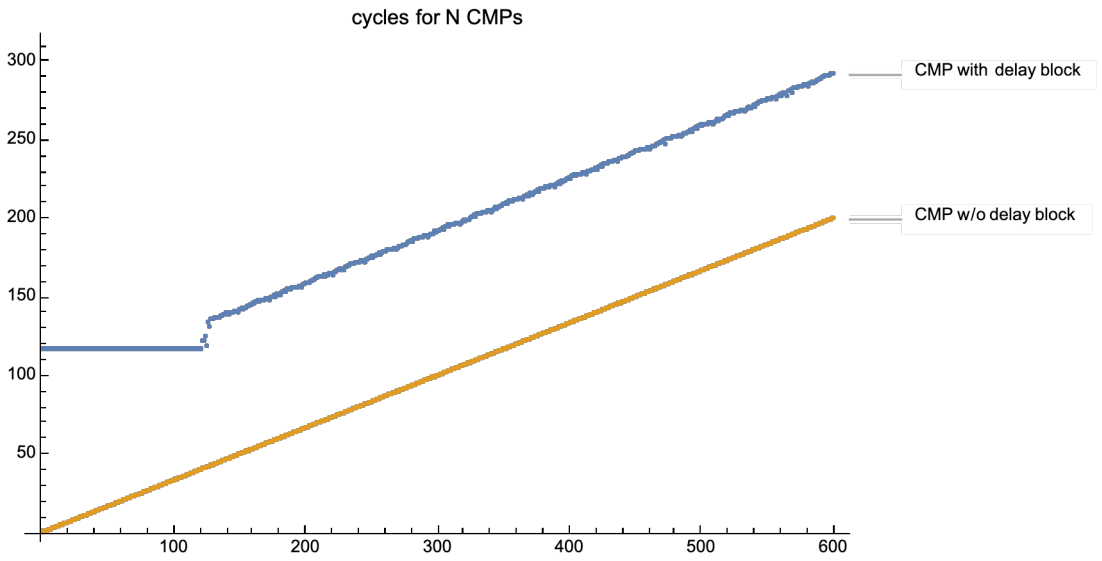
Now this is interesting!
The red curve omits the delay block, and shows a slope of 4 ops/cycle. We are throttled by the 4 fp pipes.
The green curve and the gold curve are, respectively, the pure integer test and the pure fp test.

The blue curve is the case of interest, with a probe of (`ADD+FABS`).

We see clearly a standard jump, centered at about 312 or so. How can we interpret this?
Clearly the idea that registers are shared between int and fp fails. If that model held, we'd jump at something like 192 (half the ~384 registers allocated to fp, half to int, stall).
But something clearly is shared! There's an additional resource that constrains us, a pool of ~624 somethings that's shared by int and fp registers.

We can continue our investigation by testing the third available pool of registers, the physical flags register.
Let's start by testing `CMP x0, x0`

Out[45]=

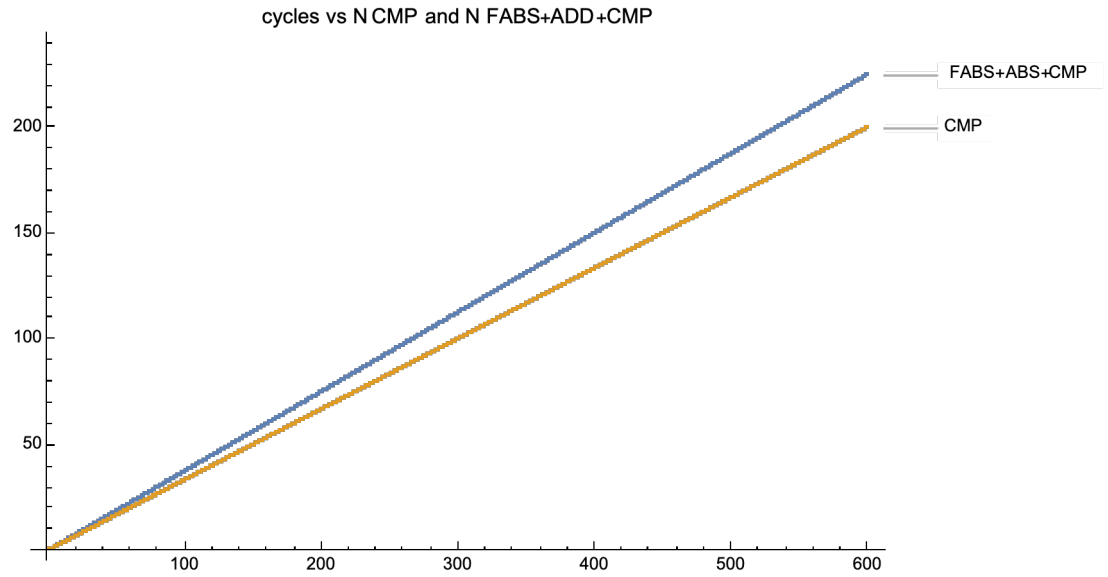


In[•]:=
```
{121,117},{122,122},{123,122},{124,125},{125,119},{126,134},{127,131},{128,136}
```

So no real surprises. This time the slope of the lines is 3 instructions per cycle;.
However the jump is at ~124 (128?) physical flags registers. So not the same number as int or fp...

Now for the exciting part, run all three together! First let's investigate the simple case, with no delay block.

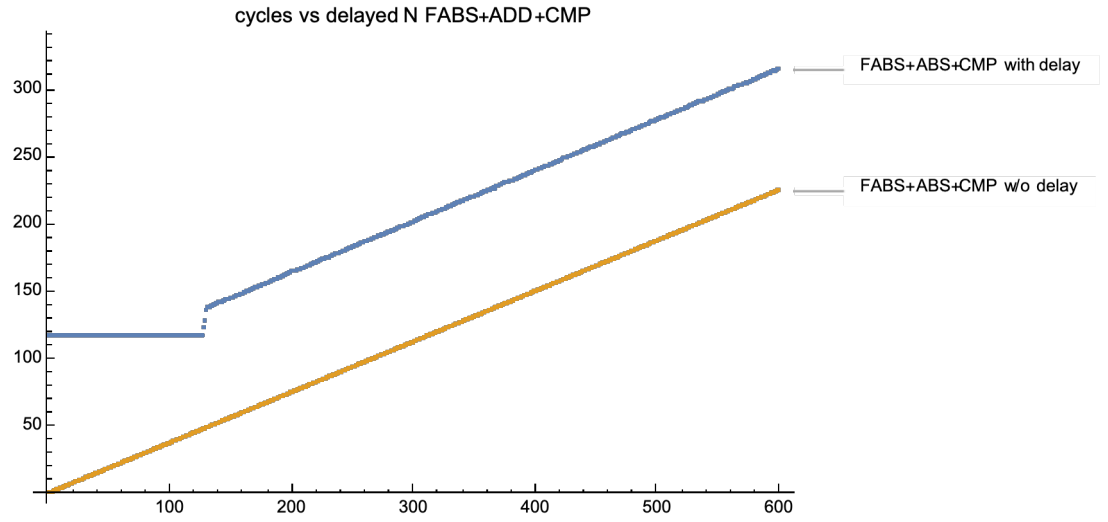
Out[47]=



Now our rate drops to 600 triplets in 225 cycles, 8/3=2.667 instructions/cycle.
The limiting point is the 8-wide front end (Decode, Map) of the machine. We have a total of 3*600 instruction, processed 8 /cycle, which takes 1800/8=225 cycles.

Now add the delay block.

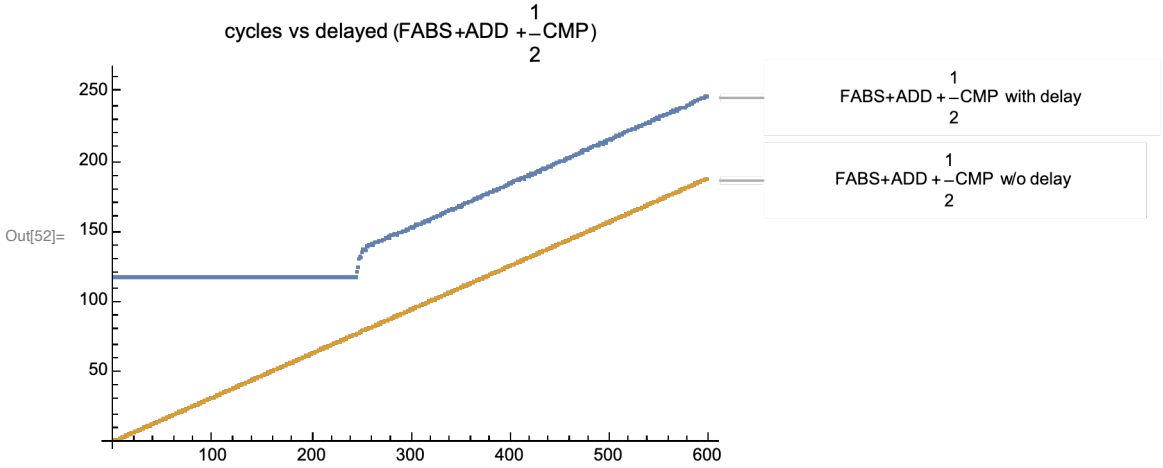
Out[49]=



In[•]:=
```
{127,117},{128,123},{129,128},{130,136},{131,138},{132,138},{133,138},{134,139},{135,139},
```

We see that we're constrained by the number of compares; we can't get beyond that to test the region of interest. If there is a common pool of 624 somethings, cut 1/3 each way gives 208. Not reachable via a unit of (CMP+ADD+FABS).
What about 624/5=124.5? Right on the cusp! So let's try one CMP paired with two ADD and two FABS. We'll implement this as only allocating the CMP for every second (ADD/FABS) pair, so the unit is (ADD-+FABS+.5 CMP).

Out[52]=



In[ ]:=
```
{245,117},{246,121},{247,124},{248,130},{249,132},{250,131},{251,135},{252,137},{253,137},
```

Almost! But not quite?

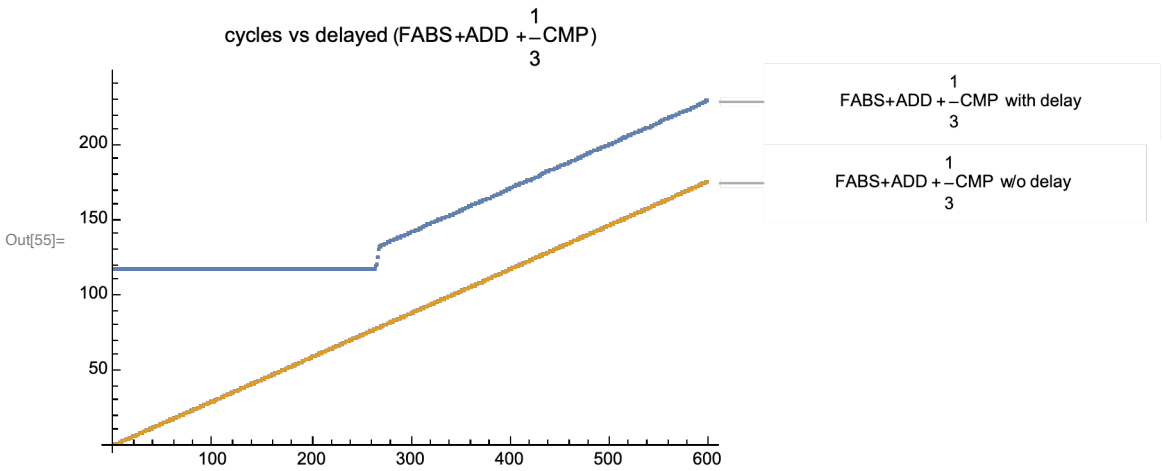
The jump is at ~250. Is this what we expect?

250 units corresponds to 250*(ADD+FABS+ half a CMP), so we're possibly still being limited by the ~128 flags physical registers before we can get to the good stuff, since half of 250 would be 125.

But 250*2.5=625. It's plausible that we're seeing a signal. But let's confirm.

Drop to a third of CMP.

Out[55]=



In[ ]:=
```
{264,117},{265,119},{266,120},{267,125},{268,130}
```

OK, finally, the signal we're looking for.

The fundamental unit of allocation is $\left(\text{ADD} + \text{FABS} + \frac{1}{3}\text{CMP}\right)$ (implemented as adding a CMP for every third ADD+FABS).

We execute 266 of this unit) so 266*(2+1/3) =620 register allocations! And this time all three register pools (int, fp, even flags) are only partially exhausted, with 266 int registers, 266 fp registers, and 87 flags registers being used up at the point of the jump.

So there is some *communal* structure related to the allocation of registers, of size ~620 entries, which is used up *in addition to* the register files.

The History File (ROB structure tracking all changes to Register Mapping tables)

What are these unknown shared ~620 resources?
I made many different hypotheses, and experimented with many different things, but the real answer appears to be in this patent: (2019) https://patents.google.com/patent/US20210064376A1/ *Last physical register reference scheme*.

One way in which Apple optimizes the Retire tension I discussed is through a structure called a History File.
Conceptually the history file sits next to the ROB, and while the ROB holds a sequence of instructions (all instructions, in-order), the smaller HF holds the sequence of changes that were made to the three (int, fp, and flags) register mapping tables.
The HF has ~620 entries, and requires an entry for every instruction that will modify the mapping tables (so basically any instruction with a destination register, including a flags destination). The HF also includes a single bit flag that this is the last mapping referencing the physical register referenced by the mapping.

To understand this aspect of the ROB, perhaps start by reading (2001) https://courses.cs.washington.edu/courses/cse378/10au/lectures/Pentium4Arch.pdf *The Microarchitecture of the Pentium® 4 Processor* which describes the P6 and then the P4 schemes.

The P6 scheme was very simple (as these things go). The ROB and the PRF were the same structure; allocation of a ROB entry was the same thing as allocation of a destination register. Speculation was handled by having a separate PRF, named the RRF (Retirement Register File), that represented the "true state of the machine as of Retire" so that you could recover just by pointing all the Map tables to the RRF.
Problems with this scheme include
- that your ROB and PRF are linked in size, though really you want the ROB to be some scaling factor larger than the PRF; and
- the copying of a GPR to the RRF on every retire costs additional energy.

The P4 scheme has separate (and separately sized) ROB and PRF, and a separate Retirement RAT (Retirement Mapping Table) that's updated at retire.
This is progress! We have
- separate sizing of ROB and PRF; and
- the energy cost of updating an entry in the Retirement RAT is less than that for copying a register to the RRF.

It’s hard to see what the problems with this scheme are, until you start being more ambitious.

Suppose we have a situation where there are many instructions in the ROB, and halfway down the ROB there's a branch that we know is mispredicted because we just executed the branch and learned that. So we need to engage in branch mispredict recovery.

The P4 recovery scheme is to mark in the ROB entry for the branch that it was mispredicted, and continue as usual *until the branch retires*. At that point
- the instructions before the branch have all retired (they were ahead of the branch so were valid);
- as they retired they updated the RRAT (so RRAT represents an accurate register mapping table at the point of the failed branch);
- and so handling the misprediction mainly means copying the RRAT into the frontend Mapping table.

But this also means that all that time from when we learned the branch was midpredicted until it retires is dead time! The machine kept chugging away at instructions after the incorrect branch instead of doing something useful.

What we would prefer, as near as feasible, is, rather than waiting till it retires, *as soon as* we know a branch is mispredicted
- we keep all the instructions *older* than the branch (and still executing) alive
- we kill all *newer* instructions
- we flush all *inappropriate enqueued* instructions and begin fetch from the new address
- the new instructions begin execution even while we're still retiring the old instructions in the ROB that were ahead of the the branch.

For this sort of scheme to work, obviously we need machinery to mark and flush instruction as appropriate; but most relevant right now
- we need to have the correct mapping table in place as soon as the newly-fetched instructions enter the machine; so
- we can't wait until the branch Retires and we can swap in the RRAT
- we need to construct the correct table, and install it at the correct point of execution (so that it’s seen by the new stream of instructions as they hit Map)

Details of how you might do this are given in http://www.cs.wisc.edu/~rajwar/papers/taco04.pdf, which we’ve already referenced once. This paper discusses many things, but in section 3.2 it discusses two slight modifications to the P4 scheme to reconstruct the mapping table as fast as possible without waiting until the branch Retires. Both alternatives require each instruction as stored in the ROB to carry additional information that looks something like "this instruction swapped pRegA for pRegB as the mapping for lRegC"; and by running through these mapping statements in sequence you can reconstruct the mapping table from a given starting point.

It's unclear what Intel does nowadays but they were aware of this limitation in P4, and addressed it in Nehalem.

(2011) https://www.intel.com/content/dam/www/public/us/en/documents/research/2010-vol14-iss-3-

intel-technology-journal.pdf *Intel Tech Journal Nehalem Issue*
page 16 discusses the problem, but says nothing beyond "we have a fix".

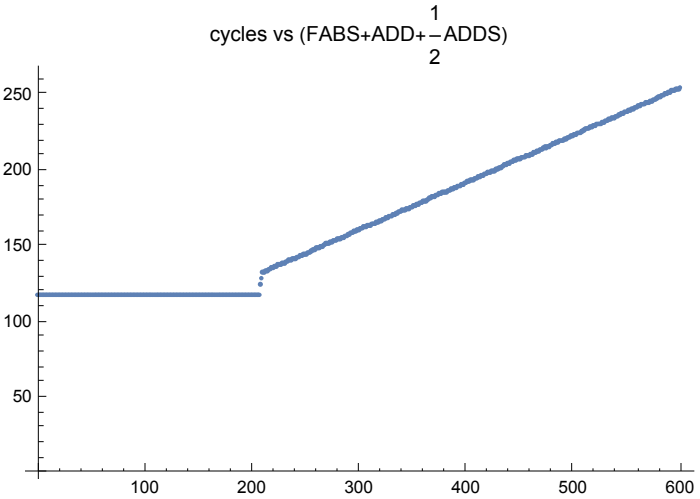
Akkary's more performant solution, the one called HBMAP+WALK is essentially what Apple uses. However rather than store the mapping updates associated with each instruction (updates that are not required by many instructions) Apple separates those changes from the ROB into a separately sized *History File*, with a pointer in each ROB entry pointing to a History File entry (or something like that).
The essence of the History File is that it doesn't track values or instructions, all it cares about is changes that were made to the register Mapping files, so that by rewinding the History File, you can rewind machine state to the last known good point that you wish to reach.
A secondary task of the History File is to note when physical registers can be freed for reuse. We'll see how that happens below in the Duplicating Registers section.

How Do “set flags” Instructions, Like ADDS, Modify the History File?

Buoyed by this understanding, let’s test `FABS+ADDS`. (Recall that an S suffix on an integer instruction means that instruction also sets the flags register in addition to the ADD or whatever.)
The question of interest is: how the flag register rename is handled by the history buffer?
The obvious expectation is that ADDS, generating both an add result and a flag result, will require two slots in the History File., since there are two remappings being performed, in two different (int and flags) mapping tables.

However we have to be careful in this test. We can’t exceed 128 ADDSs before we hit the limit of the number of physical flag registers! We’ve already seen this above when we were probing the History File. Let’s try (`FABS+ ADD+ .5 ADDS`). This consists of 2.5 instructions, but might require either 2.5 HF slots (one for each instruction) or 3HF slots (2 slots for the `ADDS`, one for `ADD` destination, one for the flag destination). So we would expect this to max out at N units of either 206(=620/3) or 248(=620/2.5).

Out[57]=



In[•]:=
```
{207,117},{208,124},{209,128},{210,132},{211,132},{212,132}
```

Hah! Isn't it nice when our understanding advances to the point where was can start to make testable predictions?!
So that's pretty clear! Instructions that both perform some other calculation and set the flags register (ie have two destination registers) require two HF slots.
More precisely, given the way I described the History File (to unwind register mappings) one slot will be allocated for each register rename. Meaning that we should expect
• zero HF slots for stores
• zero HF slots for explicit prefetch, `PRFM`, instructions
• zero HF slots for writes to special registers (somewhat... some special registers are renamed)
• one HF slot for standard loads
• two HF slots slots for load pair (which takes two destination registers).
These are all confirmed. (Note our goal right now is to understand the HF, not to probe details of loads, stores or PRFM instructions!)

BTW the special registers case is surprisingly interesting. Reading (and especially writing) special registers has usually been a very slow path, because the easiest way to handle a special register write is to delay performing the operation (and all subsequent instructions) until the relevant instruction is at the head of the ROB, ie to "serialize" the instruction stream. The reason for this is that there are multiple different special registers that all do something different (change memory mappings, change how interrupts are handled, change floating point behavior, ...) and you can't make those changes while the instruction is speculative because they would be so difficult to unwind if the speculation proved in error.

But that doesn't stop Apple. Check this out: (2019) https://patents.google.com/patent/US10838723B1/ Speculative writes to special-purpose register.
Different registers are handled differently but the basic idea is that
- Some easy cases (most obviously moving data to and from the flags register) are treated via machinery close to the Rename Mapping machinery.
- For other cases, the write to the register is allowed to proceed out of order, but the new value is retained in speculative storage. Reads of the new value (with an age stamp later than when the new value was stored) get told the new value; everything's consistent. Then at the point when the MSR instruction is ready to retire, the new value gets written to permanent special register storage and the machine state is changed.
So you don't have to pay serialization costs every time you change a minor register, or when you make a series of changes. Even when the system does require serialization, it can require this only after a sequence of changes has exhausted temporary storage.

There' s a second interesting issue here.
Any modern machine these days will crack more complex instructions into smaller instructions. For other CPUs this seems to be primarily about execution complexity, but Apple have generalized this into a powerful tool. The insight is that different stages of the pipeline may want to treat an instruction as a single unit or multiple units depending on exactly what the instruction is doing.
For example an ADDS or a LDP (load pair) want to be treated as two instructions for the purposes of register allocation (Rename) and deallocation (History File) but as a single instruction for the purposes of execute.
Conversely an instruction like ADD (shifted), which shifts one of its arguments before adding it, wants to be treated as a single instruction for the purposes of allocation and Retire, but to be split into two operations for execution. The patent is here: (2012) https://patents.google.com/patent/US9223577B2/ Processing multi-destination instruction in pipeline by splitting for single destination operations stage and merging for opcode execution operations stage.

One interesting aspect of this that I slid past you is that most instructions that split into two executed parts, like ADD (shifted), don't have to assign a physical register for the intermediate result (in this case the result of the shift); the intermediate result is just picked by the ADD off the bypass bus and no register is wasted. However, strangely, the EXTR instruction, which looks to my eyes looks like it could use this same bypass mechanism, for whatever reason (real issue? or just no-one ever optimized it?) does allocate a genuine intermediate register, with the extra resource allocation waste that implies.

ROB Duplicate Registers (MOV xn, xm)

This is hardly the end of the story. We know from other investigations that Apple provides zero-cycle moves. The idea of zero-cycle `MOV` is obvious, in that you "execute" the move merely by updating the register Mapping tables, rather than by sending an instruction through an integer execution port. What makes this not exactly trivial is that you now need a variety of extra book-keeping to track when it is safe to release a physical register (which now may have multiple duplicate users).
It's also worth noting that Apple seems to prioritize the zero-cycle aspect of this technology (also used to load Immediates into registers); it's nice that the technology allows the machine to act as though it has a few additional physical registers (because a `MOV` "reuses" a physical register, rather than allocating a new one) but that's not the priority, the priority is to reduce the latency of `MOV` in a chain of operations.

Apple appears to have used three successive solutions to this, and it's worth understanding all three because even when a solution is abandoned, some ideas from it live on in other aspects of the CPU.

The central problem around renamed registers is when they can be reused.
The first half of renaming is obvious – you need a table that maps architectural registers to the current physical register (or something equivalent, like the ROB slot of the instruction that will eventually produce the value of interest), and you need a pool of free registers from which you allocate a physical register for each successive destination register.
The difficult part is how you move registers into that free register pool – how can you, as cheaply as possible, and as early as possible, know when a physical register can be reused?
Conceptually there are three parts to when the physical register is no longer required:
- when the store to the register has been performed AND
- when all the readers of this physical register have executed AND
- when the logical register to which this physical register is mapped has been overwritten
Each of these three parts has multiple solutions, and it's a constant weighing of options as to which is best. For example for the second sub-problem, one can imagine options that include
- have a table that more or less notes the ROB slot for each reader of a physical register, and clears that entry as the appropriate ROB entry retires, until all entries are clear
- have a counter that goes up when a reader goes through Rename, and is decremented when a reader is Retired, ie a reference counter
- have a way to scan the entire table (make it a very wide matrix of bits) so that you can easily see if a column is bit-free vs has at least one bit set indicating a user

We see Apple working their way through variants of these ideas over time.
First off, an early patent is (2005) https://patents.google.com/patent/US20070050602A1 *Partially decoded register renamer*. Good luck understanding this patent first pass through! But it's actually interesting once you figure it out.
It refers to a core somewhat like A6, so out of order probably 3-wide or so, and primarily interested in the question of: how does the ROB communicate with the register allocation mechanism that some instructions have Retired? (Regardless of what solutions you adopt to the issues I raised earlier, this sort of communication is necessary.)
To understand the solution (which may be used, even today, in a fancier version) you need to know that the machine being described has a 64-entry ROB, treated as 16 "rows" which can each hold four instructions, and one row can Retire every cycle. So you want to communicate with the register allocation mechanism
- four instruction IDs that have retired
- look up those instruction IDs in a table (ie use a CAM structure) so you can set various flags for the appropriate registers
- but you don't want to pay the cost having four simultaneous CAMs
So the idea is, instead of indicating the Retiring instructions as four integers, you indicate it as a rowID (a 4bit value) and a mask (four bits) indicating which of the four instructions in that rowID are Retiring. Now you only have to use one CAM (for the rowID) and for each entry that matches the CAM you run a secondary test that it matches the appropriate bit in the bitMask.
It's a low-level patent, but shows the sort of tricks Apple is constantly playing, using whatever structure is present in a set of values (in this case, that Retiring instructionIDs are sequential) to reduce the workload.

Now let's look at what needs to be added if we want to use zero cycle moves (and thus allow duplicate logical registers associated with the same physical register).

The first Apple solution uses the RDA (register duplication array), a CAM that can describe ~8 registers that have been subject to duplication. If you create further duplicated registers beyond the CAM limits, the `MOV`'s execute like normal instructions.
Early Apple patents like 2012 https://patents.google.com/patent/US20130275720A1/ *Zero cycle move* discuss the RDA in various contexts.
(The RDA is still being referenced as of 2018 https://patents.google.com/patent/US10838729B1/, but that seems to be a lazy lawyer using obsolete boilerplater and diagrams!)

By 2014 we see https://patents.google.com/patent/US20160026463A1 *Zero cycle move using free list counts*, a scheme that (to my mind) makes a lot more sense, that trades the RDA (a CAM structure) for a few extra bits in each physical register tracking the number of users. So we have a reference-counting type system.

Finally we see in 2019 a third scheme https://patents.google.com/patent/US20210064376A1 *Last physical register reference scheme* which specifically states "It is noted that in the previously used register duplicate array (RDA) scheme, a new entry would have been created for PR6 with a reference

count of 2. However, in the new physical register last reference scheme, keeping track of the total number of references to a physical register is no longer necessary. Rather, it is sufficient to track only the last reference to the physical register. This is a more elegant scheme that is easier to implement, uses less area, and has higher performance."
(This same patent shows how Apple uses a History File.)

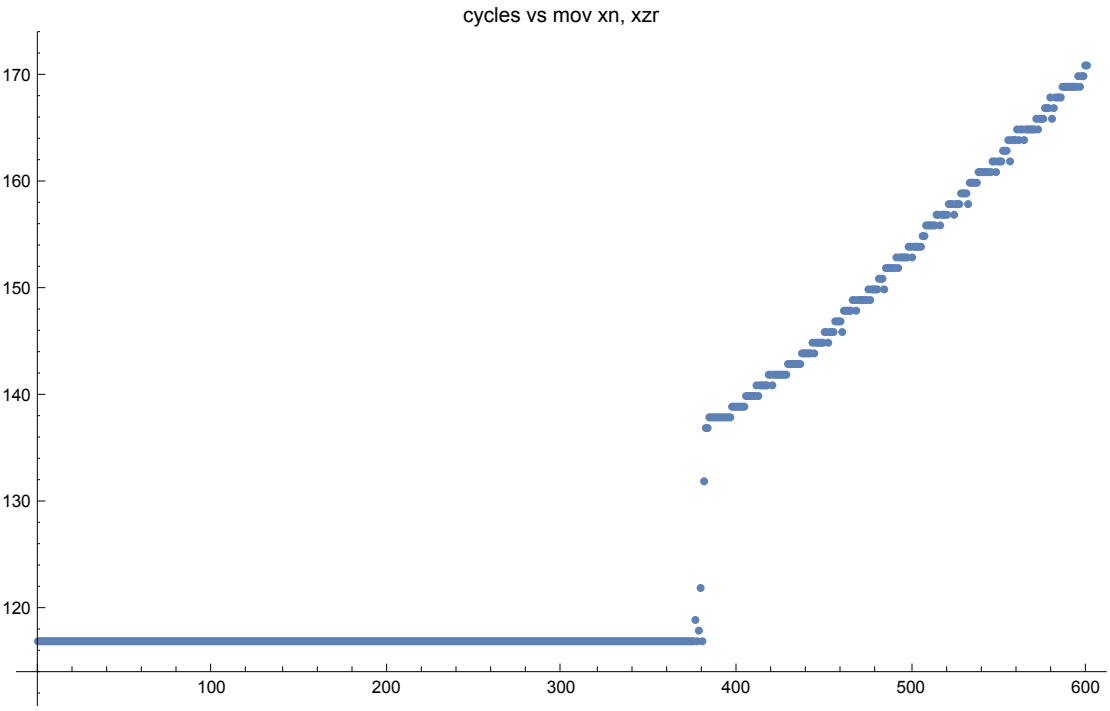
The M1 appears to implements this more elegant scheme, and appears to show none of the limits one might expect from the two prior schemes.
This third scheme imposes no limits on the number of duplications (`MOV`'s) that can occur, either on the number of different source registers that are duplicated, or the number of destinations to which a given source register can be duplicated. If you look at the patent, you may wonder how it is implemented -- it requires a frequent lookup against the Mapping table to see whether a physical register is the target of a mapping, and this looks like an expensive CAM lookup. My guess is that in fact the relevant structure is implemented as a bit matrix, a structure we will see again when we discuss Scheduling.

Let's see what we can learn experimentally about this duplication mechanism. Forget the zero-cycle aspects for now; what this means for resource allocation is that an operation copying one register to another (`MOV xn, xm`) does not allocate a new destination register as the physical register for `xn`; instead it simply updates the mapping of logical `xn` to point to `xm`'s current physical register. There are multiple ways this could be implemented (as the Apple patents already point out) so we want to probe possible ways in which this might fail, or behave suboptimally

xzr

First we'll try creating multiple references to `xzr`.

Out[59]=



In[ ]:=
```
{378,118},{379,122},{380,117},{381,132},{382,137}
```

In[ ]:=
$$\frac{600-382.}{171-130} = 5.317$$

So we see that about 380 `xzr` allocations can be performed.

Unexpectedly `MOV` from `xzr` is handled like a standard integer operation, executed in the int pipelines! We cannot see this from the graph, but the fact that the limit kicks in at ~380 (suggesting the allocation of a physical register) is indicative.
Looking at the performance counters tells us that `MOV` from `xzr` is, in fact, not eliminated.
We also see from the slop of the ramp that the rate appears to be about 6 `MOV`s/cycle, ie the standard integer execution rate.

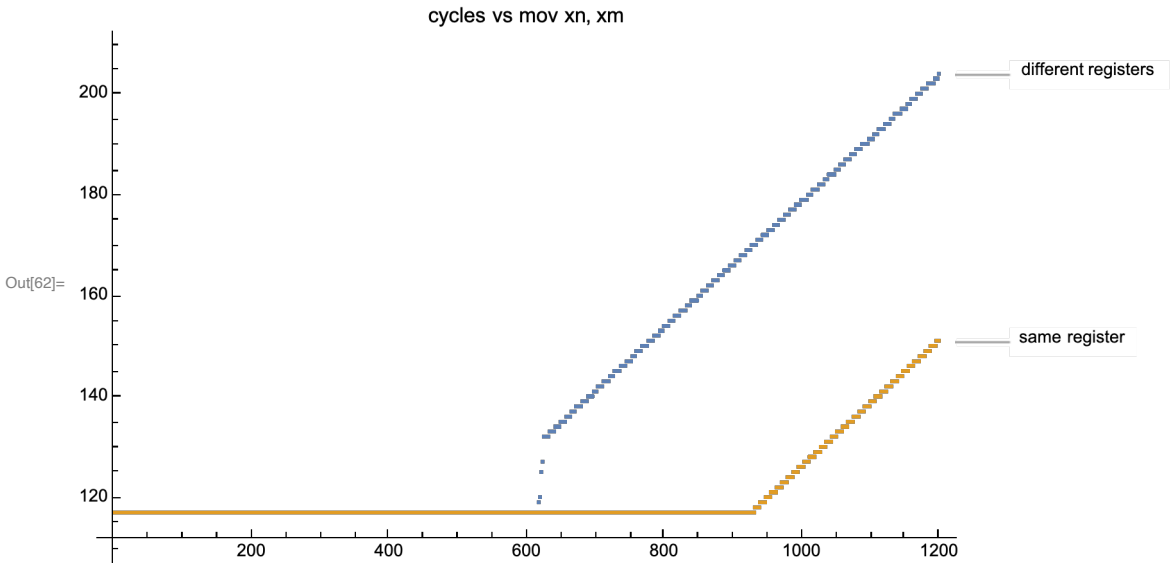
There may be something strange about the implementation here. But it may also be as simple as “don’t do that!”.
Apple has provided a fast optimal path for zero’ing registers via `MOV xn, #0`. To also special-case `xzr` would require extra work in decode, and does it make sense to spend that area, power, and cycle time rather than just telling people not to do that?
Still, as we shall see going forward, there’s more weirdness and sub-optimality in how `xzr` is implemented.

xn

But that's not the end of the story! Let's try a non-special register, like `MOV x0, x2`.
This case reaches 620 with no glitches (as opposed to the `xzr` case above)

(Just for fun I also included the case `MOV x0, x0`. This should be treated as a `NOP`, and that's in fact what we see. There's no reason your code should ever include `MOV xn, xn` !; but if it does, they behave just like `NOP`s, using no execution resources except ROB slots, and the curve jumping at around ~2300 when the ROB slots are exhausted. In the curve below you only see the curve start rising at 880 because at that point 880 NOPs/8 per cycle exceeds the 110 cycle delay.)

Out[62]=



In[]:=
```
{616,117},{618,119},{620,120},{622,125},{624,127},
```

In[]:=
$$\frac{1200-624.}{204-127} = 7.4805$$

So we're seeing not a limit at the number of physical register (380, as in the previous graph) but at the number of HF slots.

Once again to see the most interesting aspect of this, you also need to look at the PMC to see that no instruction issue occurs: all eight `MOV`'s (different register case) execute purely at Rename.

We can get some confirmation of this from the graph, since the throughput of the MOV's is ~8/cycle, which is Mapper/Rename throughput, not the 6 we'd expect if the instruction had to go through the integer pipes.

The same zero-cycle (and 8-wide!) behavior holds for FP/SIMD registers (eg `MOV.16B v0, v1`).

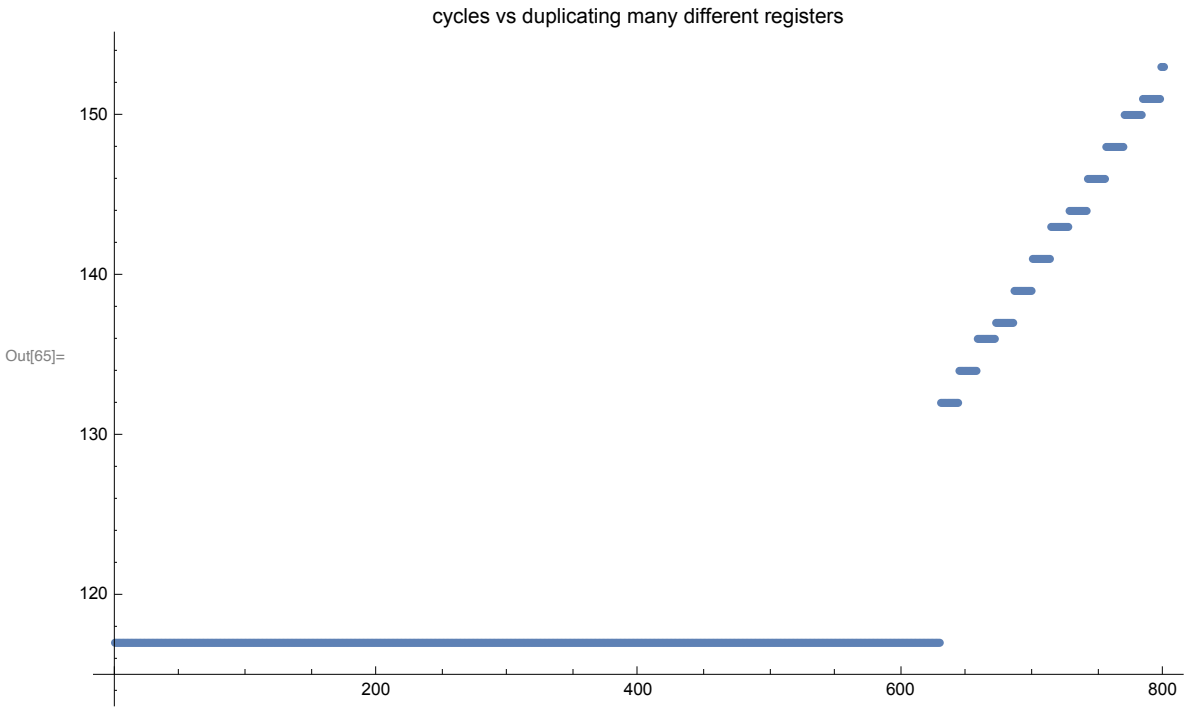
We would also expect that, since it's the common pool of History File slots used for both integer `MOV` and FP/SIMD `MOV.16B`, when we pair the `MOV` with a `MOV.16B`, the jump will be at ~620/2=~310, and that's exactly what we see.

What further aspects of duplication can we test?

The RDA (remember, the subject of the 2012 Apple patent) can only hold a limited number of entries, each describing the number of duplicated references to a register. We can use this to test if Apple is, in fact, still using an RDA.

If we create as many duplicates as possible (ie duplicate multiple different registers) presumably we will flood it, at which point duplications will have to happen via standard execution, not by register

rename.
We create a probe that consists of fifteen instructions
(`mov` x_i, x_{i-1} for i..1..29)
And replicate this (N/15) times.
(We can't do much with `x31` which is xzr/sp and special, so we are limited to 15 pairings).



Out[65]=

In[•]:=
```
{628,117},{629,117},{630,132},{631,132}
```

The result shows the expected jump (extremely sharp!) when the HF is maxed out at 630.
More important, however, is that the throughput never deviates from 8 instructions/cycle, so the Rename magic is able to continue working with so many different duplicates established; if there is a limit to the number of duplicates, it's more than fifteen.
(Remember, what we are testing for is the presence of an RDA, a Register Duplicate Array, that can hold only a limited number of Register Duplicates, ie source physical registers that have been mapped to more than one logical register.)

There are other ways to run the test, like performing fourteen one time duplications of the form mov xi, x(i-1), to exhaust the RDA; then following with N mov x29, x28. Same results.
We can also add 15 floating point duplications, for a total pool of 30 duplicated source registers, no difference.
So, yeah, seems like no RDA is being used.

subregisters

What if we try subregisters, eg `MOV wn, wm`, or `MOV.8B vn, vm`, or `MOV dn, dm`?
On the M1 we don't get any of these fancy tricks when copying subregisters . Such copying is slightly

trickier for various reasons, the most obvious of which is rules for how the high bits have to be handled; but it seems like it should be doable, (eg via an additional “zero the high bits” flag in the logical->physical map, or that same bit set in the physicalRegisterID [this concept will make more sense when we discuss immediates].)
Maybe it’s not worth doing? (But it seems like it should be, especially for code that’s using lot’s of floats or doubles but is not vectorizable). So maybe it’s on the drawing board for a later CPU?

There is something of a historical precedent:
Suppose you have a machine (like the A7) for which you expect both 64-bit and 32-bit wide registers to be common. One way you might arrange things is that your register storage behaves as two side by side banks of 32-bit wide registers, with the ability to allocate a row of two registers as a single unit (perhaps by a high bit set in the registerID). This would give you both some number of 64-bit registers while also giving you more 32-bit registers during the transition period while 32-bit software is common.
We see a (kinda sorta) version of it here (2010) https://patents.google.com/patent/US20120110305A1 *Register Renamer that Handles Multiple Register Sizes Aliased to the Same Storage Locations*, dealing with the old ARMv7 way of aliasing floats, doubles and neon registers. Clearly this is obsolete; but it deals with a slightly more complex version of the problem.
Likewise (2012) https://patents.google.com/patent/US9317285B2 *Instruction set architecture mode dependent sub-size access of register with associated status indication* discusses how power can be saved when reading or writing a 32b (W) register, and the use of an extra bit in the logical to physical mapping table to note that the logical user is utilizing the 32b subset of the referenced 64b physical register.

Generically, it seems like there is scope here for potentially massive register allocation improvement.
Imagine instead of two register files for int (each 64b wide) and SIMD (each 128b wide) we have a single register pool consisting of rows of four 32b units. Register allocation would consist of providing (appropriately aligned) a full row (SIMD register), a half row (double or x register) or quarter row (single or w register). Conceptually as raw storage this is feasible, one just has to be a little careful about how the free lists are treated; and a few high bits associated with each physical registerID clarifying the width of the item that is being extracted.
Where things become slightly tricky is in register aliasing (eg you write a signed value to a w register, then read that as an x register).

In the integer case this is fairly easy to handle; you just need to give the register file some logic so that when values flow in or out they are appropriately shifted and sign-extended, and you need an extra bit associated with each 32-bits to clarify how the sign extension should be performed on the outflow.
I haven’t bothered to look at the precise neon SIMD/DOUBLE/FLOAT aliasing rules to know if there’s a difficult problem on that side. But the vision is tantalizing! A unified register pool means that instead of being limited by just the 400 or so integer registers, code that used no fp registers would have many more int registers available, and code that used many w registers would have a pool of even more registers available.

(Of course to get full value out of this, the size of the history file needs to be increased, but that doesn't seem like a problematic structure to grow.)

There is also the upcoming issue of SVE/2 and the A15/M2. It seems like a natural way to try to handle these would be
- to have Apple support SVE/2 at a 256b width
- to have a register file that can provide 256b registers, but also provide 128b registers for neon during the transition period, and to try to not be wasting the area of the 256b registers if only half is being used by most code.
That by itself, is better than nothing, but a totally unified integer/SIMD file supporting widths from 256b down to 32b would clearly be optimal!
We'll see.

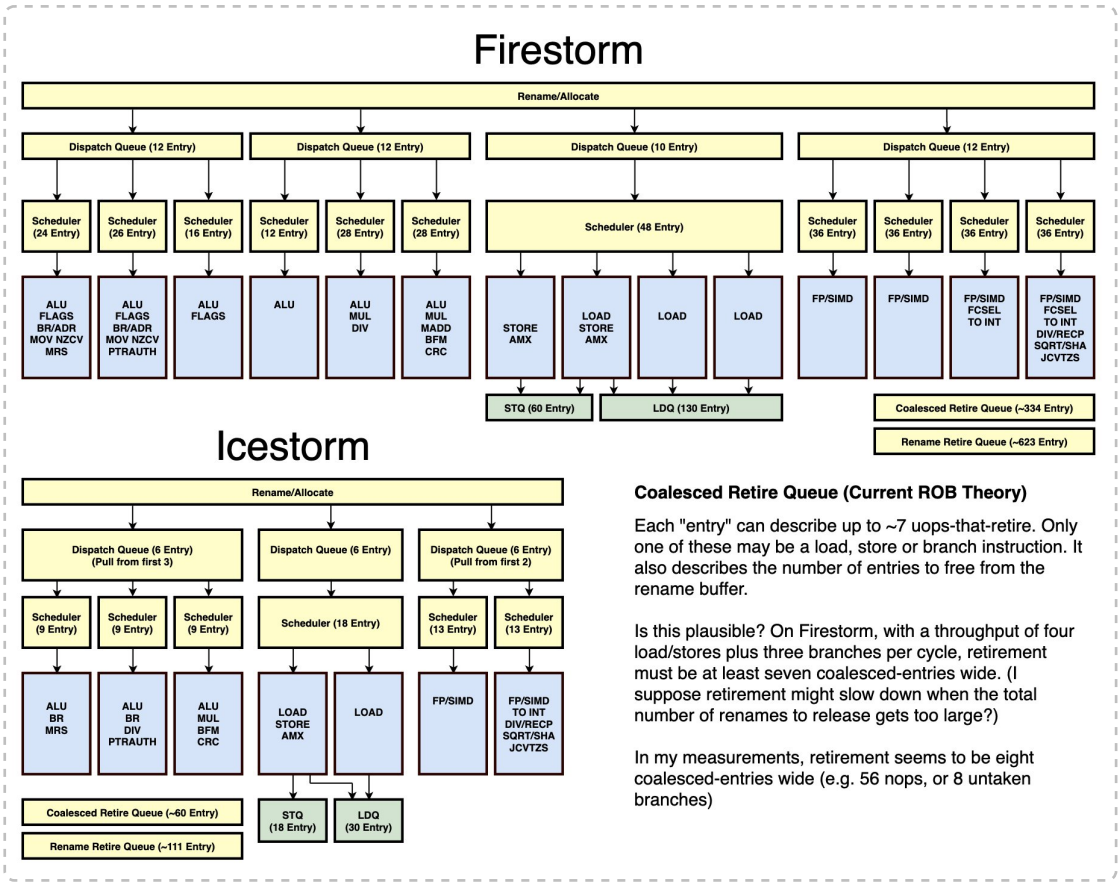
Instruction Scheduling

Going forward, you may want to keep in mind this diagram by Dougall. The ROB stuff is provisional (and I think better explained below) but the basic ideas and numbers match my investigations.
And what he calls Dispatch Queues should, I suspect, be thought of as Buffers. Buffers are cheaper than Queues because they don't make the same ordering promises. I suspect the Apple Dispatch Buffers can lose ordering (in cases where it doesn't matter much, namely when the Scheduler Queues are so filled up that the Dispatch Buffers are more than just pass-through).

Note that this diagram should be treated as provisional. In particular (eventually both of these will be explained/justified below)
- the monolithic 48 entry Load/Store Scheduling Queue is in fact 4 separate 12 entry queues, just like the other cases
- the other entries apart from load/store are probably about 2x too large (that is, the size of say the two MUL queues is probably ~13 each), likewise the FP queues probably 18 each. Essentially (as will be explained below) while each queue feeds a primary execution unit, the queues are paired, and if one queue does not have a runnable instruction, it can accept the "second choice" runnable instruction from the companion queue; thus for many testing purposes it will look like the queue feeding a particular execution unit is twice as large as it is, because the "amount of scheduling space" is the sum of the two queue sizes.
xxx I have not yet had time to probe the full details of this pairing beyond having seen it in action in the case of load store.
Given the numbers, an obvious guess is that on the integer side
- the two MUL queues are paired (each 13 in size)
- two two BR queues are paired (maybe one of size 12, one of size 14, or both of size 13)
- the ALU/FLAGS queue and the ALU queue stand alone and are not paired.
On the load/store side, the store queue is paired with the ambidextrous (store+load) queue.

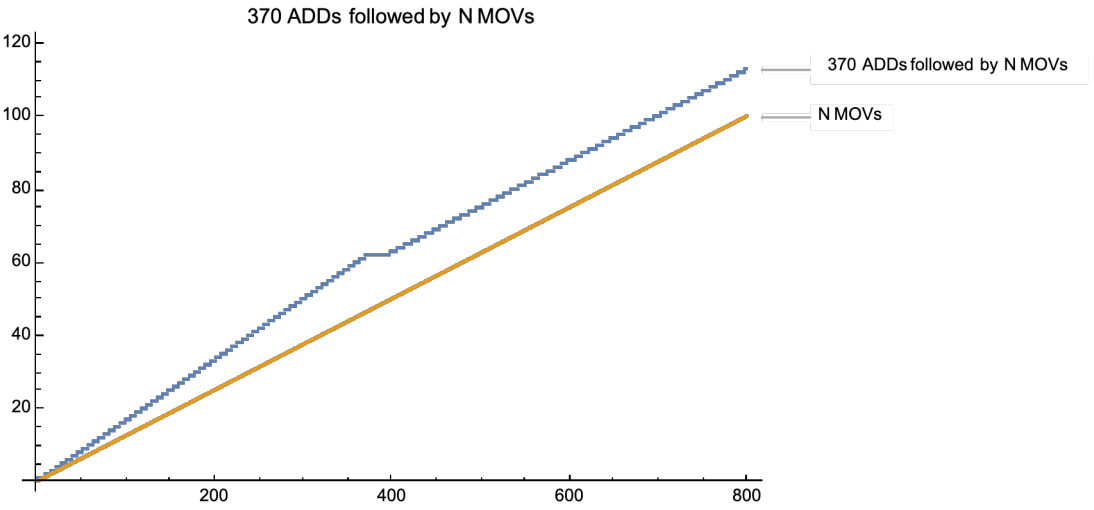
I would guess that the two FPCSEL queues are paired, but have not tested that.

It makes sense to pair together queues as similar as possible (except where prevented by the fact of the two different integer Dispatch Buffers) because that gives you the maximum amount of sharing flexibility; whichever queue you dump an instruction in, if the other queue can't find a runnable instruction, two instructions can still be scheduled. This doesn't work as well when one type of instruction can only go into one execution unit – which is the case for the store vs ambidextrous unit, which is how I discovered the phenomenon.



Consider now the following experiment. Suppose we use up almost all the int physical registers (via `ADD`s) then try to perform some `MOV`'s. What we are interested in is the transition between these two types of instruction, one of which is executed in the OoO pipeline, one of which is executed in Rename.

Out[67]=



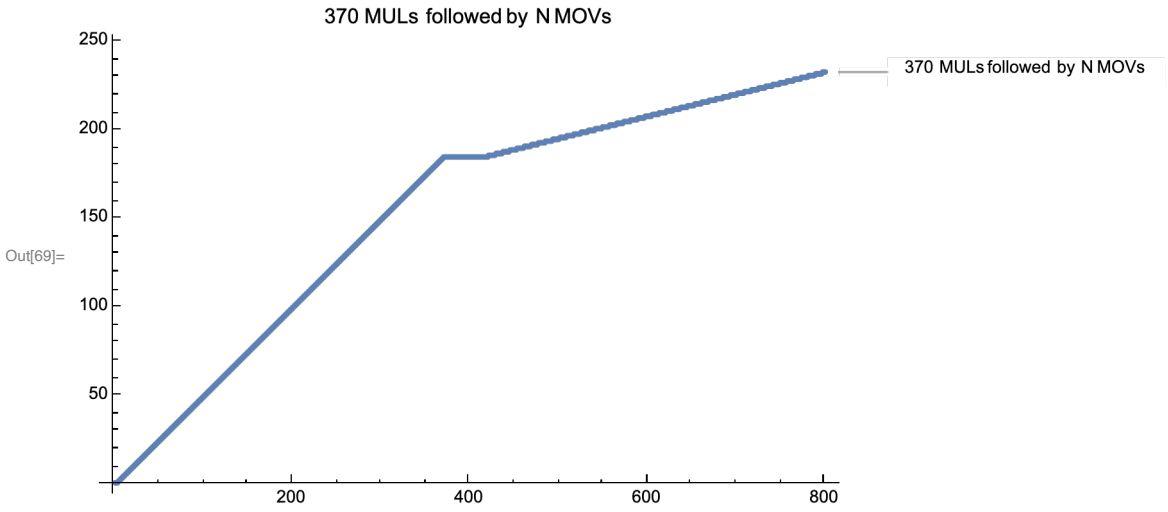
The low-N and high-N parts of this curve are easy enough to understand. We have no delay block. The first segment of the graph corresponds to the 370 ADDs, processed at a slope of 6/cycle; these are followed by the MOVs, processed at 8/cycle, and the difference in slope is visible, compared to the gold curve which is just MOVs without the initial ADDs.

More significant for us is the flat region from 370 to about 397
What's happening there? This tells us something about the int instruction scheduling.
To simplify (this will all make sense after you read below then come back here)
- Rename can generate 8 instructions (8 ADD's) per cycle.
- These can be moved (8 per cycle) into a Dispatch Buffer
- But only 6 instructions per cycle can leave the (integer) Dispatch Buffer
- And those 6 instructions can immediately execute, so they do not fill up the Scheduling Queues
- Hence every cycle the Dispatch Buffer gains two instructions, until it is full
- The integer Dispatch Buffer has a capacity of ~24 instructions, so it soon fills up.
- Once we add a few MOV's to the end of the instructions to be executed, at first those MOV's take zero cycles to execute
+ more precisely they execute in Rename at the same time as the ADD's that were in the Dispatch Buffer are drained
+ so we get 4 free cycles (24 buffer size/6 ADDs per cycle) of NOPs that will not add to our cycle count
+ and 4 cycles of NOPs is 32 free NOPs before the additional NOPs start to increase the cycles/loop iteration

This is the basic idea with the one final tweak being that integer code actually uses two buffers, each of size 12.
(BTW there's no significance to the use of 370 ADDs, you just need "enough" where enough is more than 12 cycles*8/cycle=96)

If you understand that, then you should understand this:

Out[69]=



This is very much the same idea, only using `MUL x0, x5, x5` as the probe. What should we expect?
We get an initial slope of two instructions per cycle (only two execution units support multiply, but they are fully pipelined).
The flat region runs for {370,185} to {420,185}, so we get about 50 free `NOP`s. Does this make sense? Yes! The Dispatch Buffer that feeds the two `MUL` units has a capacity of 12, so will drain in 6 cycles. 6 cycles allows 48 `MOV`s to execute in Rename.
So we validate both our understanding and that there are two Dispatch Buffers feeding the six integer Scheduling Queues, as in the diagram.

M1's Implementation of instruction scheduling

The machine is 8-wide all the way though to Rename; and throws out 8 ADD instructions per cycle into the Scheduling Pool. By Scheduling Pool we're being deliberately vague, but mean the idea of some pool of instructions between Rename and Execute from which instructions are Scheduled for Execution in an out-of-order manner, as their dependencies are resolved.
One's natural assumption, the usual model, is a single level scheduling pool (called something like a Scheduling Queue) but it's time to revise that.

Recall that the scheduling queue holds instructions that can't yet execute because at least one of their dependencies is not yet available; for example the instruction may be waiting on a load, or it may be one of the final instructions in a chain of ten or fifteen sequentially dependent instructions.
We want the scheduling queue to be as large as possible, as this allows the CPU every cycle to look over more instructions to find something to execute. Or to put it differently, the larger the scheduling queue the more independent chains of sequentially dependent instructions you can run in parallel, so the more ILP you can extract.

separate scheduling queues

However scheduling queues are phenomenally expensive in power! (Every cycle you need to scan the entire collection of instructions to check which have become runnable, then of those instructions choose the oldest for execution.)
The most obvious solution, at least as a first step, is to at least split the single scheduling queue into multiple queues. This means that you sometimes waste some queue space (if you're doing only integer computation, all that space devoted to the fp scheduling queue is wasted) but it allows the sum of all the queues to be larger for the same, or lower, power.

Apple takes this idea to an extreme by adopting a queue in front of each execution unit, rather than, perhaps, a common queue for integer, a separate common queue for fp, and a third separate common scheduling queue for load/store. But a problem with these multiple queues now is that you want to ensure that they are load-balanced. Obviously some integer instructions can only go to one or two pipelines (there are only two branch execution units, only one integer divide unit), while other integer instructions like ADD can go to any of the six integer execution unit queues. You can't do anything about the specific pipeline(s) that some instruction are forced to go down. But you can balance other instructions as much as possible around those restrictions.

dispatch buffers

Apple's solution to this (which has accumulated other benefits over time) is to install a Dispatch Buffer, a secondary instruction storage pool, between Rename and the Scheduling Queues.
The initial goal of this Buffer was load balancing -- after instructions with strict requirements are sent to the appropriate queues, the remaining instructions are sent out according to which queues are least full, so that overall queue fullness remains even across all queues.

Of course the the Dispatch buffer also acts as a secondary storage for instructions, a way to put them somewhere and allow Decode and Rename to keep going, even if all the integer queues are full and can accept no more instructions. A Dispatch buffer is cheap because it doesn't have the ordering requirements of a Scheduling queue (which instructions are oldest?) and doesn't have to be scanned every cycle to figure out which instructions have become runnable. AMD uses a Dispatch Buffer for this reason, for FP/SIMD instructions. You can see the AMD case here: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2 (Look for the big diagram, then in the FP section for the Non-Scheduling Queue)

However a Dispatch Buffer is also imperfect for these same reasons -- if Scheduling Queues all fill up, and then the Dispatch Buffer, so that this Buffer contains more than just a few instructions, then, once execution starts flowing again, it's somewhat random which instructions will be the first to be moved from the Dispatch Buffer to the Scheduling Queues and on to execution; and choosing the wrong instructions from the Dispatch Buffer might delay the execution of critical instructions for a few cycles. So a Dispatch Buffer is not absolutely free storage that should grow as large as we like; we want it to be large, but we don't want to have to pay the side effects (poor scheduling) of it being large! It needs to be kept to a limited size so that, even if there is some degree of sub-optimal issuing of instructions to

the Scheduling queues, there's also limited delay that can occur.

There's one other, less obvious aspect to the Dispatch Buffers: each one can accept up to 8 instructions from Rename. Assume you have a long run of integer instructions. Each of the six integer queues can only accept one instruction per cycle, so in the absence of a buffer, Rename would only be able to clear 6 instructions per cycle. The machine wouldn't halt, but the front-end would be forced to run 6-wide rather than 8-wide. It's even worse if there's a long run of FP or load/store instructions where only 4 instructions could be enqueued per cycle.
Each Dispatch Buffer can accept a full eight instructions per cycle, thus it can clear Rename for the next eight instructions, and this can continue for at least a few cycles, though obviously not indefinitely. As I said about designing not just for the mean of a program, but for the standard deviation...

evolution of Apple's scheduling queue

The Scheduling Queue is another case where we can see the evolution of Apple's thinking over time. (Recall what I said, that Apple use the term Reservation Station for what I'm calling Scheduling Queues.)

2013 (two level scheduling -- dispatch + scheduling queues)

We start with (2013) https://patents.google.com/patent/US9336003B2 *Multi-level dispatch for a super-scalar processor*, which describes a basic two level scheme, where the main point is load-balancing. BUT with the secondary point that the buffers are capable of accepting up to eight instructions (whereas queues only need to accept one instruction per cycle). These details matter! Most hardware structures grow quadratically in area as you increase their inflow or outflow. So there's a real win if you can dump all this quadratic complexity (eight-wide inflow) onto a very simple structure, a buffer with no ordering or other properties, while keeping the queues as 1-in, 1-out per cycle.

2015 (non-shifting queue, based on relative age field)

This is somewhat augmented with (2015) https://patents.google.com/patent/US20170024205A1 *Non-shifting reservation station*, a non-shifting Scheduling Queue which I will describe soon, and the technical companion patent (2016) https://patents.google.com/patent/US10120690B1 *Reservation station early age indicator generation*.
To understand this 2015 patent, recall that the usual way a Scheduling Queue is implemented is exactly as a physical queue. Using a physical queue means that temporal ordering is trivially preserved -- it's easy to see which instructions are oldest (at the head of the queue) vs youngest (at the tail of the queue). But a queue uses a lot of power because it has to be "collapsing"; that is, every time an instruction is scheduled from the middle of the queue, you have to move all the other values along the queue to remove that vacancy and open up space at the end of the queue.
Now you can imagine various possible ways to reduces the cost (leave the holes as long as possible, ie only collapse when you have to; maybe use indirection, like a linked list rather than a linear sequence of storage; ...). The Apple solution is to use an age matrix, which is, just like aspects of the 2019 History File patent, a structure that is large (by the standards of tech many years ago -- now transistors are

cheap!) but low power and has low complexity wiring. The age matrix tracks the temporal ordering of the instructions while not requiring them to continually be moved from one slot to another.
I urge you to look at this one; it's fairly easy to understand but so neat!
Suppose a particular scheduling queue has 30 entries. Then associated with each entry is a 30 bit vector, each bit of which represents one of the queue entries. If a bit is set to 1, that means this queue entry is older than the entry represented by the bit. It's a fairly simple simple algorithm when to flip bits to one then back to zero, and the flipping doesn't have to happen very often (ie low energy).
(A similar solution is used for the Load/Store Queue as we'll discuss when we get to Load/Store.)

2016 (earlier testing of relatives ages, allowing larger queues)

Now we have this 30 bit vector indicating age of each slot relative to others lots. OK, *in principle* we understand how it encodes the age info, but what do you actually do with that info? That's what the 2016 patent covers, somewhat.
One can imagine at least two ways of using these age vectors. One idea is you perform a pop count on each vector, then find the slot with the highest popcount. Conceptually easy, but it assumes that we have fast cheap circuits that perform popcount, and perform "find largest in a set of 30 integers"...
Alternatively we can use good old divide-and-conquer. Compare adjacent pairs of entries to find which is older (which is a simple test of the bits at the two appropriate positions in the two bitvectors), and replicate that 5 times (binary log of 2). The patent points out that you can do at least the first round of this in an earlier cycle (relative age won't change, but you may have to test all slots, you can't mask out slots that aren't executable); and by doing one or more of these relative age comparisons in an earlier cycle you have to do fewer (easier to meet cycle time) in the next cycle.
One interesting aspect of this recursive structure is that really what you are trying to calculate is not actually "oldest runnable instruction" it is "best choice for runnable instruction". You could replace/augment the age bitvector with something encoding criticality information while still using this same tournament "compare successive pairs" structure to get a criticality-based scheduler; perhaps via something as simple (at least at first) as adding a single extra "critical" bit to the relative age bitvector that overrides age if it's set for only one slot, otherwise it's ignored and age wins. That allows us to start with a basic criticality predictor that generates just a single yes/no prediction.

2017 (pairing scheduling queues and issuing from either)

Finally we get (2017) https://patents.google.com/patent/US10983799B1 *Selection of instructions to issue in a processor*.

Consider on the one hand Intel's scheduling solution which is (more or less) to use a single large age-ordered queue.
In theory this gives you total information – you can see all the instructions, all their ages, and can schedule the oldest ready instructions. That may sound ideal but
- you pay a substantial cost for that in energy, in inability to scale, and in lack of time to implement various smarts

- you're optimizing for something (oldest ready instructions) that is not even *really* what you want (see my mention a few pages below of *criticality*).

The Apple solution runs in the opposite direction, starting with the simplest solution that will work (multiple queues, each independently scheduling only their contents). Clearly such a solution can result in multiple types of imbalance, and so we see every year simple tweaks that substantially improves the balancing while not costing much in area or energy. So
the Dispatch Buffer tries to keep a collection of queue reasonably balanced in their fullness, and
- the two patents described above track relative age (so that, approximately, albeit not perfectly, oldest ready instructions schedule first).

However we could still have an imbalance where queue A has two instructions ready to execute while queue B has no instructions ready to execute. That's not ideal! Can we do anything easy to fix this? Consider how the age mechanism I described above works. Essentially we have multiple rounds of "find the better instruction [by runnability and age] from comparing these two instructions". Each such round take in two candidates and outputs one candidate.
So suppose that at the very last round, we also offer the second choice candidate from queue A to queue B (and vice versa).
Queue B will accept the second choice candidate as better than what it has (which is nothing!) and will schedule the second choice candidate from queue A to execution unit B!
Obviously one needs a few interlocks and tests and so on to ensure that you don't land up scheduling a Divide instruction to a queue without a divider; but it's one more easy'ish tweak to the pre-existing system to improve load balancing and throughput a little more.

2017 (splitting scheduling queue into a fast front half and a slower back half; not implemented? or only for FP?)

Finally we get (2017) https://patents.google.com/patent/US10452434B1 *Hierarchical reservation station*.
Honestly I don't get how this one fits into the general pattern of Scheduling Queues, or any of our test results. Maybe it represents a set of ideas that were ultimately discarded? Or that are only used for FP?

The problem this claims to be solving is that age/readiness comparing the full set of instructions in the Scheduling Queue may not fit within cycle time. The *early age* (2016) scheme was one of dealing with that. But this suggests a different scheme.
The idea is that we split the Scheduling Queue into two parts, primary and secondary. We run the scheduling machinery on both halves.
- The results for the primary queue are some number of instructions to be issued (oldest ready instructions);
- the results for the secondary queue are both
+ some number of instructions that could be executed (oldest ready instructions) and

+ a second array, of *oldest instructions*, to be moved to primary on the issue of the newly scheduled instructions from primary.

Of course I don't know, but the logic looks to me like the queue they had in mind
- was split into primary and secondary,
- initially all secondary did was figure out which instruction each cycle to move to primary;
- then the inventors realized that by adding a little logic, they could also issue instructions from secondary if primary could not supply enough instructions.

But there's a lot I don't get about how this fits into the 2015/2016 scheme! We don't want to be moving instructions around (that's the point of the aging bits) but this scheme seems to require moving instructions from primary to secondary.
Could the primary and secondary queues be interleaved, or side by side, so that the distance moved (from one slot to the next) is not very large?

Perhaps this scheme is specific to FP?
Look at the sizes of the different Scheduling Queues. My rough heuristic for the size of a Scheduling Queue is that it wants to be able to hold enough instructions so that something can always be found to Issue from that Queue. If the Queue is too small, then an instruction that could be Issued is unable to Issue because it’s hiding up in Dispatch or even in Rename, invisible to the Scheduler. But the Queue costs power, and if it too large, those extra instructions are always just sitting in the second half of the Queue are doing nothing for performance because an instruction ready for Issue can always be found in the first half.
This heuristic suggests that the optimal length of a queue will vary depending on both
- how much (on average) instructions tend to depend on the previous instruction (chained dependencies mean the later instruction has to wait in the queue) and
- on the average latency of these instructions being waited upon.

This seems to match the pattern we see for Apple.
In particular for FP/SIMD both long dependency chains and long instruction latencies are common, hence we see each Scheduler Queue as a hefty 18 entries in size. So maybe this patent is more or less specific to FP, splitting these 18 entry queues into two 9-entry halves, or a 12 entry front-section and 6 entry back section, that are separately scheduled as per the patent?

dependency tracking

register-based dependencies

The other part of handling Scheduling is tracking instruction dependencies. The traditional way to do this is through physical registers: `ADD x0, x1, x2` depends on x1 and x2. We map those to p11 and p12, and route the instruction to scheduling as `ADD p10, p11, p12.` Then Scheduling won't

consider the instruction for Issue until P11 and p12 both become valid.

This physical register based scheme is an obvious grandchild (after various improvements and tweaks along the way) of the original Tomasulo scheme from the 1960s. You may enjoy reading (1987) https://www.cs.auckland.ac.nz/compsci703s1c/resources/Sohi.pdf *Instruction Issue Logic for High Performance Interruptable Pipelined Processors*; an *old* paper which is interesting precisely because it's written at such an early time, talking about the precise mechanics of (one early version of) how you do this, before the details became routinized as just the words Rename and Scheduling.

A similarly old paper (2000) which at least introduces many of the ideas I've been discussing about how to handle register renaming is
(2000) https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.3852&rep=rep1&type=pdf *The Design Space of Register Renaming Techniques*.
These historical papers are helpful in two ways
• they explain the problem that needs to be solved and
• reading them you helps you understand the terminology and mindset of comments you find on the internet.
However it's important to realize just how long ago those papers were, in a Moore's Law world. What was optimal and state of the art in 2000 may make zero sense in 2020.

This register-based scheme works, but isn't the only way of doing things. The biggest problem is that, for today's version of Tomasulo-type schemes to work, registers have to be *early-allocated*, ie have to be allocated at an in-order stage of the pipepline (traditionally in the Rename stage). And the problem with early allocation is that a very expensive and limited resource (a physical register) is now locked up from Rename until the instruction Retires. But if you think about it, the register is only required from the point at which the instruction completes Execution (to hold the instruction result); all the time before Execution (while the instruction is waiting to be scheduled) that physical register is achieving nothing useful; its only purpose is for its registerID to establish instruction dependencies.

So: can we *late-allocate* a register, at the point of instruction execution (in fact completion, when we want to write out the result!) rather than at Rename?
Yes – if we have some alterative way of tracking instruction dependencies...
One version of this, an extension of Tomasulo, uses *virtual registers* (1999) https://upcommons.upc.edu/bitstream/handle/2117/101362/00809456.pdf *Delaying Physical Register Allocation Through Virtual-Physical Registers*.

The flip side of late register allocation is *early register release*, ie releasing a physical register as soon as it's no longer required (because all potential users have executed) rather than waiting until Retire. The details of how this could be done (not optimally, but picking up some of the value) appear in the Haithim Akkary paper referred to earlier, describing Checkpoints and other instruction for very large OoO processors.

As far as I know no commercial CPU (even Apple) currently uses register late allocation or aggressive early release. However IBM (in recent POWER) does use late-allocation of load/store queue slots.
Apple definitely uses at least one of late-allocate or early-release for load/store queue slots, probably early release. (A 2019 patent described below suggests a scheme with traditional allocation of LSQ slots, but early release.)

instruction-based dependencies

An alternative is, rather than saying that `ADD x0, x1, x2` depends on (the physical registers mapping to) x1 and x2; to say that the `ADD` depends on the two previous instructions, call them I27 and I35, that will produce x1 and x2 respectively.
This could be implemented by, eg, having I27 and I35 in fact refer to the ROB slots of the relevant instructions, and essentially instead of marking a physical register as valid at Execution completion, the ROB slot is marked as valid. Let's call these numbers associated with previous instructions SCH#'s. Thus associated with each instruction in scheduling is a Dependency Vector, which describes the instructions that have to complete before this instruction can execute. (This may sound more sophisticated than virtual registers, but in fact it harkens back to the mid-1990s when, as I've already described, something like the Pentium III attached a physical register to each ROB slot, and so marking a physical register as valid was the same thing as marking a ROB slot as valid.)

The Apple Scheduler patent is (2016) https://patents.google.com/patent/US11036514B1. To understand it, we need some history.

The original 1960s Tomasulo renaming stuff, already mentioned, is from an utterly different world in terms of transistor budgets and the relative speed of different components. (You will see, in the history, an explanation of some of the SCH# naming conventions in Apple patents, and will see why they refer in the patent to possible ways of doing things that clearly seem dumb!) For our purposes the aspects of Tomasulo that matter are
- Registers were renamed via a mapping table
- A destination register was NOT allocated right away.
+ the scarce resource was a slot in the Reservation Station and physical registers.
- When a slot in a reservation station became available, a waiting op was moved to that slot. Call the address of that slot a "tag"
+ it should be clear that this tag conceptually functions as an ID for an operand for all subsequent instructions. It acts *like* a physical register ID.
- When an instruction in the reservation station is able to execute (dependencies satisfied) a destination register is allocated.
+ that register is marked with the tag we mentioned earlier.
+ so at this point we have a destination register marked with the tag of an instruction, and possible dependent instructions marked with such a tag
- Once the instruction has executed, it broadcasts its result on the bypass bus along with the tag.
+ dependent instructions in the reservation station look for the matching tag and grab the result (their

operand value) from the bypass bus.
+ likewise the register file looks for the matching tag, and stores the result in that physical register.

Compare all this to the model you should have of renaming today. We
- Provide the destination register at Rename, not much later at Issue. This means the destination register is idle for those cycles between Rename and Issue (as we have already discussed) but it means much of this tagging nonsense goes away.
- Instead of a tag (allocated when the instruction moves into the Scheduling Queue) dependency can be built upon the destination register's physical registerID.
- An instruction has as unresolved dependencies, not a tag (the ID of an instruction that has yet to execute, attached to the result of the execution on the bus) but a physical registerID on an invalid register. When an instruction executes, it dumps the result (plus the destination physical registerID) on the bypass bus. The destinationID tells the bus where to place the value in the register file (so no tag comparison needed there), but *all* waiting instructions in *all* scheduling queues still have to look at *all* the physical registerIDs flowing over the bypass bus to see if one of those registerIDs matches a dependency so that they can now be marked ready to execute.
You can see that the two procedures are very similar, just with the role of the tag being replaced by the destination physical registerID. But you can also see that we have a huge scaling problem here, trying to compare all the possible physicalRegisterIDs generated in a cycle (there are 14 units that could all generate a result in one cycle on the M1, and some of them [think load pair, or an instruction like ADDS that also sets flags, could generate two results) with the physical registerIDs that are the dependencies for every instruction in every queue. We need to rethink the problem based on today's concerns, not the concerns of the late 60's.

For now assume that the earlier instruction on which we're dependent is identified by ROB slot. Given ~2300 ROB slots, that would mean ~2300 instructions on which we could depend. But think about what we are trying to do.
Suppose instruction i0 depends on architectural register x1, which is mapped to physical register p2.
At the point where i0 enters the scheduling queue, p2 can either be valid or invalid.
If p2 is valid, then i0 has a dependency on p2, but it's not an "op-dependency". As far as *scheduling* is concerned, i0 can execute right away, because p2 is known.
On the other hand, if p2 is invalid, then there is an op-dependency on p2, and i0 cannot execute until the producer of p2 has executed. Thus for *scheduling purposes* we want to know what *pending* instructions i0 is dependent on.
This simplifies the problem; we don't actually care about every instruction in the ROB, we only care about instructions that are in the scheduling queues and haven't yet executed; an instruction that executed before i0 even entered the scheduling queue has stored its result in p2 and is not relevant for scheduling.
So the number of instructions we need to track for scheduling is really the number of instructions that are waiting to execute ahead of this one, which is a maximum of ~400 (worst case total occupancy of all the scheduling queues and dispatch buffers).

How are SCH#'s implemented? I think the original implementation (when the SCH# terminology was born) had no Dispatch Buffers, and had the Mapper allocate an instruction slot in a particular scheduling queue. Thus the SCH# could literally refer to the slot of the queue in which the instruction was placed.
But the current implementation is, I would guess, much like register allocation. Imagine a pool of 400 numbers, 0..399. Each is marked busy or not. Map allocates each instruction a free index from this list, marking the index as busy; at instruction is removed from the scheduling queue, that same index reverts to free, and that index is used to broadcast "instruction with this index has generated a result". This gives every instruction a unique persistent ID from the beginning to the end of its scheduling career, from a set that's as large as required but no larger.

bitvector-based dependencies

Now how is dependency implemented? You might imagine something like an array for each instruction that holds a maximum of maybe three SCH#s. That seems obvious, but If we trust the patents, the obvious answer is wrong.
It appears to be that each instruction has an associated dependency bitvector, with a bit set for each dependent SCH#. This means a dependency vector is ~400 bits long, though only a few (perhaps two or three) of those bits are set.
There are a *lot* of patents that suggest this bitvector implementation, eg (2006) https://patents.google.com/patent/US7647518B2 *Replay reduction for power saving*, which says "The dependency vector may comprise a bit for each entry in the buffer 42" (buffer 42 is essentially the scheduling queues).
The most detailed explanation seems to be in this patent (which otherwise covers very different material, but has a full explanation in the section covering Figure 2): (2014) https://patents.google.com/patent/US20150207496A1 *Latch circuit with dual-ended write*.

Why run your dependency based on instruction number rather than the Tomasulo physical register-based scheme?
One possible answer is that it allows you to have dependencies beyond just register-based.
We will discuss more details around load/store issues and, in particular, load-store dependency, below. But the issue that matters right now is easily understood:
suppose you have a load that depends on an earlier store (ie the load load's from an address to which a store stored in the recent past). Clearly, for correctness, the load must execute after the store has executed, otherwise the load will load stale data from the cache.
There are a variety of aspects to exactly how this is done, but summarizing it all
- there is a predictor table (the LSDP -- Load Store Dependency Predictor) that records these dependency pairs as they are encountered.
- interestingly, this table is associated not with the Load/Store unit but with the Mapper unit (the unit that figures out inter-instruction dependencies in a group, and the physical registers upon which an instruction [encoded as logical registers] depends. Why is this?

- because the LSDP implementation is handled as just one more dependency of the load! Just like the load might depend on instruction M to produce one of the registers that goes into the address calculation, it can also depend on instruction N (which will perform the store) as an instruction that has to be completed before the load can execute.
(I have to say I find this a genuinely cool idea, one that I have not seen before and did not think of for myself.)
Details of this (along with many other details of how the LSDP detects loads that depend on stores, and how those load/store pairs are aged out of the LSDP table) can be found in (2012) https://patents.google.com/patent/US20130298127A1 *Load-store dependency predictor content management*.

This requires being able to add an additional dependency to the Load – but that is not a problem given the dependency bitvector, it's just one more bit flip!
This idea can be (and is) generalized by Apple in *many* different directions.
- For example another aspect of Loads (to be explained later, for now we just give the idea) is Replay. A Load may fail temporarily because a resources it requires is unavailable (eg a TLB lookup or an L1D cache lookup). This can be resolved simply by having the load retry every few cycles, but that wastes energy and resources. For a range of Apple devices (from the A6 to the A10; as of probably the A11 an even more sophisticated scheme was implemented) Replay was handled by adding to the bitvector a synthetic dependency associated with the event being waited upon.
- Another version of these synthetic dependencies is used by a scheme that performs thread context switching in hardware and in the background as other instructions execute.
- Yet another version is used in Apple's value prediction scheme.

The dependency bitvector makes it easy
- to add new dependencies to any instruction.
- to define new types of dependencies. These merely have to grab a free SCH# from the pool (which we may want to expand to 450 or so) and broadcast that SCH# at the point when the synthetic dependency (eg service this particular TLB miss) has been completed.
None of this flexibility would be possible with a fixed sized dependency array of say a maximum of three dependencies per instruction.

implementing the scheduler

Finally how do we use all this?
Step one is we realize that the problem splits into three conceptually distinct tables.
Consider a Scheduling queue with 30 entries. Treat each entry as actually composed of three fields that are very different, so it's like we have three tables each thirty entries in size.
First table holds the operation, its physical destination registerID, and its source operands. These could be immediates, they could be physical registerIDs. Point is, this first table tells you everything you need to execute the instruction, but nothing about how to schedule it. It does not tell you when the instruction can/should run, just what you need once you've decide to run the instruction.

Second table holds the dependency information.
So this second table is going to be a huge bit matrix. 30 rows high, around 400 columns wide. We've already discussed the dependency bitvector. So for any given row (a pending instruction) there will be just a few bits set. Maybe bit 7 of row 4 is set , meaning
- the instruction in Scheduling Queue slot 4 is not yet ready to execute. It depends on some physical register that hasn't yet been filled, and the instruction that will fill that register is the instruction that has been given SCH# index 7.

Third table ties these two together. It holds the SCH# of each instruction, and the relative age data (discussed in the *Non-shifting reservation station* patent above).
So now think what happens conceptually during execution. The bypass bus still exists, still carrying all the generated results along with their destination registerIDs.
But there is a parallel bus that exists about 400 bits wide. Each time an instruction issues, from all the different execution units, each unit sets one of those bits active depending on the SCH# of the instruction that they are issuing. So that bus is carrying say 10 or so hot bits representing all the instructions that will soon be generating a result. (It doesn't matter whether they generate two results or one; all that matters is that a younger instruction depends on this instruction to have been completed before it can execute.)
That 400 wide bitvector can be grabbed by every scheduling queue, copied to the top of the scheduling bit matrix, and the hot bits allowed to "flow" down their columns. Every 1 that they encounter in the column gets flipped to a 0. (The 1 meant this row's instruction was waiting for this column's instruction to have been completed.) Then every row can be or'd to see if it's all zero; if so it's marked as ready and can be scheduled as soon as other details (like age relative to other instructions) allow.

There are plenty of details I'm omitting here.
One obvious detail is, as we have mentioned, there are additional dependencies that can be added to the bitvector, but those trivially fit into the model.
Trickier, and I ignored it to get the idea across, is what to do about instructions that take longer than one cycle to execute – I was sloppy about the timing details, but conceptually, of course, what you want is for each execution unit to place its "this instruction is about to complete" bit on the 400-wide common bus in the cycle just before the instruction will complete.

In one sense this is no different from the earlier tag and physical registerID schemes
- 14 units are each broadcasting that they have completed an instruction, and
- the identifier for that instruction has to be tested against every waiting instruction.
But we have dramatically regularized the problem! We have a single wide bus carrying the info from all those execution units. We have a single bit matrix (per Scheduling Queue) that is tested against that wide matrix. Our scheduling problem now boils down to figuring out nice circuits to perform the two tasks of interest (flip all 1s in a hot column to 0s; then find all rows with only 0s) and because these are clean bit-matrix problems we can solve them however is most convenient, even by analog techniques.

Note also that we simplify things by dealing with three different structures:
- a scheduling matrix that tells us which instructions are able to execute (all dependencies fulfilled)
- an age matrix which tells us which of the ready instructions are oldest
- an instruction list holding what needs to be passed on to the execution unit

This idea is called matrix scheduling. It may not sound like the patent if you read the patent blind, but I think you will agree that it's the only reasonable interpretation of how things have to work (given how wide the M1 is, and given everything else we know about the machine, like distributed scheduling queues and dependency bitvectors).

I *think* the specific point of the 2016 patent is the exact timing in this scheme. If you followed the tagID/destination physical registerID scheme, you'd broadcast the SCH# of each instruction in the cycle that it completes. This is easy, matches history, and the signal provided clearly means that the instruction has completed.

But it makes the Scheduler timing very tight; and it's not necessary! With the matrix scheme I have described, you know, from the moment the decision is made as to which instructions will issue, what the SCH#'s of interest will be, so you could broadcast them in the issue cycle.

That's fine for most (one-cycle-latency) instructions, but you need something extra to handle the multi-cycle latency cases, hence that patent. I think, for example, that the bit-shifting scheme used by load-dependent instructions to maintain them in the scheduling queue until the load is proved valid (or needs to Replay) could be reused as a timing mechanism for this purpose.

(But honestly this is one of the uglier Apple patents I have encountered – determined to claim everything, equally determined to reveal nothing. If Apple gets this rejected by a court at some point, they'll have only their lawyer to blame – part of the patent deal is that you explain how you do something, you don't just claim that something can be done. And this patent elides the explanation of the only part that's interesting, namely the multi-cycle timing.)

The one other thing the patent covers is a few technical details that reduce energy consumption by allowing rows that are no longer of interest (have all their scheduling requirements already matched) to opt out of the matching procedure.

criticality based scheduling (future, not yet implemented by anyone)

BTW instruction scheduling, though it's been thought about for so many years, is hardly a solved problem! In particular, scheduling based on the oldest runnable instruction is merely a heuristic it's not optimal. Better would be to schedule instructions based on *criticality*, which is a measure of how much subsequent instructions will be sped up by executing this particular instruction right now. There are way to measure (approximately, of course, we can't look into the future!) criticality fairly easily, and there are criticality predictors that do a good enough job. The concept is explained very nicely in Pierre Salverda (2008) https://www.ideals.illinois.edu/bitstream/handle/2142/11442/Principles%20of%20Instruction-Level%20Distributed%20Processing.pdf?sequence=2&isAllowed=y, *Principles of Instruction-Level Distributed Processing*.

But as far as I know no-one is yet using criticality to inform their instruction scheduling.

split scheduling queue (obsolete, only of historical interest)

For completeness, there's an additional early Apple Scheduler patent here (2008) https://patents.google.com/patent/US20100162262A1 *Split Scheduler*, but I think it's only of historical interest. The

problem that patent wants to solve is to be able to schedule dependent instructions back-to-back, and the solution adopted is to try to segregate “immediately dependent instructions” from “other” instructions, so that the immediately dependent instructions form a very small pool that can easily be scheduled.
However the problem to be solved is better solved via speculative scheduling, so this is all moot.

There are more modern interesting ideas in the space of splitting a scheduling queue; one of my favorites is (2015) https://hal.inria.fr/hal-0122519/document *Long Term Parking (LTP): Criticality-aware Resource Allocation in OOO Processors*. But so far no-one appears to have adopted any such ideas.

Experiments on instruction scheduling

Explaining the integer Dispatch Buffer

So back to the issue that got us interested in scheduling in the first place -- in the previous graph, what’s up with the pronounced flat region in the middle?
Here’s my best explanation:
Assume a two level Scheduling Pool: an initial buffer (the Dispatch Buffer) followed by a genuine Scheduling Queue. We know that Rename can dump instructions into the integer Dispatch Buffer at 8/cycle, but maybe instructions can only leave that buffer at 6/cycle, so one per scheduling queue? (Faster might be desirable under rare conditions, but it’s usually unnecessary, and it’s power/area expensive.)
This would mean that the Dispatch Buffer is going to fill up at a rate of 2/cycle (8 in, 6 out). Or, more precisely, perhaps each of the two Dispatch Buffers fills at 1/cycle (4 in, 3 out)?
After 96 ADDs, a Dispatch Buffer that can hold 24 (2/cycle*12 cycles) will be saturated. In fact Dougall’s experiments (probing for exactly this size, not detecting it as a side effect like we are doing) has the 1st level int Dispatch Buffer as 24 elements (split into 12+12).
So effectively the machine runs like
First 96 ADD’s: throughput 8-wide, fills up first level Dispatch buffer with ~24 instructions
274 ADDs from 80 to 370: throughput 6-wide, since only 6 instructions can pass from Rename to Dispatch
excess cycles in this phase is (290/6-290/8=11.4. We have permanently lost ~12 cycles in our 8- vs 6-wide comparison.
MOV’s start: for about 4 cycles (24 ADDs/6) we get to process ~32 MOVs (or NOPs) for free in Rename, in parallel with int Execution as the excess ADDs stored in the 24-slot Dispatch Buffer are drained.

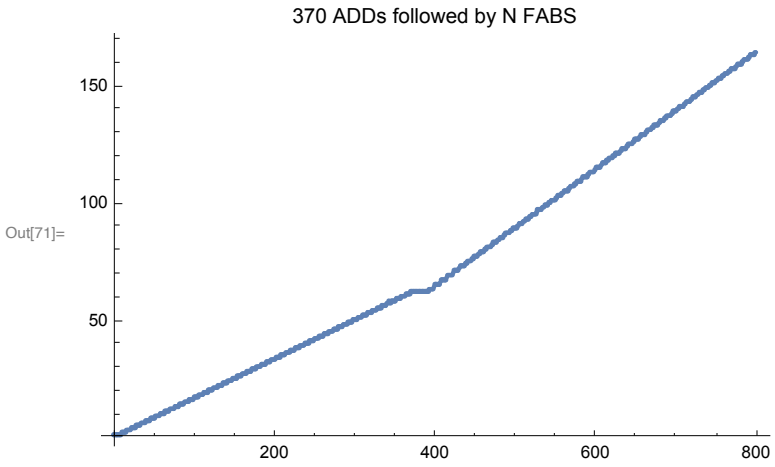
In other words the flat region (when it’s apparently costing us no extra cycles to process MOVs (or NOPs, or anything else executed in Rename) are when we’re overlapping execution in Rename with instructions that were present in the Dispatch Buffer and are now making their way to Execution. For

this to happen we need the flow rate into Dispatch Buffer to be higher than the flow rate out from Dispatch Buffer, so that a backlog of instructions builds up in Dispatch Buffer that can then drain in parallel with the subsequent instructions executing in Rename.

testing the fp Dispatch Buffer

We can check this understanding by testing a few variants.
If this analysis is correct, we should see the same phenomenon if we follow the ADDs with something like FABS which executes not in Rename, but in a different pipeline, so can again run in parallel with the int Dispatch Buffer draining.

Out[71]=

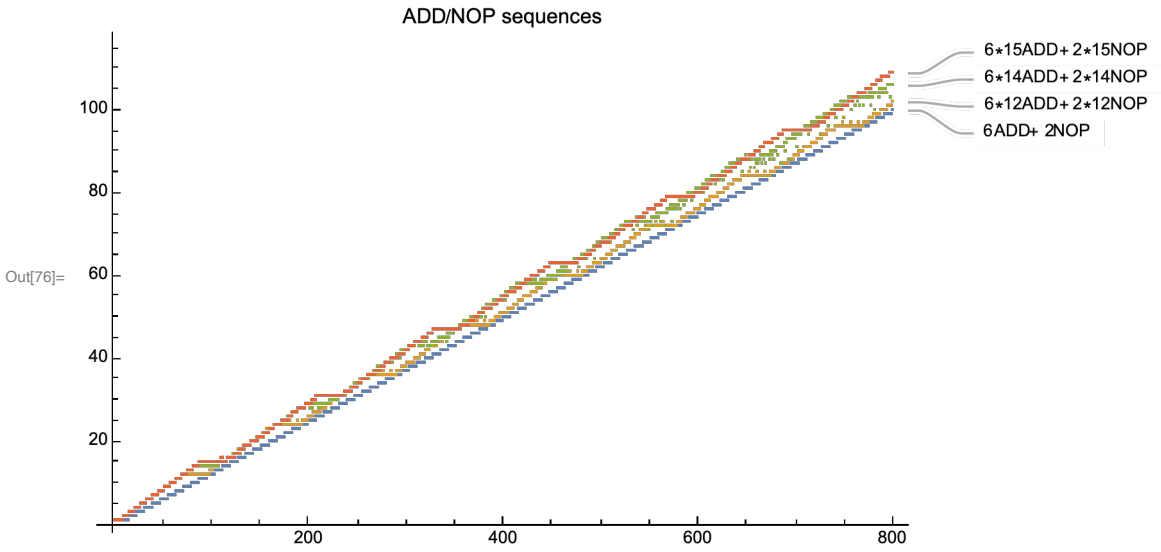


Well, that's nice! As we predicted :-) And in fact the signal is cleaner (the flat run is 23 instructions long)

Interpretation as smoothing mechanism

Now let's try something a little different.
Suppose we interleave (6ADD+ 2NOP). We know that can run at 8 wide indefinitely, it's perfectly balanced with, every cycle, 6 instructions going into the integer pipeline and 2 NOP instruction being executed at Rename.
Now suppose we change this to (12ADD+ 4NOP). Now it's not *perfectly* balanced: The first cycle we dump 8 ADDs into the Dispatch Buffer, but only withdraw 6. The next cycle 4 ADDs into the Dispatch Buffer -- along with 4 NOPs executed at Rename, and then 8 NOPs executed at Rename. But still balanced overall – as long as we have a buffer that can hold 2 ADDs.
We can continue like this through (k*6 ADDs + k*2 NOPs), until the point where the Dispatch Buffer is not longer quite large enough to hold all the excess ADDs. Let's see what I mean:

Out[76]=



Consider the case of 6*15 successive ADDs. This is 90 ADDs, so takes just over 11 cycles (8*11=88) in the pipeline up to Dispatch. Each of these 11 cycles, 2 excess ADDs remain in the Dispatch Buffer, so at the 15*6 case we have almost maxed out that buffer. Throw in imperfect alignment and imperfect balancing of instruction and we can say the buffer reaches its capacity. And we see that it is indeed noticeably slower than the predecessor cases.

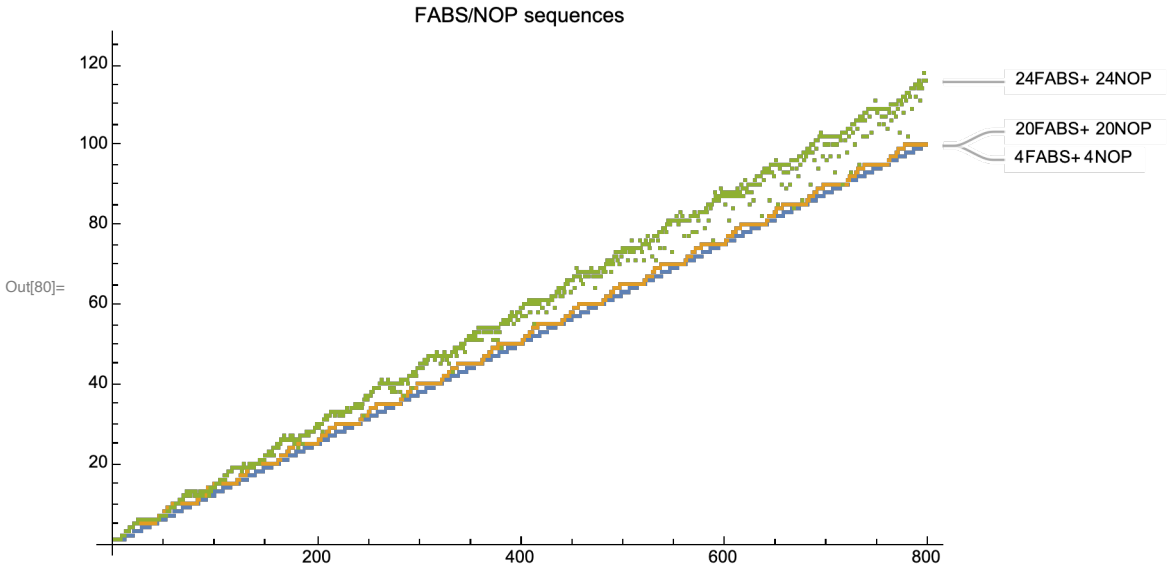
Essentially the case 6*1 through 6*13 always take the fast path (102 cycles per loop iteration), anything above 6*15 takes 108 cycles, and 14 is a slightly noisy version in between.
To put it even more simply, the Dispatch Buffer acts to smooth local variations in the types of code, so that a dense run of ADDs can still run 8 wide, as long as the dense run is not *too* long. We see the transition here between the case of *not too long* and *too long* here at between 6*13 and 6*15 successive ADDs. Once you are unbalanced for too long buffering cannot save you and your maximum throughput is reduced (ie slope of the line moves upward).

The same sort of buffering over regions of dense code (for example a stretch of FP or load/store dense code should likewise help us to still maintain an IPC of 8, as long as
- overall the balance of instructions is not too biased to only one type of instruction, and
- the runs of a single type of instruction are not too long.

acceptance width of fp buffer

Now the next obvious question is what are the exact properties of these Dispatch Buffers. For example consider the floating point dispatch buffer. Presumably this releases instructions 4/cycle, but does it accept instructions 8-wide or something narrower? Let's see.

Out[80]=

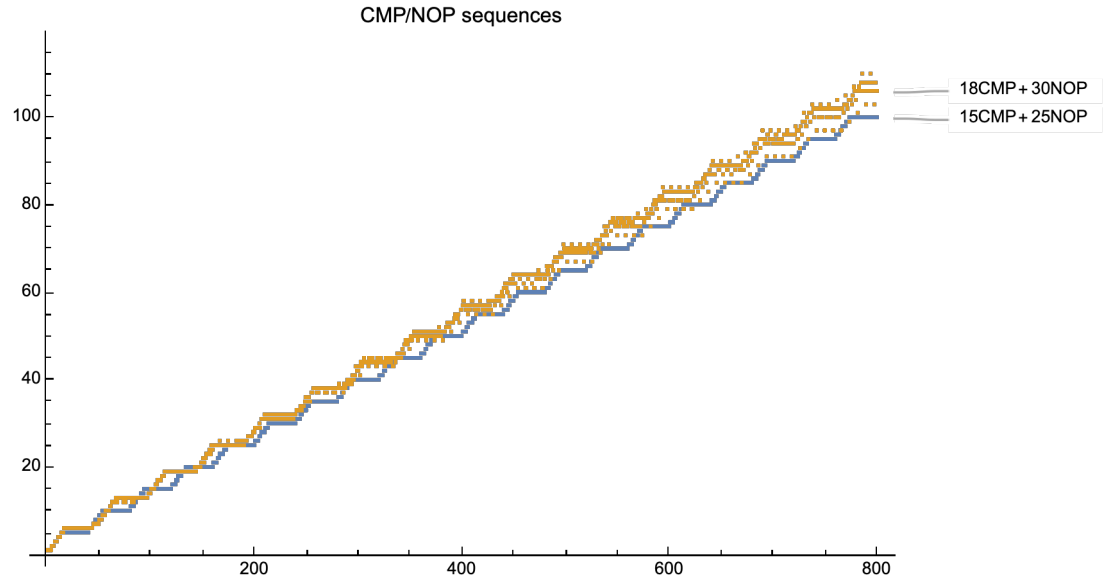


So let's assume that the green line were exactly matching the lower two lines. That would mean that three successive (8x `FABS`) instruction probes could be processed through Rename (and into fp Dispatch), followed by three successive (8x `NOP`) instructions without a hiccup. Which would mean that the Dispatch Buffer must be able to accept 8 instructions per cycle, and be able to hold at least 12 instructions (since each of those three cycles, its net occupancy will grow by 4 instructions.
We see that we can't quite achieve that with 6*(4x `FABS`+4x `NOP`) but we can with 5*(4x `FABS`+4x `NOP`). Suggesting that the Dispatch Buffer is ~12 in size. (The methodology is somewhat sloppy because it will be sensitive to exactly how the FP instructions are aligned in the run of eight instructions that's passing through Decode to Rename – sometimes there will be perfect alignment, sometimes there will be a straddling of some FP instructions at the head and at the tail of a line.)

probing that the integer buffer is split in two

Now what if we try this same methodology with CMP? We believe that CMP can execute of three of the six integer pipelines, and Dougall's diagram suggests that one of the two integer Dispatch Buffers feeds these three pipelines, meaning we should expect to only be able to buffer 12 of these instructions. Do we see that?

Out[83]=



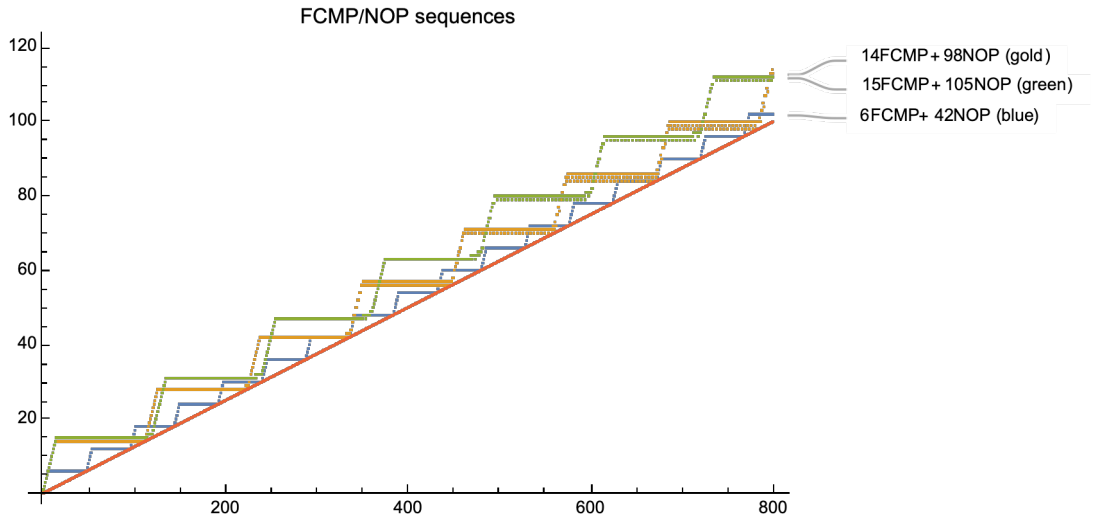
So let's think about this. When we run 6* (3x CMP+5x NOP) we have a sequence of 18 successive CMPs running through Rename. That's a little over 2 (round it up to three!) cycles.
If we take this as meaning that we can transfer all those CMPs into a (12-sized) buffer while, during two cycles, extracting 2*3=6 CMPs, the numbers kinda work out – certainly better than assuming a 24-sized buffer. It also seems that these two integer Dispatch Buffers can accept 8 instructions per cycle.

You will recall when we started down this path we also ran the experiment for MUL, another instruction specialized to only some integer execution units, and again we saw the same sort behavior, a buffer of ~12 in size.

probing if the fp buffer is split in two

What about the fp Dispatch Buffer -- is it possible that there are two of these, each feeding two of the four fp pipelines?
We can try the same technique with FCMP (which can only execute down one of the four fp pipelines). The results are

Out[87]=



Now this one is actually surprising! The jump (deviation from a long term rate of 8 operations/cycle) occurs at 15* (1x `FCMP`+ 7x `NOP`). It's hard to square that with the precise numbers on Dougall's graphic.

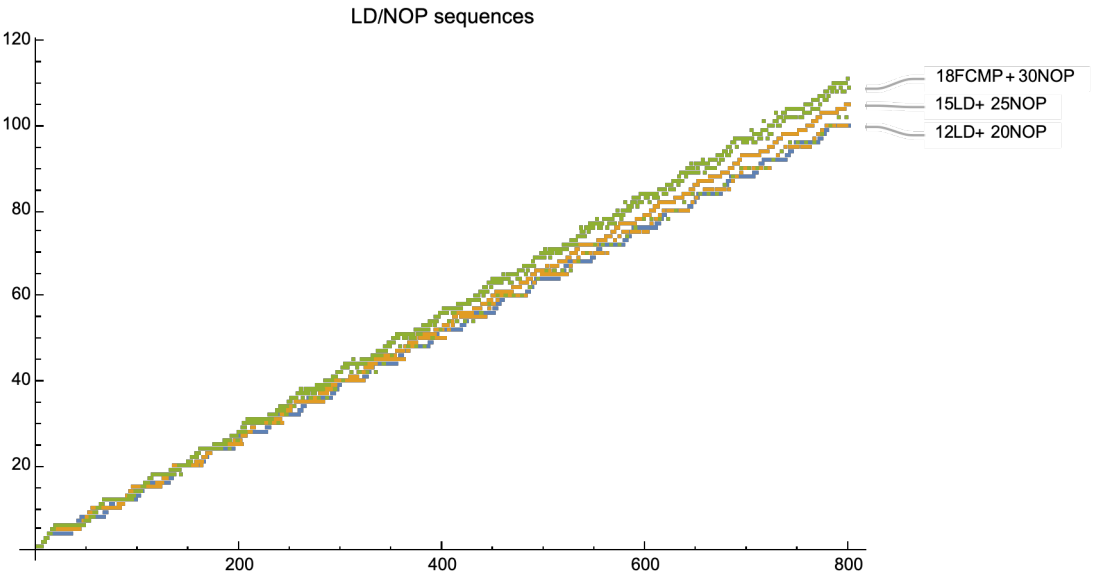
If we can sustain 14 `FCMP` without reducing the overall throughput (ie overall the gold curve stays at the same slope as the blue curve [6 FCMP] and the red curve [8 ops/cycle perfection]), that implies we must be able to in a first cycles move 8 `FCMP` from Rename to FP Dispatch, of which one moves on to Execute. Then in the next cycle we move over 6 FCMPs, one goes to Execute, leaving us with a net of 12 in Dispatch. This is so perfectly matched it suggests that perhaps the size of the FP Dispatch Buffer is slightly larger, perhaps 14 rather than 12?

Either way, the point we wanted to establish is established – we have a single FP Dispatch Buffer, rather than a split buffer like integer.

probing the ld/st Dispatch Buffer

Since we have the machinery in place, we might as well also test load/store!

Out[91]=



If we compare this to the CMP results (`CMP`, like `LD`, is 3-wide) we can see that we overflow the LD/ST Dispatch Buffer slightly sooner than for `CMP`, which agrees with Dougall's result of a size of 10 for the LD/ST Dispatch Buffer.

Handling Immediates (MOV #)

Overview

So far we've seen that we have what feels like ~380 int physical registers, but ~624 entries in the History File. If we're doing integer-only work, we'd like to be able to make use of all those HF entries without first being limited by the physical registers. How can we do that? Well

• some of the HF entries will probably be used by instructions that set flags.

• another aspect of bridging the gap is register duplication (zero-cycle MOV), which requires an HF slot, but not an additional physical register.

• a final aspect, which we'll examine now, is the special treatment of immediates in the context of register initialization.

There are a few different types of immediate, and, unfortunately for us, they all seem to have somewhat different paths, and different characteristics!

zeroing

Most common is the need to zero a register, and one can imagine multiple ways to do that, eg

`MOVZ x0, #0` (aliases to MOV x0, #0) runs 8 wide

`MOVI x0, #0` (normally aliases to ORR, not allowed for #0 so we will try `AND x0, xzr, #1`) runs 6 wide

`MOVN x0, #0` (not allowed for #0 or #-1)

`MOV x0, xzr` runs 6 wide
`EOR x0, x0, x0` runs 1 wide! Not only not an idiom, but forces the chain dependency!

So Apple clearly want us to use `MOV #` (whatever the preferred machine code might be), and aren't going to make an attempt to catch other cases. (It is remarkable that `MOV x0, xzr` is not special-cased as a Rename-time remapping...)

non-zero immediates

How about constructing non-zero constants?

```
MOV x0, #1
MOV x0, #-1
MOV x0, #0x1234
MOV x0, #0x12340000              (coded as MOVZ x0, #1234, lsl 16)
MOV x0, #0x1234000000000000      (coded as MOVZ x0, #1234, lsl 48)
MOV x0, #0808080808080808        (one of the "boolean" coded logical immediates)
```

ALL of these (!) are recognized and run 8-wide.

Now let's not get carried away with the 8-wide goodness! Apart from an immediate of zero, these are implemented as 2-wide in rename plus 6-wide in execution, so not quite as free as register duplication, but often free.
The common case of an isolated initialization or two will be free; along with 6 other integer ops, and even the common case of many back-to-back initializations will run 8-wide, albeit often using integer slots.

dependencies of immediates

How about dependencies?
`MOV x0, #0; MOV x1, x0` seems to run fine (800 ops in 100 cycles), though I'm not doing true latency tests.
In other words both these zero cycle operations either stack together in the same Rename cycle, or appropriately pipeline in successive Rename cycles as you would hope.

Likewise for `MOV x0, #0; MOV x1, x0; MOV x2, x0; ...MOV x7, x0`.
M1 even appears to be able to note and appropriately handle (via zero-cycle rename) all the dependencies inherent in the chain
`MOV x2, x1; MOV x3, x2; MOV x4, x3; ..MOV x0, x8 !`
Apple appears to feel (I assume this is what LLVM will generate) that if you want to set many registers to the same value, the way to do that is via
`MOV x0, #; MOV x1, x0; MOV x2, x0; MOV x3, x0; ...`

Out of interest, we can also test
`MOV x0, #1 + MOVK x0, #0x1234, lsl 16`

to create a 32-bit wide constant.
This runs at 4 (of these pairs) per cycle, suggesting that 2 of the ops (probably always the baseline MOV#s) run at rename, while the other six (two of the baseline MOV#s, four of the "with keep" bit-insertion `MOVK`'s) run at execute. This matches Dougall's throughput for `MOVK`. And also tells us that `MOV+MOVK` is not fused.

floating point immediates (no interesting support)

ARM supports a few floating point immediates, but the M1 does not handle these in any special way at Rename.
Likewise the way to zero an FP register is via `FMOV dn, xzr`, but (like everything xzr related...) this is not special-cased.
So there are various dimensions along with the current scheme could do better, especially for FP.

implementation

How is this immediate-handling implemented?

2012 scheme (special register names)

The patent 2012 https://patents.google.com/patent/US9430243B2/ *Optimizing register initialization operations* discusses the original implementation, suggesting the goal of zero-cycle initialization is to perform initialization one cycle earlier, not either to offload work from the integer pipelines, or to give us some free registers; those are nice side effects but not the primary goal.
The implementation has changed since 2012, but the *primary* purpose (lower latency initialization) appears to be unchanged.

The 2012 patent suggests that unused physical registerIDs can be used to encode a few small constants (think eg 0, +1, -1). The idea is that if you have, say, 192 physical registers, you can use registerIDs 192..255 to encode special immediates. Beyond that the details are vague (apart from a suggestion that registerID 255 is hardwired to 0).
However it is important to note that (like many good ideas from the 2012 A7 era of Apple cores) this idea seems now to have been superseded by a new scheme, that's not yet in the patent literature.

I can imagine an extension of the 2012 idea that uses wide'ish registerIDs, about 22 bits or so. The first bit or two would indicate an immediate (of four or so types) vs a physical register. If a physical register, subsequent high bits could indicate the register file (int vs fp vs flags), and even special purpose registers and (indirect, ie delayed allocation) ROB IDs.
This would allow immediates to be "executed" 8-wide at Rename time, and would allow fp immediates also to be executed at Rename time. When a subsequent use case wished to read the register, it would submit the registerID, as usual, to the register file, which, rather than performing a lookup, would have

logic to decode the tag and generate the appropriate immediate onto the bus delivering the register value. (That part is described in the 2012 patent.)

This is not as outrageous as it might appear. Something like this already has to be done in part: One of the "registerID" slots of instructions transported down the integer pipe for int execution today has to be wide enough to hold all the various forms of immediate as part of the instruction definition. So what I am suggesting is not that absurd a modification, it's more simply allowing that same wider registerID across *all* the registerID slots.
But that's not what we have today... Oh well, there will be more chips! And Apple has changed the scheme at least once, they can improve it some more!

So that's one path to handling immediates, via encoding them in a wider "register dependency" field.

current scheme (separate register pool)

Another path (which appears to be what Apple is doing right now) is using a separate register pool. Imagine that at Decode you interpret that an instruction is a `MOVI xn, #123.` What can you do with this information? One possibility is to find a free physical register and store the immediate value in that free register; then the next cycle you can map that physical register to xn.
This sounds good in theory, but runs into a wall as soon as you think about implementation details. The basic problem is that the primary integer register file has a certain maximum number of write ports and the storing of the immediate in that register file adds to the pressure on those write ports. One could just accept that cost, that you can't always get what you want including as many writes to the register file this cycle as you want; but another alternative is to create a separate "immediates-only" register file which can only be written to by this Decode path. We'll see below that when using immediates we appear to have access to about 38 more integer registers than usual! Which strongly suggests that there is indeed such a separate pool.

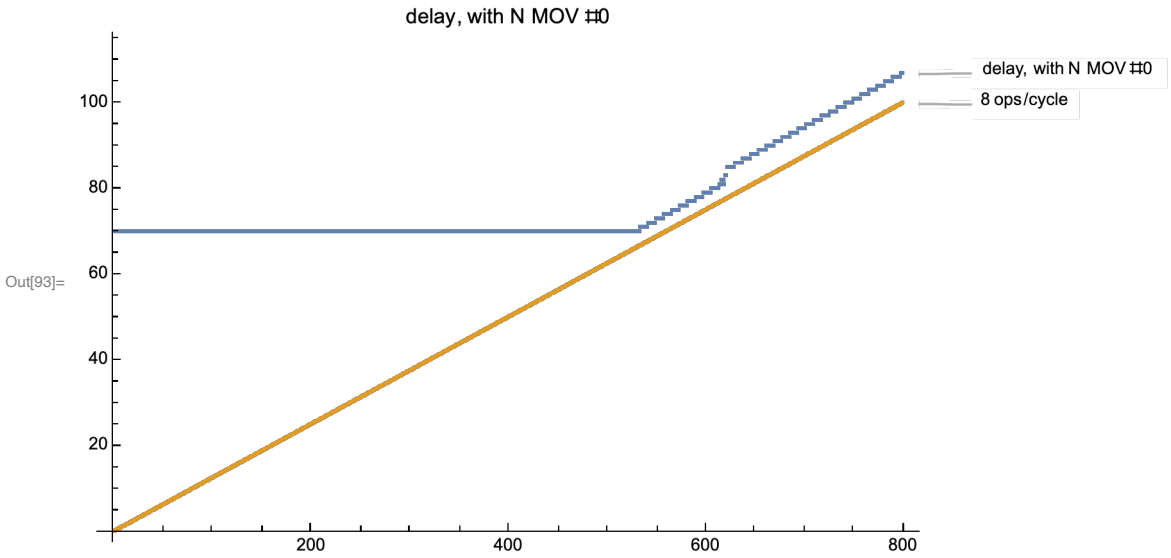
I can't find any academic discussion of how common immediate constants are in code, and so whether they are common enough to justify all this effort.
However eventually we will discuss Value Prediction, which also strongly benefits from having a separate register pool (again for reasons of write port pressure), and it's possible that this pool of additional registers also services the (currently very limited) value prediction in the M1.
xxx need to probe/validate this

Probing immediates experimentally

mov #0

Let's test a delay block along with N `MOV #0`'s.

Out[93]=

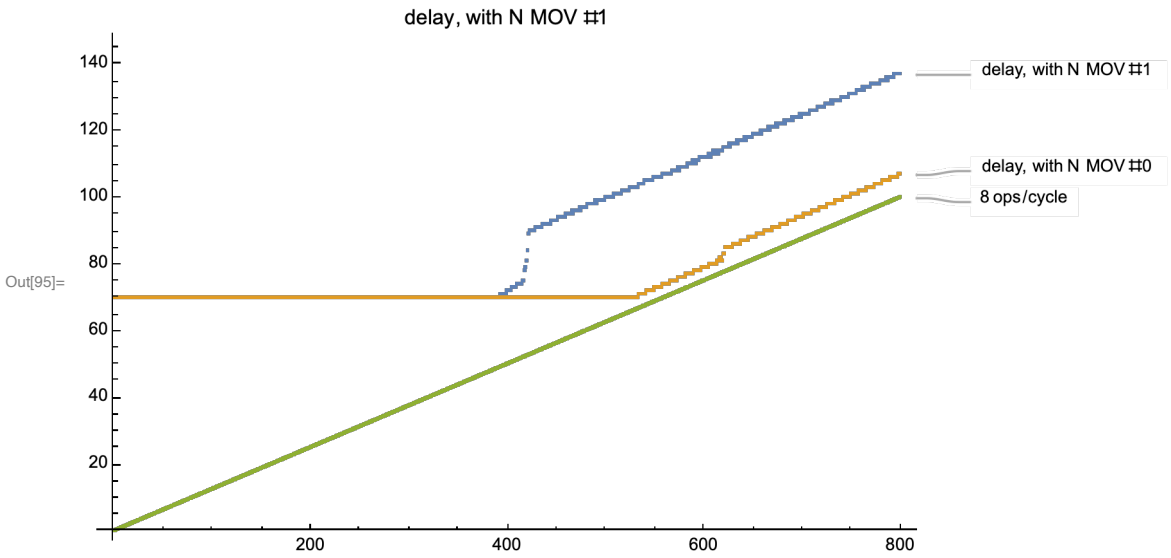


No real surprises. The slope is 8/cycle. There's a brief stutter around N=620, presumably something to do with the History File, but let's not get distracted.

mov #1

Compare this to the exact same structure but using `MOV #1`.

Out[95]=



Rather different!

It's appears that `MOV xn, #0` is special-cased and is handled like `MOV` or `NOP`, fully resolved at Rename.
However any other immediate, eg `MOV xn, #1` is "semi"-special-cased.

existence of additional, special, immediate registers

We already know that we get two free MOV# Rename's per cycle; any more than that have to route through Execute.

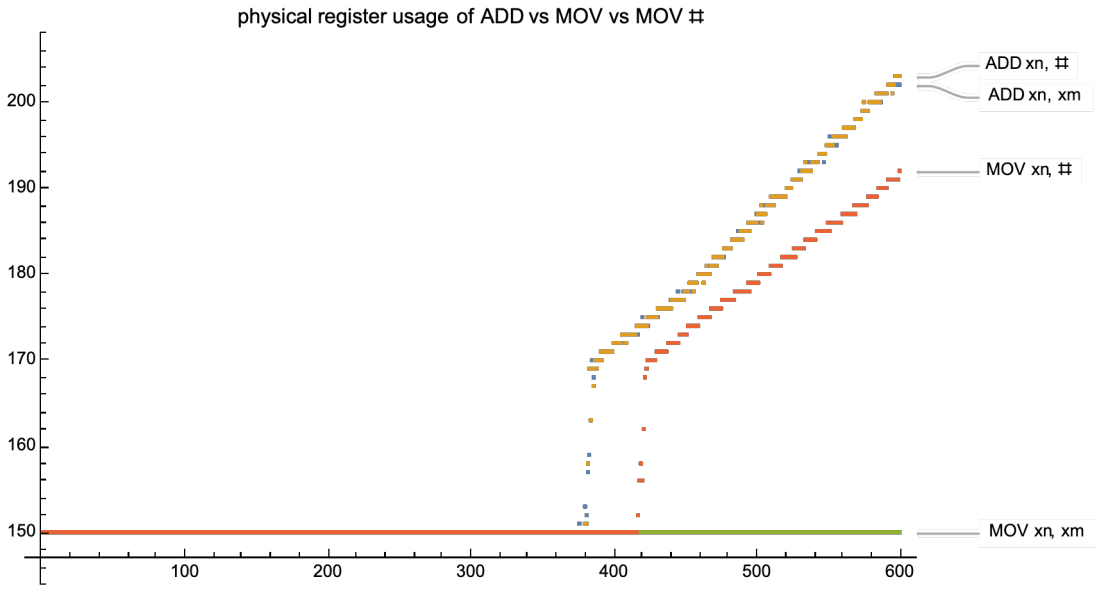
But compare the the MOV #s to standard register usage, in terms of the use of physical registers.

(For reasons that will in time become apparent, we'll also switch to a much longer delay time.)

So see what I mean, compare

```
ADD x0, x5, x5
ADD x0, x5, #1,
MOV x0, x5
MOV x0, #1
```

Out[100]=



So

• the ADDs (two registers, or register and immediate) max out at around 380 physical registers.

• the MOV x0, x5 does not use physical registers at all, and maxes out at around 620 History File entries.

• MOV x0, #0 is like MOV x0, x5; fully executed at Rename, doesn't use a physical register.

(No surprise since it's doubtless a rename to a hardwired zero – but not xzr.)

• The unexpected case is MOV x0, #1, which maxes out at around 418 physical registers.

It's like it has access to about 38 more registers than a standard integer instruction!

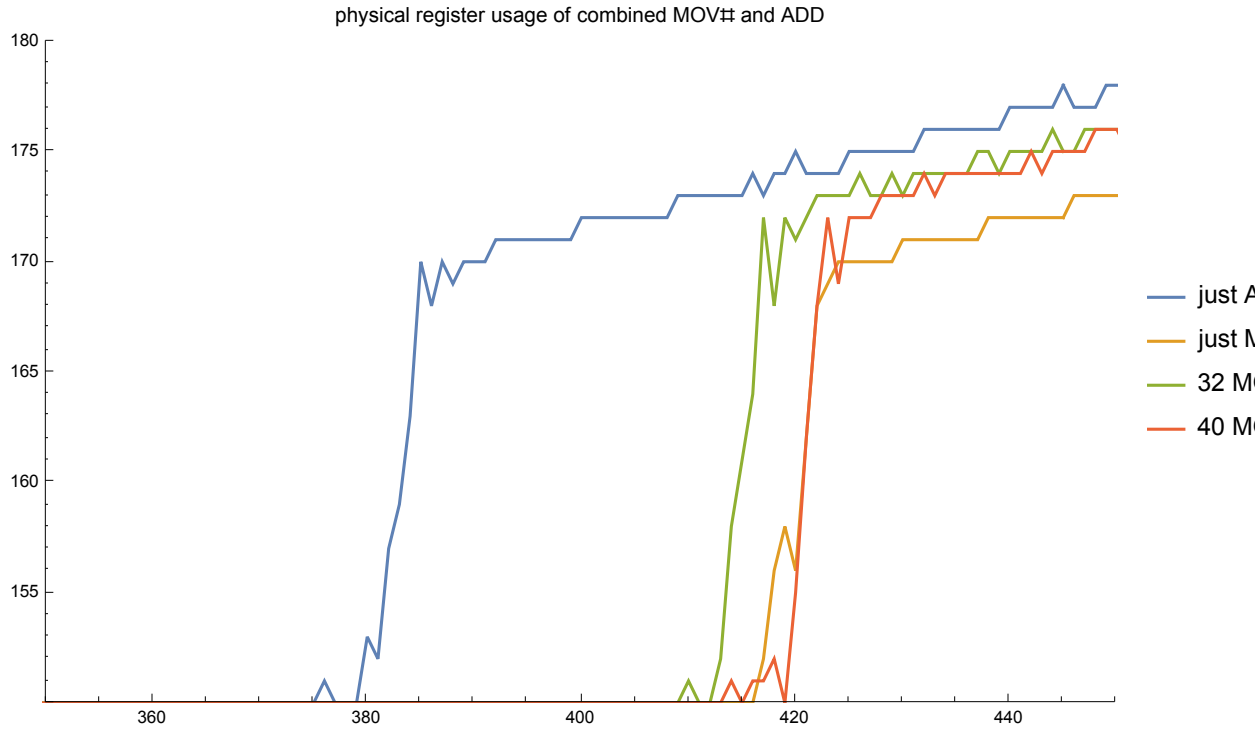
Let's examine this more carefully.

Our hypothesis is that there's *some* extra storage available for holding immediates; ie a few special registers (call them I registers or iRegs) that can only be written to at Rename, and that aren't part of the main int physical register set.

But we need to be careful in testing this because, remember that we can only execute two MOV#'s in Rename per cycle. So we need to pad those MOV#'s with NOPs to ensure that nothing goes down the standard integer pipeline (where it presumably fills a standard integer register).

We start by considering a probe that consists of 32x (`MOV x0, #1; NOP NOP NOP`) followed by ADDs. The idea is to see whether the `MOV x0, #1` took storage away from the ADDs.

Out[103]=



(Normally I haven't added "joining" lines between actual data points, but in this case the lines are helpful to guide the eye, even though they exaggerate the noise in the image.)

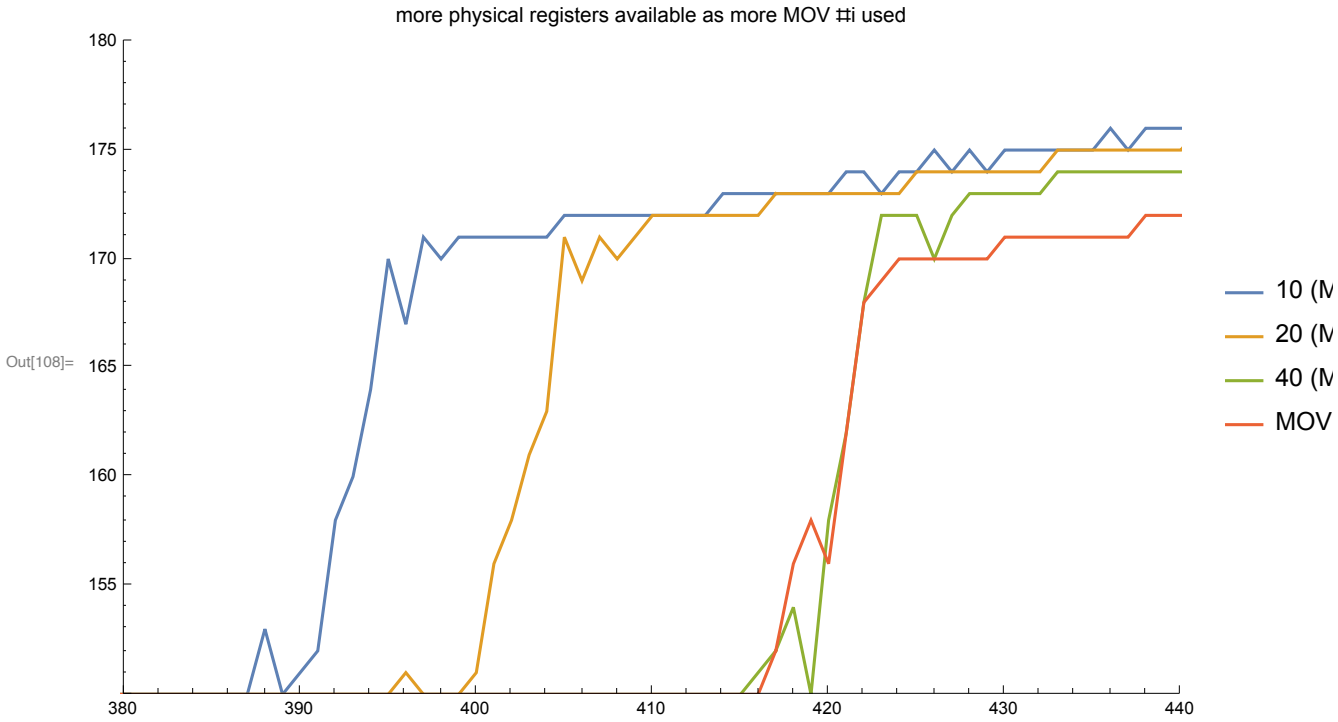
Hypothesis confirmed!
Look at how the green curve jumps at close to the same place as the gold curve; ie the MOV# register usage has not removed registers from those available to the ADD.
The red curve (40 rather than 32 `MOV #1`'s) suggests that there are more than 32 I registers available, perhaps as many as 40.

how many immediate registers? how many values can they hold?

So the next question is how many of these registers are there really, and how they are "organized".
For example one possibility is that there is only a single immediate register, which can hold a single non-zero immediate value (but with multiple references).
Another possibility is a set of 40 registers, each of which can hold a different immediate.
Or anything in between, like a CAM that can hold up to 8 distinct values?

First let's try 40 `MOV x0, #i` before the ADDs. (ie the immediate value ramps from 1 through 2, 3, ... 40)

Out[108]=



So we see that if we use 10 `MOV#i's`, we get essentially 10 additional registers (we max out at 390 rather than 380). Same for 20.
For 40 we don't get quite all the way to 420, so the number of additional iRegs appears to be around 36.
Also we are using a different immediate value for every `MOV`, so we must have at least 36 distinct "storage objects" available.

what happens when we use up all the immediate registers?

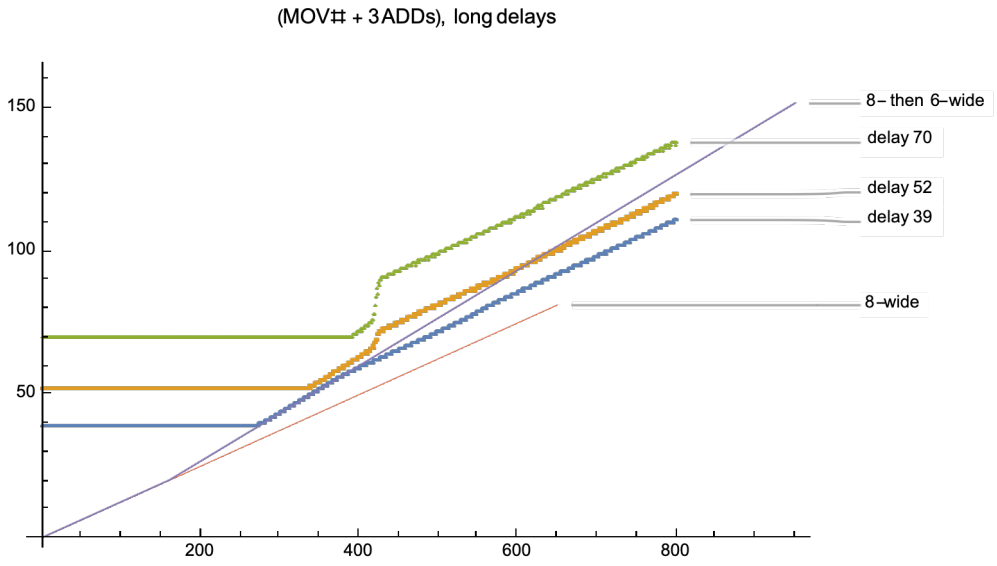
One final test . We know that the I-registers can be written to from Rename.
Consider the following probe N times (`MOV x0, #i  ADD x0,x5,x5  ADD x0,x5,x5  ADD x0,x5,x5`).
This produces a curve that jumps at around 416 or so, no surprise, and that has a slope of 8 instructions/cycle, again no surprise.
But think what the jump at 416 means.

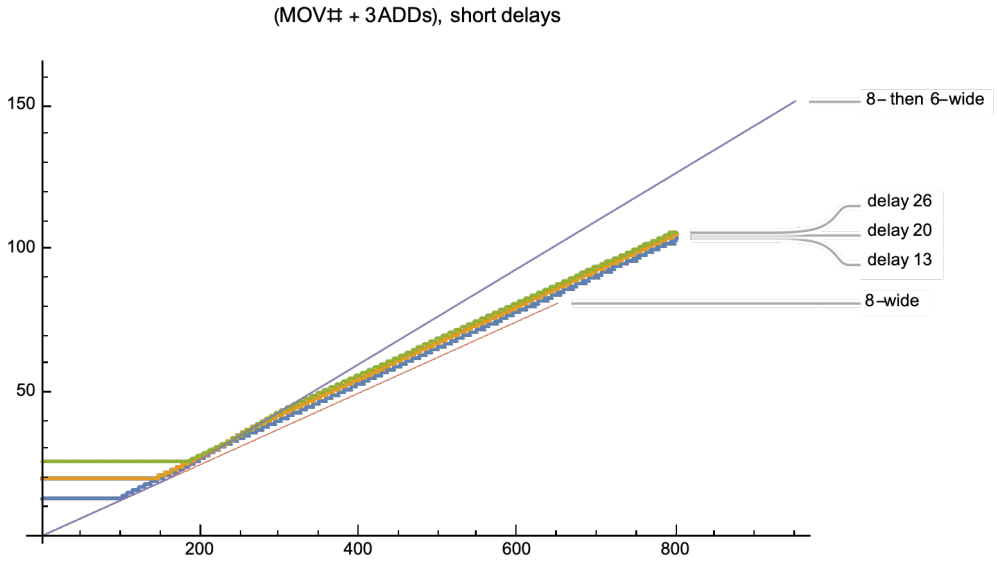
Presumably there are no more than about 36..40 iRegs. And 416=52*(2+6).
So we know that the first 40 or so MOV#'s are handled at Rename, apparently via being written to special iRegs. What about the (2*52-40=64 `MOV #i`'s that execute after those first 40, once the 40 iRegs have been allocated?)
One can imagine two possibilities.
• The first is that there are generic write paths from Rename to all registers in the int register file.
If this is the case, then we should be able to process the 416 or so operations in 52 cycles.
• Alternatively, if the write paths from Rename to the int register file are only present for the iRegs, then

we'd expect the 416 operations to be processed using 20 cycles for the first 160 operations, and then (416-160)/6=43 cycles for the remainder, taking a total of 63 cycles.

Can we see this difference?
It turns out we don't actually need to work so hard!

Out[117]=





To make things simpler, I split this into two plots, one showing small delays, one showing longer delays.

The longer delay plot (look at eg delay 39) answers the question that interested us, namely we see a stretch of 6-wide execution before reverting to 8-wide.
In other words, it appears that only iRegs can be written to from Rename, and when an iReg is not available, MOV# has to run 6-wide.

But beyond this, one might ask about the other details in the plots.

• What I think is happening is that for short delay, the pattern for every loop iteration is first to use up the iRegs, which are then rapidly released when the delay completes; thus there is a very limited to no window during which no iRegs are available.

• On the other hand assume a longer delay, say 39 cycles. One's first quick thought is that the iRegs should always be used at the start of a loop, then released as soon as head of ROB clears, so should be available again and we should immediately revert to 8-wide. That is what happens on the first iteration through, but it's not what happens long term.

Long term I think what happens if, say, you're running with N=300 and delay of 39 is that iRegs are used when they can be, released at some later time, and land up being spread throughout the ROB rather than being all bunched at the beginning of the ROB.
And so even when head of ROB clears, that doesn't free all the iRegs right away. In fact it's only if you have N>~420 or so that you provide a long enough period of time after head of ROB clears for *all* the registers to be freed and the a reversion to 8-wide.
Later we will see that freeing registers (when it is possible, ie if there's not a delay instruction blocking the head of the ROB) occurs at 16 registers/cycle, which is fast but not infinitely fast.

summary of experimental findings

So I think the model is that
- there is a special hardwired zero physical register (ZPR) (which, strangely, appears not to be xzr!) and a MOV #0 is treated as a Rename duplication to that ZPR
- there is a augmentation of the physical register file that can hold perhaps 36 distinct values, and can be treated as part of the register file (can be looked up via RegisterID)
- there is a special write path from Rename to the iRegs, but not generic pRegs, to allow up to two writes from Rename per cycle
- but these two Rename writes are only possible if free iRegs are available

- thus if Rename sees that an instruction involves the write of a known value AND a free iRegister is available, then Rename will simply write the value and mark the MOV# as complete, so that it doesn't need to pass on to Scheduling and Execute.
But if any of the conditions fail (not a known value, no free iReg, already both ports to the iRegs are in use) the MOV # is not marked as complete, and it passes on to scheduling for normal execution.

- `MOV #` is primarily a latency reduction tool, but it does also give us the effect of a few additional physical registers.

Some Register File Implementation Details

You may recall that when we first started looking at exhausting the physical register file we saw some unexpected blips. Now is the time to consider what they tell us.
Suppose you have to create a register allocator – what are you being asked to do? You need machinery that performs the following tasks
• you need a pool of registers
• you need a way to know which registers are in-use versus which are free
• given the pool of free registers, you need to provide Rename with up to 8 physical registerIDs every cycle.
Each of these tasks has many further details!

Structure of the register pool

Consider the pool of registers. We talk about a "register file" but there are many possible implementation details here.

For example: in Apple's first 64-bit cores, the integer file was in fact split about 40% 32-bit registers and 60% 64-bit registers! This allowed saving some area, and the free register allocator would preferentially provide 32-bit wide registers for instructions taking a w- register rather than an x-register.
Details here: 2013 https://patents.google.com/patent/US9639369B2 *Split register file for operands of different sizes.*

Apple no longer does that (at least I can find no evidence for it), but what they do do is split the register file into four banks which can each be individually powered on or off. To put this in context however, we first need to look at how registers are marked free.

Apple’s implementation of a “free register list”

Obviously one part of freeing registers is having the ROB tell you when a physical register is no longer in use; we'll assume that's a solved problem and ignore it. More interesting is how do you preserve this information of what registers are free?
One option is to maintain a queue- or list-like structure (so basically an array of slots, newly freed registers go in one end, registers to be allocated come out the other end). This works and is easy to run wide (just pull 4 or 8 or however many you need values from one end), and has been the standard solution for years, hence the usual term free register list. But this solution limits your control over the process.

Apple have (once again) gone through at least three refinements.

bitvector (2012)

The first method, (2012) https://patents.google.com/patent/US20140013085A1 *Low power and high performance physical register free list implementation for microprocessors* is based on bitvectors. Instead of a queue, imagine that when a physical register is free we flip a bit associated with the register, so that a bitvector consisting of *NumPhysicalRegisters* bits tells us which registers can be reused. This uses little storage but does require, every cycle, a search along the vector for bits that are set to 1.

The patent describes multiple ways to make this scheme work better. The important ideas are
• we actually use multiple bitvectors. Remember the older A7 design was 6-wide, so the maximum number of register allocation per cycle is 6. We have three bitvectors which, between them, cover the entire set of free registers.
So 1/3rd of the free bits live in the first vector, 1/3 in the second, 1/3 in the last.
• we have three pairs of scanners that run over these three bitvectors, one scanning forwards, one scanning backwards; so each scanner has to find one free bit, and between the 3*2 of them you can find 6 free registers.

• but for this to work well, the free registers have to be evenly distributed over all three bitvectors, and most of the patent is about how that is done.
• some ideas for how to do this are obvious (like, to the maximum extent you can, if you only need to deliver say 4 registers, deliver them from the two bitvectors that are most full, and don't scan the one that is emptiest).
• but one particular part of the solution is of interest going forward, namely that Apple draws a distinction between two classes of free registers.

There are, what you might call, “long-term free” registers which have been marked via the bitvector as free.
There are also what you might call “short-term free” registers (Apple calls these “returning physical registers”) which the ROB has just told the Register Allocator, are free. These are treated differently with the register allocator (as far as feasible) trying to *immediately* recycle the “short-term free” registers so that, ideally, they do not have to have their setting in the bitvector toggled on, then immediately off again, and to avoid the more power-expensive operation of scanning the bitvectors.

multiple bitvector banks (2016)

The second stage of evolution we see in (2016) https://patents.google.com/patent/US10372500B1 *Register allocation system*. This uses a system that’s clearly built on the ideas of 2012, but with additional techniques for saving power.

The idea is that, as I mentioned, we split our register file into multiple banks (the patent suggests 4, and

I suspect that's still the case; it matches the results I have seen). The most important goal of this banking is to save power by trying to limit (as much as possible) activity to just some of the banks. You only need many physical registers under a few circumstances; much of the time you can do with far fewer. So we want to arrange things to, as far as possible, stick to using registers in as few active banks as possible, and only activate a sleeping bank when really necessary. The patent is about how we do that.

The details are very low-level, but the overall idea is: as far as possible, ensure that new register allocations (while building upon a scheme much like the earlier 2012 bitvector scheme) are bunched towards a single register bank. Sounds good, but how to do this?
• One part is fairly obvious, namely the use of a "short-term free" register queue, presumably on the theory that such registers come from register banks that are already powered up. (The usual situation, by far, is that the ROB is not especially occupied, and registers are rapidly used then freed, they normally don't hang around in the ROB for hundreds of cycles!)
• The second part is much more ingenious! It still uses the multiple bitvectors of the 2012 patent, but rearranges the mapping of these bitvectors into the register banks so that all the (now eight rather than six) scanners are likely to find free registers in the same single bank. You need to look at the patent to see what's done, but it is very elegant.

elide register allocation where possible (2017, not yet implemented)

The third stage of evolution would be some sort of resource amplification, one or more of
• late register allocation (virtual registers)
• early register release (as soon as there are no users, no need to wait till Retire)
• no register allocation (read the register value directly off the bypass bus). This can be done when architectural register values are immediately overwritten.

We know that the third option is in limited use for some cracked instructions, and is covered by the patent 2017 https://patents.google.com/patent/US10691457B1 *Register allocation using physical register file bypass,* which is discussed in more detail below in Register bypassing and early release. Today the only case where these are implemented is a few (not all) cracked instruction cases, but this patent suggests we may see these ideas in a future design.

xxx need to do some tests of cracked instructions here, plus analysis of numbers from dougall's throughput+counter pages

Power aspects of the register file

I'm always reluctant to blame an anomalous result on "power saving" because that's too easy, and can be used anywhere! At the very least before making such a claim, I want to have a valid model in mind of how the power saving might work.
However I think the glitch we saw here is indeed the result of power saving, in the form of the details described above.

Specifically what I am assuming is something like
- we have four register banks
- under the particular delay conditions that lead to the glitch, while the delay is active 3/4 of the registers (assuming 380 registers, that would be 285 registers) land up active and waiting in the ROB. Pretty much the cycle that the ROB clears is the cycle that allocation moves into the fourth and final bank. All allocation then occurs from this bank while the other three banks run in low power mode.

- but the timing is such that: remember I said that the times just balanced so that all the free registers were used up in exactly the same number of cycles that the registers were freed from the ROB... So I think what happens is simultaneously
+ the ADDs are allocating free registers from the one bank that is powered up
+ the ROB is freeing registers that are allocated to the three powered down banks
+ the registers that are freed by the ADDs (which complete very soon) are nonetheless still in the ROB (and not yet marked free) behind all the registers that were allocated during delay

so we get to a cycle where simultaneously
+ the last register is allocated from the powered up bank
+ no registers have yet been released to the powered up bank (though that's about to happen)
+ plenty of registers have been released, but they are all allocated to the three powered down banks

Presumably the system copes by powering up one of the powered down banks, and that takes two or three cycles, generating the glitch we see. In theory this might not actually be necessary -- waiting just one cycle would result in the first set of registered from that fourth, powered up bank, being released, but the machine can't look into the future!

There are still aspects to this that are unclear. Does this bank power-up delay always suspend register allocation for a few cycles like we are seeing here? Or under normal circumstances does the machine have heuristics that indicate it makes sense to begin power up before everything runs absolutely dry, and that power-up can happen "in the background" without requiring all register allocation to halt?

I'm open to alternative explanations for this glitch, but so far I see nothing I consider reasonable except the above.

Context switches

A single core will occasionally engage in context switching, sometimes within the same thread (servicing an interrupt), sometimes between threads in the same process, sometimes between processes. Is there any way we can reduce the cost of these?

dirty flags

Let's begin with the paradigm case, switching between threads in an app. The essentials of how this is done include
- creation of a new thread includes creation of a Thread Control Block, which includes a region of memory that will hold the registers of a thread when it is not running
- switching threads by the OS includes storing all the registers of the old thread to that thread's TCB, then loading all the registers of the new thread from its TCB

One way to reduce this cost is to have some sort of "dirty" flag that is cleared when a new thread is swapped in, and set as soon as the thread uses a register (or more generally, one from a set of registers). It was not uncommon, for example, to do this for either the FP or SIMD registers on the assumptions that many processes did not use these and so time spent context switching them could be avoided. The PowerPC Altivec VRSAVE register is an example.

hardware based register save/restore

Next up in sophistication is to delegate saving register context switches to the hardware. In principle this would involve steps something like
- the OS queries the hardware as to the size of a thread storage block (so it knows how large to make a TCB)
- the threadID/some thread-associated register is the address of the TCB
- a context switch would consist of something like four instructions
move specialPurposeRegister1 oldThreadID
move specialPurposeRegister2 newThreadID
switchOut specialPurposeRegister1
switchIn specialPurposeRegister2

Done correctly, this in theory allows the CPU to add additional state without requiring an immediate OS update, and allows the CPU to engage in the previous "dirty" flag optimizations without bothering the OS. It also allows optimizations like storing the registers directly to, say, L2 instead of wasting L1 cache space on data that will likely not be reused for many cycles (in principle an OS context switch could do the same thing via non-temporal stores, I don't know if any do).
In practice the one ISA that does this, x86, in the usual x86 fashion screwed up every possible angle of the implementation so no-one actually uses it.

apple's 2015 patent (not yet implemented?)

What Apple does is to unite and improve both of these ideas. (2015) https://patents.google.com/patent/US9817664B2 *Register caching techniques for thread switches.*
The basic ideas are
- there is some way (unstated, but presumably based on a special purpose register) to indicate the current threadID, and that a context switch has occurred (which could be as simple as changing the value in that SPR)

- after that point, under "ideal" circumstances the hardware will automatically save out old registers and load in new registers
- but it will do it on demand, not as one large block of time that stores all then loads all.

The idea is to add a new field to the mapping table that maps each logical register to a physical register so that the mapping says
x0 is mapped to physical register p3 with a tag of threadID.
Now imagine what happens right after a context switch. The first new instruction wants to read register x0, so it consults the mapping table. The mapping table tag does not match the current threadID so the following happens:
- we write out p3 as the value of x0, using the threadID tag and some offset algorithm to know where to write x0 in the TCB
- we load in x0 using that same offset, but based on the the new threadID, and we place that new threadID in the mapping table tag for x0
This is the basic idea, and it has as a consequence that writing out and loading in the new register values will be spread out over time, hopefully mostly interleaved with real computation.

There are a number of details to make this more performant.
- First is that these tags may be associated with more than one register. It would make sense, for example, to have a single tag for two integer registers since loading or storing two is no more expensive than loading or storing one. Presumably you pair these based on statistics about what registers tend to be used together.
- Secondly the stores are straight to L2, without wasting space in L1. It's unclear if this is done at cache line granularity (ie, as an extension of the first point above, aggregate under one tag enough registers to pack a cache line, and store these out sequentially filling a single cache line transfer buffer); or perhaps at some smaller granularity that's an optimal match to the bus width between L1 and L2 (maybe 32B quarter of a cache line?)
- Third the threadID tags are in physical address space, not virtual address space. This I assume makes life easier for the OS, and allows a minor advantage of not having to perform TLB lookups (and perhaps some other slight streamlining relative to a normal load/store). It does make me wonder the nature of the connection between L1 and L2 and how partial lines are handled. This use of a physical address also mean the OS has to be careful about these TCB pages; they can't be swapped out or compressed or similar shenanigans!
- Finally there's an additional flag that is set when a register is modified (ie a dirty flag) to be used in the obvious way, to avoid having to save an unmodified register.

Note also that this means the CPU is capable of doing something very strange, namely creating synthetic instructions, with no warning (and allocating whatever resources they might need) in the middle of the pipeline! This is something I've never heard even remotely considered in any other CPU, and Apple has provided zero patents on it, at least that I've found. These synthetic loads and stores (required to write out the old thread's version of a register and load in the new thread's version) are close enough to normal instructions that they fit nicely into the operand dependency model we're

about to discuss. The machine does not stop while the registers are brought up to date, instead the instructions that depend on say x0 are given additional dependencies that they cannot proceed until these synthetic register load/store instructions have completed, but other OoO behavior can as usual route around them. Much later we will see another version of this creation of synthetic instructions when we look at Apple's (currently very limited) use of value speculation.

Note how nicely general this scheme is. It can (not necessarily unlikely, for the SIMD registers) allow a SIMD register to persist, unreferenced, through multiple context switches and still be there when the initial threadID runs again. It can also easily be adapted (with an appropriate interrupt "threadID", perhaps NULL) to interrupt handling, allowing handlers to use whatever registers they feel like, and have these transparently saved as appropriate then reverted on return from interrupt.
The one place you have to be careful is: what if a thread moves to a different core? The patent suggests having (Apple custom) instructions that do things like force flush all the modified registers to the TCB.

The scheme as I have described it is general and could be used for all the registers that are stored during a context switch, which extends to the NZCV flags register, the FPSR floating point status register and possibly various SPR's. As a practical matter, the SIMD registers (and the AMX registers?) are the most likely immediate candidates. My intuition says that even the integer registers would benefit, especially, as I suggested, for interrupts, but who knows *exactly* what Apple has implemented

It should be pointed out that I have engaged in some correspondence with an author of the 2015 *Register caching techniques for thread switches* patent. His belief (though he cannot be sure because he left Apple a few months before the patent was filed, let alone any hardware released) is that while the idea is valid and will work, it has not yet been implemented in an Apple core.

It's interesting to compare this scheme (perhaps not yet implemented) with the register security tag scheme I described at the very start of this document., and which does appear to be implemented. The security tag is a much simpler mechanism than context-switching, but is a very nice half-measure. It requires a similar tag associated with each entry of the architectural to logical mapping, a validation of that tag on every access, and machinery to update the tag under certain conditions. But it does not require the subsequent steps of automatically writing registers values out to DRAM, or reading them in from DRAM.
Hence it would be not crazy to imagine that the context switch mechanism might eventually be implemented, by expanding the existing security mechanism.

apple's 2019 patent (reduced context)

Paired with this we have the related but different (2018) https://patents.google.com/patent/US20190220417A1 *Context Switch Optimization*.
This is a very strange patent, meaning I have to be rather speculative in trying to understand it. The basic idea is very easy: suppose that Apple defines, for some processes, an alternative slightly simplified version of ARMv8 that uses a subset of the registers, for example it uses half the integer and one

quarter of the SIMD registers. Defining this is not too hard; it's mainly small modifications to the compiler. But if we do this (for whatever reason) and rely on it, then we want the CPU to flag when we don't follow the rules. Thus we define a reduced processor context (the set of allowed registers), a register (swapped on context switches) that states whether the processor is operating with access to all registers or (possibly more than one alternative) reduced processor context, and some minor machinery to generate an exception when an inappropriate register, for the reduced context mode, is touched.

Now why bother to do this? Hypotheses:

• the 2015 patent sounds good, but if it iss not yet implemented, then the actual cost of context switching remains a notable block of time one might want to reduce. If the OS can define many of the ongoing background processes as reduced context, it can save context switching time.

• even if the previous patent is in place, and even if you modify the compiler to use fewer registers, without some sort of explicit contract, the OS has to allocate a full-sized TCB to hold all the ARMv8 ISA registers. If we want to save some space (again for these many background processes and OS threads) because we know they have minimal register requirements, it's not enough to just compile them with fewer registers, we need to also have the CPU's automatic register saving mechanism incapable of overwriting the bounds of a reduced context TCB.

relevant for chinook?

• (most speculative) this has nothing to do with either Firestorm or Icestorm! Few people know that, along with Apple's large and small cores, there is at least one other core they have designed, a tiny core, which is the controller for things like the GPU, NPU, ISP and suchlike. We know that the codename for this core in the A12 was Chinook, and that it is an AArch64 core (apparently with NEON) at the same architecture level as the A12 (eg it provides Pointer Authentication). It would be reasonable to assume, given how Apple has designed the small vs large cores, that this is mostly a "parameterized" version of the basic large core design (ie same overall OoO structure, branch prediction, scheduler, etc) just with even fewer of everything than the small core.

With that in mind, consider that Icestorm has ~80 physical integer registers and ~88 physical SIMD registers. Maybe Chinook has the bare minimum the design can support without locking up (?34 of each or whatever), and is most performant when actually locked into a subset of these, using 16 integer and 8 SIMD or whatever?

The justification for this wild speculation is

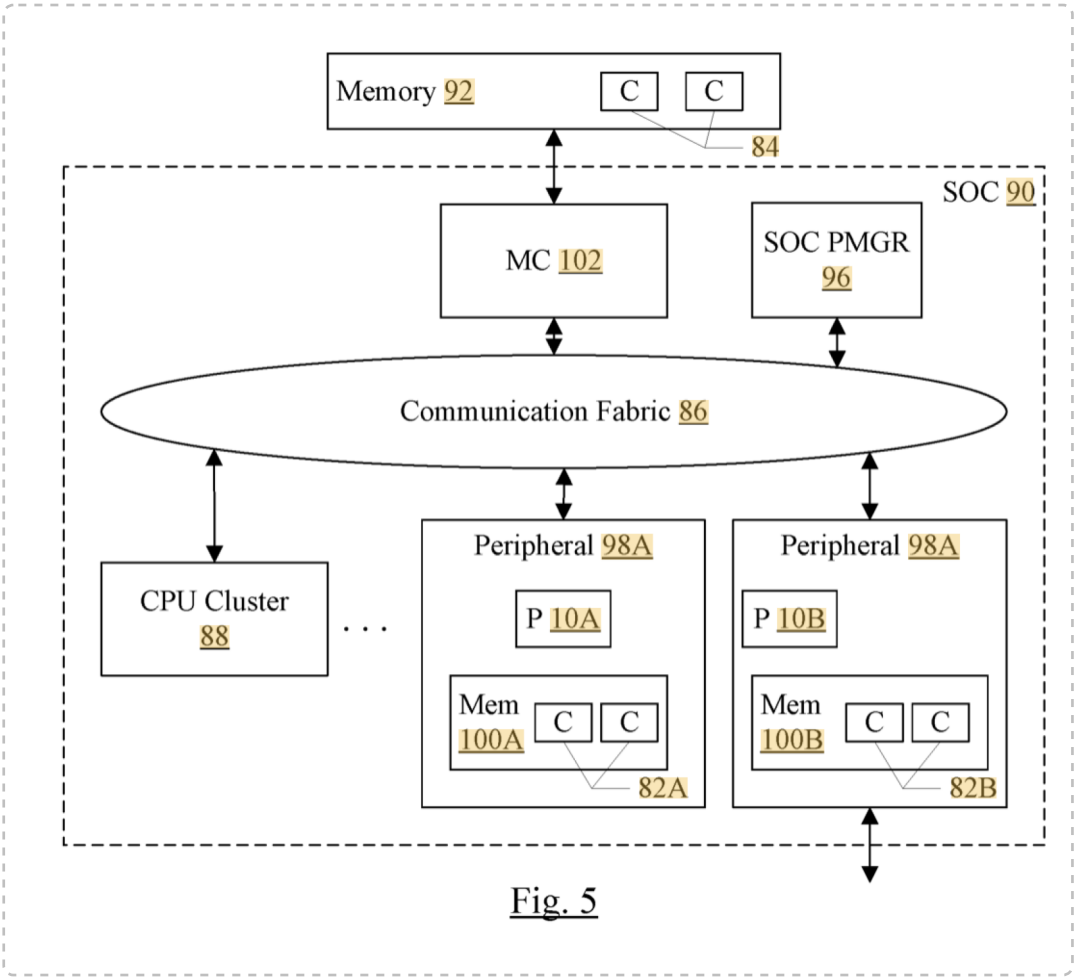


Fig. 5

with accompanying text (removing some irrelevant parts)

- The workload of the processors 10A-10B may be characterized as having more frequent context switches than the workload of the CPU processors in the cluster 88. In some cases, the context switches may be much more frequent (e.g. one or more orders of magnitude more frequent). Additionally, the workload of processors 10A-10B may also be characterized by infrequent, but non-zero, use of one or more data types specified in the ISA. For example, in an embodiment, the workload may include infrequent, but non-zero use of vector registers. Accordingly, reducing the context saved and restored in the processors 10A-10B may be significant in terms of improved performance, reduced power consumption, and memory footprint...

The size of the local memories 100A-100B may be limited, e.g. compared to the memory 92, and storage in the local memories 100A-100B may be used for other data besides the contexts 82A-82B, so reducing the context memory footprint may improve performance as well since more local memory space may be available for process data other than context save data.

- The peripherals 98A-98B may be any set of additional hardware functionality included in the SOC 90. For example, the peripherals 98A-98B may include video peripherals such as an image signal processor configured to process image capture data from a camera or other image sensor, display controllers configured to display video data on one or more display devices, graphics processing units

(GPUs), video encoder/decoders, scalers, rotators, blenders, etc. The peripherals may include audio peripherals such as microphones, speakers, interfaces to microphones and speakers, audio processors, digital signal processors, mixers, etc.

In other words, although this may seem initially like crazy speculation, I think it's reasonable to interpret the patent as essentially confirming
• the existence of Chinook
• that it's used as a general purpose controller all over the SoC
• that it's essentially a further scaled down Apple small core, not a separately designed bespoke core
• and that Apple makes it "effectively" smaller by things like careful ABI.

It could be argued that any sort of AArch64 OoO design is far more than what's needed for many of these tasks, but one has to wonder if that's 1990's thinking. The Apple solution may use a little extra area (cheap) but in return delivers
• security (this isn't a second class core; it gets the full security treatments of all the other Apple cores including things like TLB protections and PAC)
• easy integration with existing developer tools, compiler, and the OS
• enough performance that engineers can spend their time worrying about how to make the entire device better, rather than worrying about how to make an ARM M4 do whatever needs to be done.

Register bypassing and early release

We've described multiple ways that Apple can make the physical register file appear larger. But they haven't taken the next step in doing so, namely using virtual registers or something similar (for late allocation) and allowing for early register de-allocation.
They clearly have thought about the issue, as in this patent: (2017) https://patents.google.com/patent/US10691457B1 *Register allocation using physical register file bypass*.

Consider an instruction sequence like `REV x0, x0; CLZ x0, x0; ADD x0, x0, x1`
Note two things.

- First the data that is transferred from the `REV` result into the `CLZ` input does not need to ever be stored anywhere – there is no way to access it after the `CLZ` executes. So why not omit allocating a register, and just have the `REV` read the value from the bypass bus?

- Second the `ADD` overwrites the value of the x0 output from the `CLZ`, so even if we did allocate a physical register to the result of the `CLZ`, why not just deallocate it right away because, once again, no code can access it after the `ADD` has overwritten logical x0?
This gives us two different techniques for requiring fewer physical registers, one avoiding the allocation of a register by use of a tag, the other allowing for early deallocation of a physical register.
These are both nice amplification technique -- not incredibly powerful, but also not especially difficult.

However a test on the M1 suggests that, in spite of the patent, neither appears to be implemented, at least not as of the M1.

The test sequence I used above is drawn from the patent (so you'd hope it would demonstrate the effect, if anything!) However, to simplify the problem I also tried the easier test sequence (no complications as to which execution units different instructions can execute in)

```
ADD x0, x5, x5; ADD x0, x0, x0; ADD x6, x7, x7; ADD x6, x6, x6;
ADD x8, x9, x9; ADD x8, x8, x8; ADD x10, x11, x11; ADD x10, x10, x10
```

As expected this will run at 6 instructions per cycle, but more interesting is, if we add in a delay, how many instructions will execute before the jump (ie how many registers will be allocated). Note that these pairs all satisfy the conditions of the patent – the instruction result is generated to a register which is immediately overwritten by the next instruction. Even so, we see the jump at the usual ~380 instructions, no difference in the apparent register file size.

One might ask why this is not implemented, especially given that, as we will see, an equivalent is implemented for load/store?

Before answering this, note is that there *are* apparently cases where cracked instructions [like `ADD (shifted)`] are implemented as

xxx test/explain the cracked case

- two instructions where
- the second instruction reads the intermediate value straight off the bypass bus without allocating a register, as we discussed earlier.

So why not implement the same mechanism more generally?

I suspect the issue with using tags as the patent describes is that you *also* need a way to force the two instructions to schedule together – it doesn't do you any good to have the second instruction plan to read the value off the bypass bus if it isn't scheduled *immediately* after the first instruction. Presumably this tying mechanism exists for the case of cracked instructions, but Apple hasn't yet implemented something allowing generic tying of instructions? Ideally such a solution might be part of a more generic fused instruction mechanism, which is one reason it might be delayed.

Similarly, consider the failure cases for virtual registers, eg what if, at the point of instruction execution, you discover there is no free physical register available? It's not an impossible problem, but once again it requires some support machinery that won't exist until you add it.

What about implementing the alternative of early register release?
In that case, my guess is the fly in the ointment is the last physical register scheme, a patent I have already referenced, (2019) https://patents.google.com/patent/US20210064376A1.
Getting that scheme to work along with early deallocation is not impossible but, once again, it's one more thing that needs to be figured out. It's certainly possible, perhaps even likely, that the initial machinery for these various ideas is in the M1/A14 hidden behind chicken bits, but ready for later release when they're considered 100% working.

How rapidly can the ROB retire instructions?

We still haven't nailed down everything register related!
We know that registers are released at Retire (no early release) but how rapidly are they released at Retire? The minimal rate must be 8/cycle (to maintain 8-wide throughput) but could it be larger?

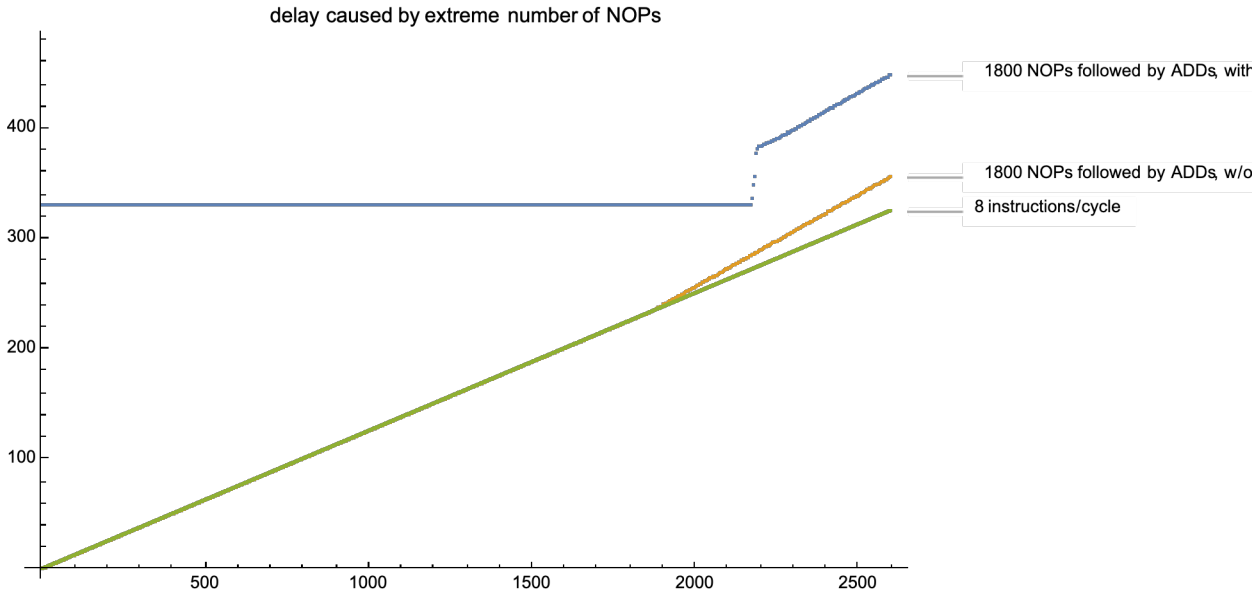
Why do we care about this? If we're waiting for an integer register to be released when head of ROB clears, what's the problem if 8 integer registers are released per cycle, given that integer execution can only use 8 integer registers/cycle? The issue is that after a long delaying head of ROB (like a miss to DRAM), there are many instructions, all blocked on different things.
Yes, there are some integer instructions blocked on an integer register. But there may also be some fp instructions blocked on fp registers, likewise for flag registers. And there may be load or store instructions blocked waiting for load or store queue slots. So ideally when the head of ROB clears, we'd like to run through the ROB as fast as possible, freeing resources as rapidly as possible, so that as much stalled activity as possible can resume as soon as possible!

retiring NOPs (56/cycle)

Given this insight, let's start with an easier case.
We begin with a delay block consisting of our usual `FSQRT`, followed by a large number (~1800) of `NOP`s, followed by instructions that use up all the int registers then wait on them.
The question of interest is how long it takes to clear the `NOP`s (ie clear out the head of the ROB) before the `ADD`s can start work.
So the structure is
- delay block of 33 `FSQRT`s (33*10 =330 cycles; needs to be long enough to be sure everything is packed into the ROB and scheduling queues!)
- delay block of 1800 `NOP`s (should take 1800/8=225 cycles, to pack the ROB behind the `FSQRT`s)
- resource depletion block of 370 `ADD x0, x5, x5` that should use up all the physical registers
- probe block of variable number of `ADD x0, x5, x5`

The probe block should not be able to run until more physical registers become available, which cannot happen until both the delay block ends *and* the NOPs are cleared.

Out[120]=

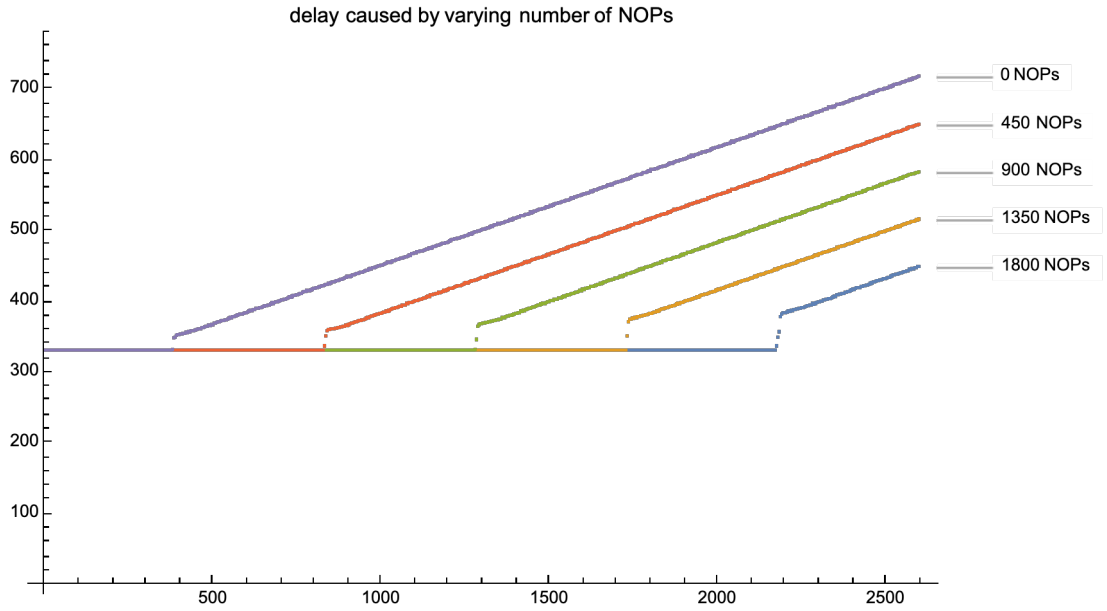



```
{2172, 330}, {2176, 336}, {2180, 348}, {2184, 356}, {2188, 377}, {2192, 381}
```

Well that's nice and clean -- once you figure out the correct probe!
The gold curve is, of course, the non-delayed version. Easy to see the break in the slope as we switch from 8/cycle `NOP`s to 6/cycle `ADD`s.
The obvious fact about the blue curve is that the jump is at ~2180=1800+380, exactly where we'd expect (stall once the `ADD`s run out of physical registers).
But more significant is that the ramp jump is about 53 cycles. Of course much of that could be because of xxx time, ie the time spent waiting for the `FSQRT`s to complete, which is not interesting. What we care about is the time spent clearing the ROB after the last `FSQRT` completes.

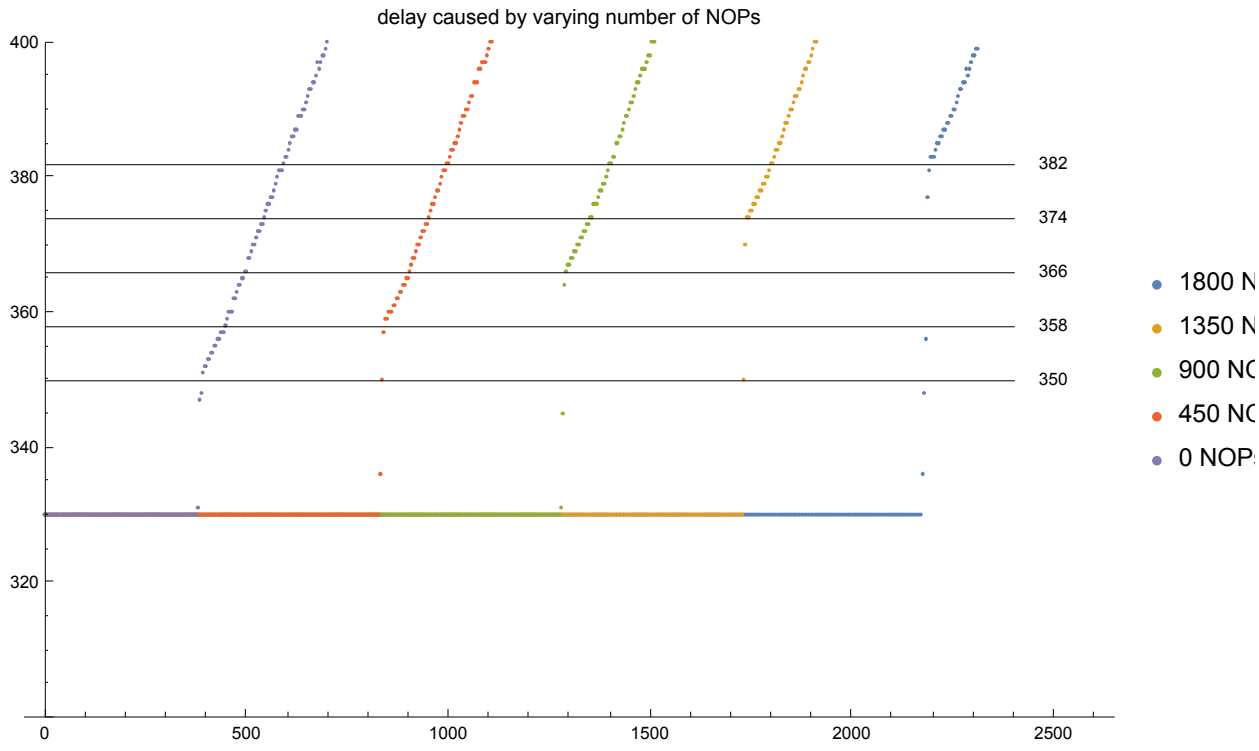
We can get better insight into that by varying the number of NOPs.
Consider the image below, where we use the same structure, but vary the number of NOPs from 0 through 450, 900, 1350, to 1800.

Out[125]=



There's clearly a variable delay (look at the size of the jump) from when the machine stalls until when ADDs start being processed again, and that stall grows as the number of NOPs grows.
What's the exact relationship? Let's zoom into the relevant part of the image and add some guide lines.

Out[128]=



This image looks messy but really all we have done is zoom in on the interesting part of the original image (the jumps) and overlay some lines.
There's room for disagreement, but to my eye the lines are plausible cycle values at which the ramp of

each curve starts.
The interesting point is that the lines tell us the amount of delay (the time it takes to again start processing ADDs from when the machine stalled.
There is a baseline delay of 20 cycles (even with no NOPs). More interesting is that the additional delay increases by 8 cycles for every 450 NOPs.

So the time it takes the ROB to clear 450 NOPs is 8 cycles, ie the machine clears 450/8=56 NOPs/cycle!

structure of the ROB

We will see evidence for this later as we cover load/store and branches; but in fact the ROB is best thought of as consisting of ~330 “rows” where each row can hold 7 instructions. Most instructions can go in any slot, but “failable” instructions must go in the last slot (which provides extra storage). Failable instructions are those that might require the CPU to flush and restart. These can be branches (mispredicted...) or loads/stores (possibly some exception conditions, but the main case I am thinking of is a load/store dependency misprediction when a load occurs out of order relative to a store and so reads possibly invalid data [all to be explained later].

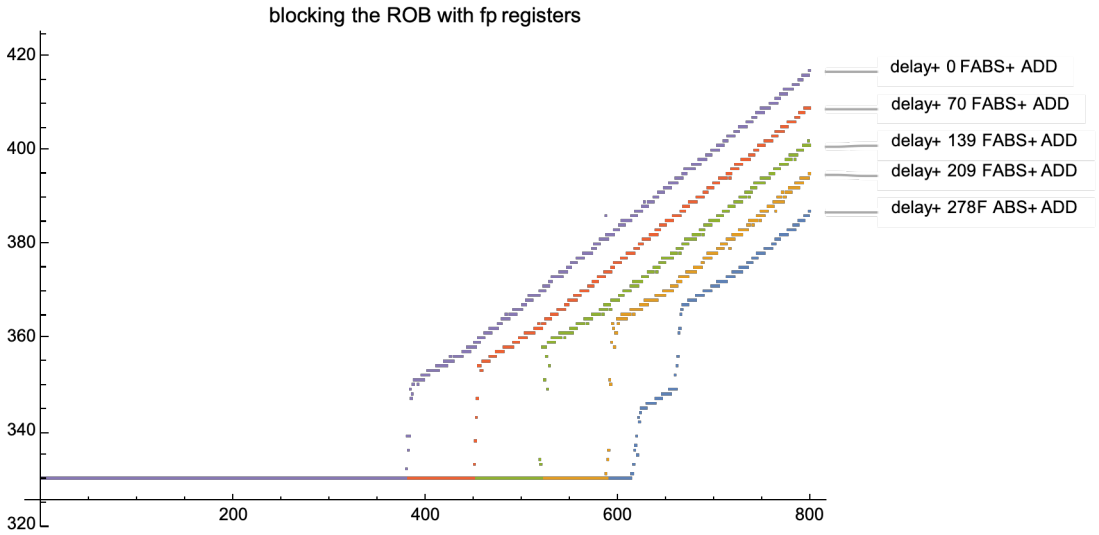
The usual structure of the ROB, when not running weird test code, will look something like maybe two or three instructions in a row, then a load at the end of the row, then maybe five instructions, then a branch terminating that row, and so on. The ROB slots are cheap, so Apple is happy to give us thousands of them! The real constraints in real code will be either load/store/branches filling up the ~330 failable ROB slots, or int/fp code using up all the HF slots.

So a better way to think of Retire is that Retire can clear 8 ROB rows per cycle. This can ultimately mean clearing as few as 8 instructions (if we had eg a sequence of eight successive loads, or some combination of successive loads, stores, and branches but nothing else). Or it could mean as many as 8*7=56 instructions if every one of the eight rows was fully populated.

freeing up registers and HF slots (16/cycle)

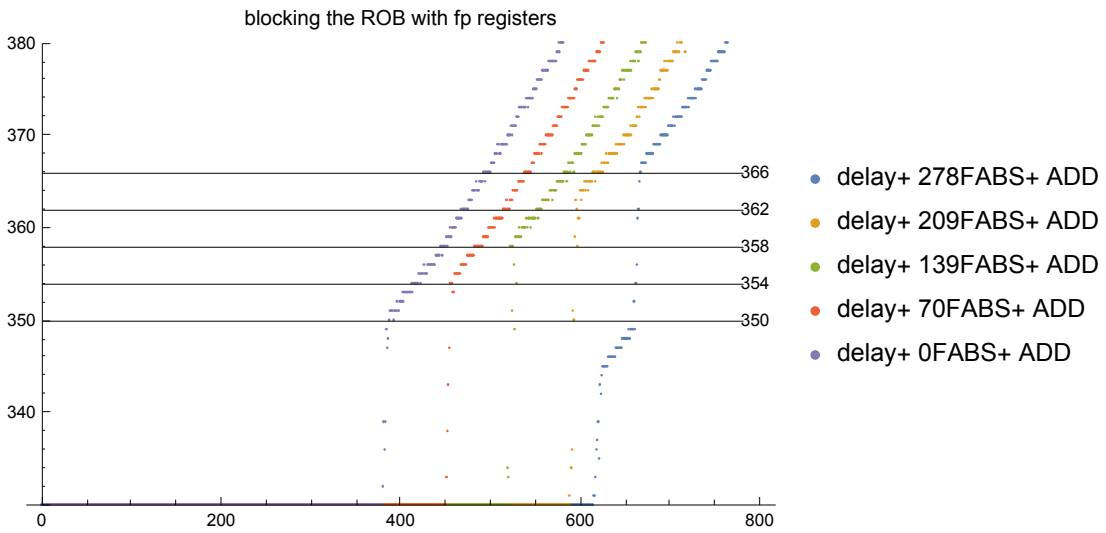
That's fairly impressive, but clearing NOPs is easy. What about a more difficult task like restoring registers?
The plan now is to replace the NOPs with FABS, which will use up slots in the History File. How many registers can be freed per cycle?
We want to use enough FABS to make the phenomenon visible, but not so many that we flood the History File with floating point register manipulations and don’t leave space for all the ADDs. Using a maximum of 278 FABs seemed to do OK.
If those can be cleared at 56/cycle, clearing the whole 278 will take 5 cycles.
Hopefully we can see that against the 20 cycle baseline (and the noise that will arise from using FP for both this timing and the delay.

Out[137]=



Well clearly the delay is a lot longer than 5 cycles! So let's use the same "step through quartiles" trick we used for NOPs, and zoom in.

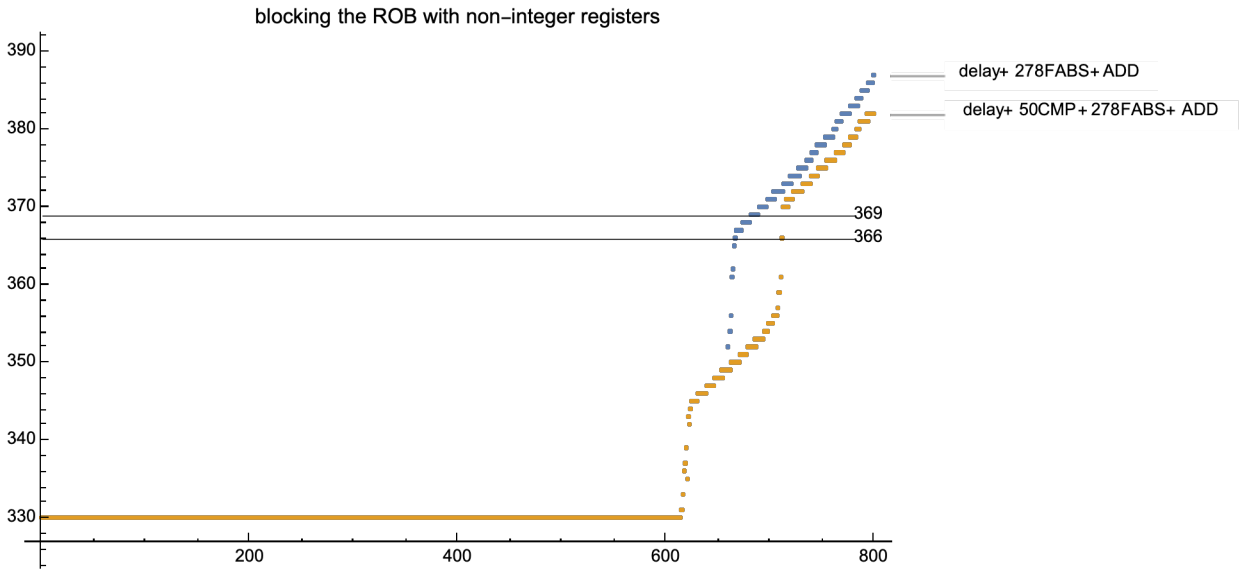
Out[140]=



I think it's legitimate to see clearing 70 `FABS` from the ROB as taking 4 cycles, so 17.5 clearances/cycle. Probably best to read that as 16 per cycle.

We can attempt to validate the above hypotheses by prepending, before the 278 `FABS` that use up physical FP registers, 50 `CMP`s that use up physical flags registers. If our hypothesis is correct, this should result in an additional delay of another three cycles (50/16=3).

Out[143]=



We see the point is validated; an additional delay by what can plausibly be viewed as an additional three cycles.

Obviously the methodology I am employing is not great for incontrovertible, exact measurements! But I'm interested here in large-scale exploration to figure out the overall design of the system; I'll leave it to others coming later to perform the high precision experiments.

Experts might want to know why the curves show two jumps. Consider the blue curve, and compare with the previous graph.

The ADDs face two possible constraints: physical registers and HF slots.

When there are not too many `FABS` enqueued (the cases of 70, 139, 209 `FABS` above), then the `ADD`s run out of physical registers first and block.

But when there are enough `FABS` enqueued (the case of 278 `FABS`) then the first constraint that is hit is running out of HF slots.

After the `FSQRT`s clear the `FABS` start to clear, and HF slots are released. This allows `ADD`s to restart for a few cycles – until they now run out of physical registers. The `FABS` HF slots were releasing FP, not integer physical registers.

The `CMP` case is an even stronger version of this. Think about it. In the `CMP` (gold curve) case at any given N value, 50 fewer `ADD`s have been enqueued than for the blue curve case. We still run out of HF slots at the same point because `CMP`, `FABS`, and `ADD` all use up slots from the same HF pool; but once HF slots start to be freed, there are 50 more integer physical registers available than in the blue curve case, hence the intermediate run before the second delay (running out of integer physical registers) can proceed until N is 50 operations higher.

freeing just HF slots, not registers (also 16/cycle)

There's one more thing we need to test.

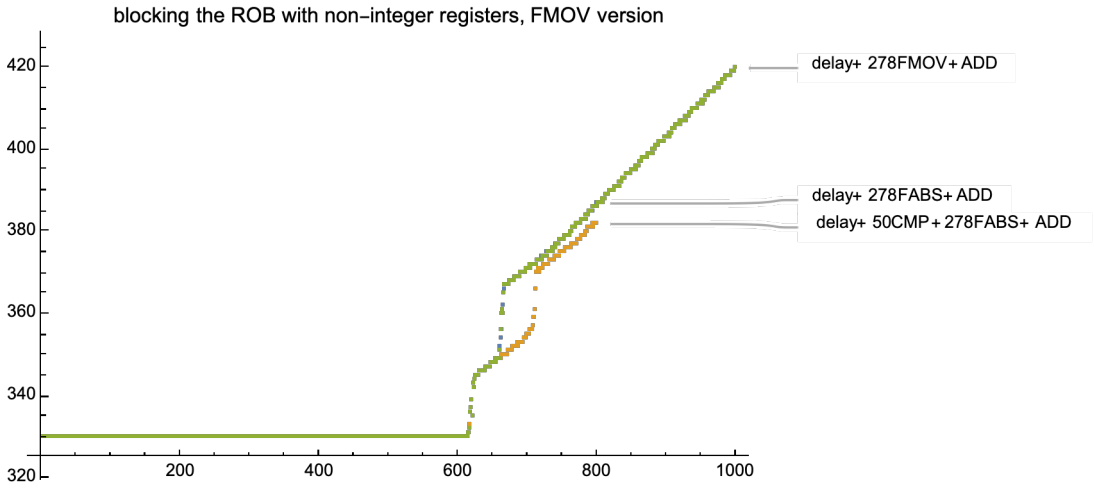
To clear entries from the ROB to eventually free up those integer registers, we have to

- clear entries in the HF, and
- release registers to the free list.

It's possible that these two tasks take a different amount of time (for example releasing HF slots happens rapidly, but then moving registers onto a free list so that they are available for reuse happens at a slower rate.

So let's try a variant that doesn't involve freeing registers, just HF slots.
Replace the `FABS` with `FMOV d31, 30.` The idea is that while clearing a FABS entry from the HF involves freeing an actual physical register, almost every FP HF entry only records a change in the mapping tables; it doesn't actually change a register to be free to move to the free list. The idea is to test if manipulating physical registers in Retire is slower than manipulating HF entries.

Out[145]=



As you can see there is no difference!
It appears that HF slots and registers are both freed at 16/cycle, there's no faster path for HF slots that are not freeing a physical register.

so how close is the M1 to a KIP (kilo-instruction processor)?

Just as a fun test, how close is Apple to a KIP (kilo-instruction processor)?
A KIP has been the CPU designer's dream since ~2000CE, ie a design that can maintain "in progress" one thousand instructions, the idea being (at the time the phrase was coined) that this would be enough usually to keep a CPU from stalling even if a load missed all the way to DRAM. Unfortunately as CPUs have become higher GHz and wider, 1000 instructions is no longer enough (if a miss costs you, say, 330 cycles, and you're 8-wide, you'd like to be able to power through 8*330 instructions before stalling), but the number is still a good milestone.

Obviously Apple is way beyond this 1000 limit in trivial fashion (sequence of NOPs) but what about real code?

As explained, the ROB size itself is not a problem, it's various other structures that are more difficult to scale up (eg LSQ, physical register files, history file) and we have seen how Apple has continually

disaggregated or rethought these into different structures that scale better (perhaps parallel structures like the banked physical register file, perhaps move the most difficult to scale part out of a structure as we will see with the splitting of the traditional Load Queue into the LEQ and the LRQ).
Given all this, if we want to pack as many “real” instructions into the ROB as possible, what are the limits?

The first is that anything that changes a register will use a History File slot, and we have ~620 of those. But we also have some instructions that don’t change the register file, most notably stores and branches. Unfortunately stores and branches (and loads) use the same “failable” slot of the ROB, even the simplest non-problematic and non-conditional branches like “B .+4”, so we might as well just use stores (since branches are also limited by other data structures beyond the number of failables).

If we use the probe (`ADD x0, x5, x5; FABS d0, d0; STR x5 [x2]`) we can in fact reach about 310 iterations of this before he see a jump!
So ~930 realistic’ish instructions! Not bad! It might be possible to go slightly beyond this with a few other instruction classes; I didn’t push the issue.

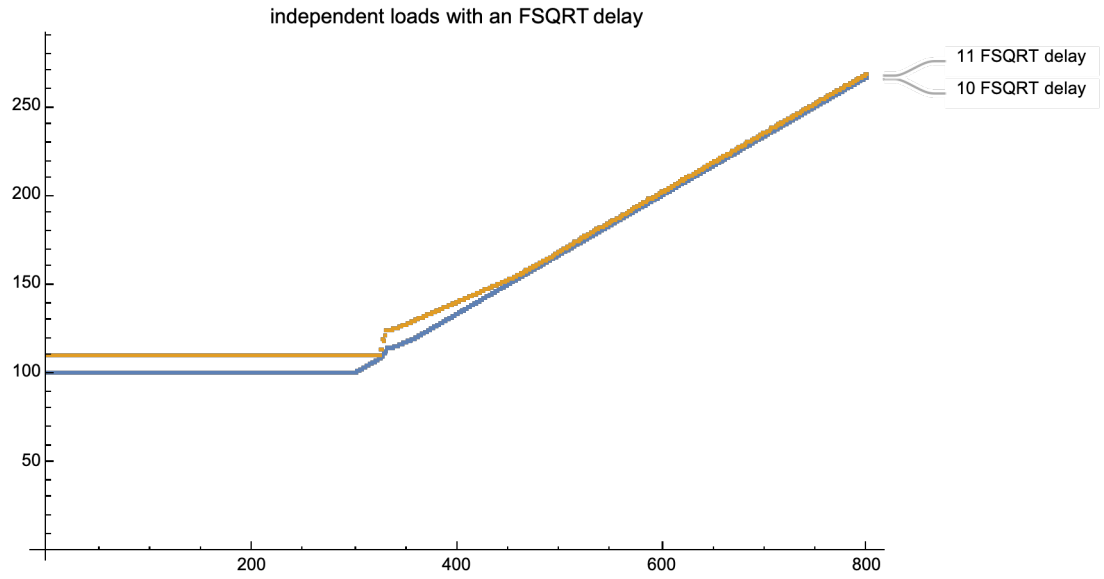
Loads and Stores

Experiments to test LSQ sizes

first attempt at testing queue sizes (FSQRT based)

We' ve so far explored some aspects of the ROB, and the size of the physical register files. Now let’s explore the size of the load/store queue.

Out[148]=



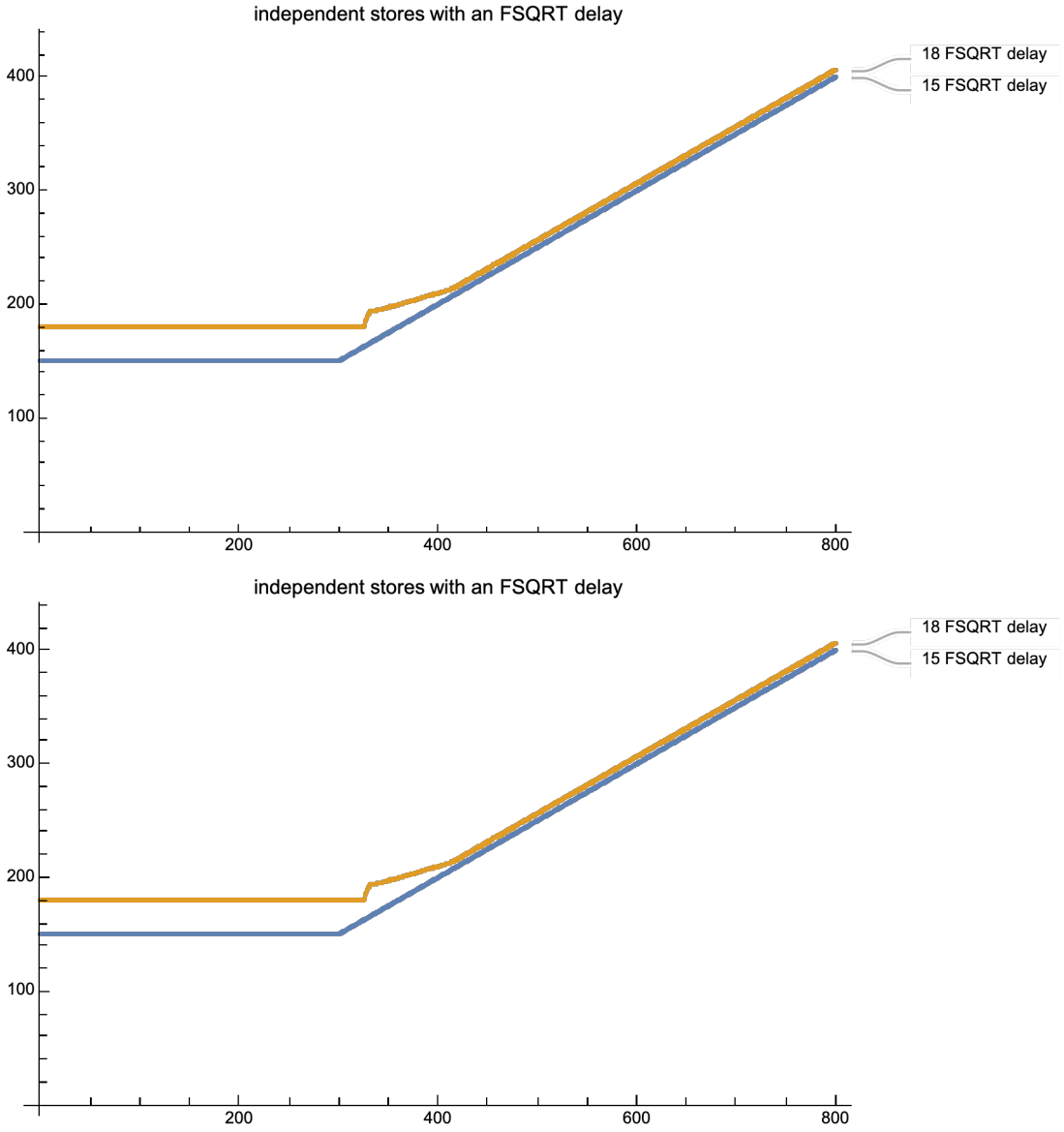
The gold curve gives us the immediate info of interest, a jump at ~330, which would suggests that the LSQ can hold 328 loads. But there are unexpected issues here!

- First is that the limit seems suspiciously close to the size of what I have called the “failables” part of the ROB; ie the limit looks like how many loads we can hold *in the ROB*, not in the LSQ.
- Second is that we don’t seem to actually lose any cycles! Yes we have the jump at ~330, but that’s followed by a regime where we appear to be executing loads at 4/cycle rather than 3/cycle, till we get back on trend!

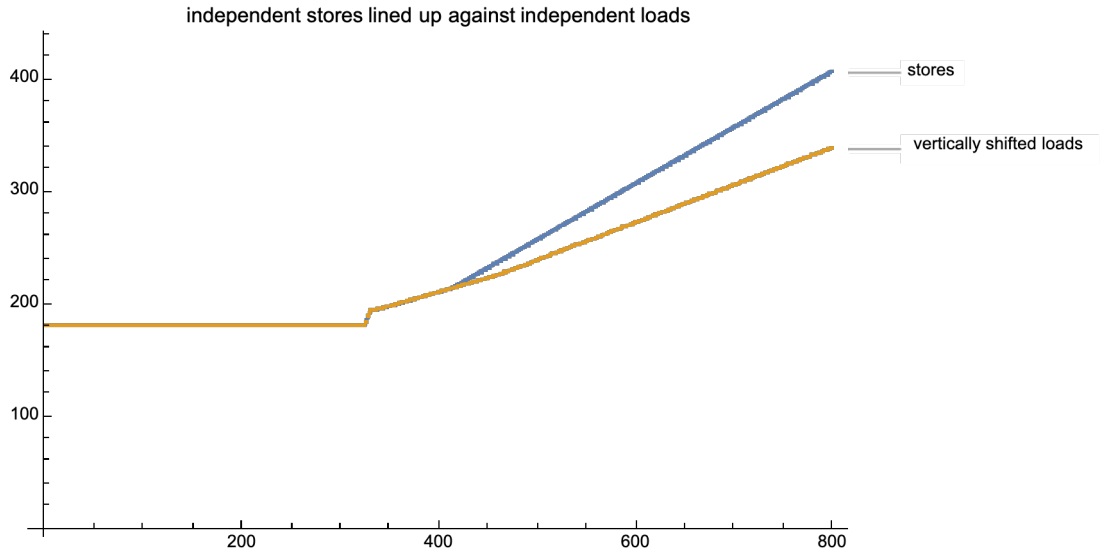
So there’s something weird going on.

Do we get the same behavior for stores?

Out[151]=


independent stores with an FSQRT delay
18 FSQRT delay
15 FSQRT delay
400
300
200
100
200
400
600
800
independent stores with an FSQRT delay
18 FSQRT delay
15 FSQRT delay
400
300
200
100
200
400
600
800

Out[152]=

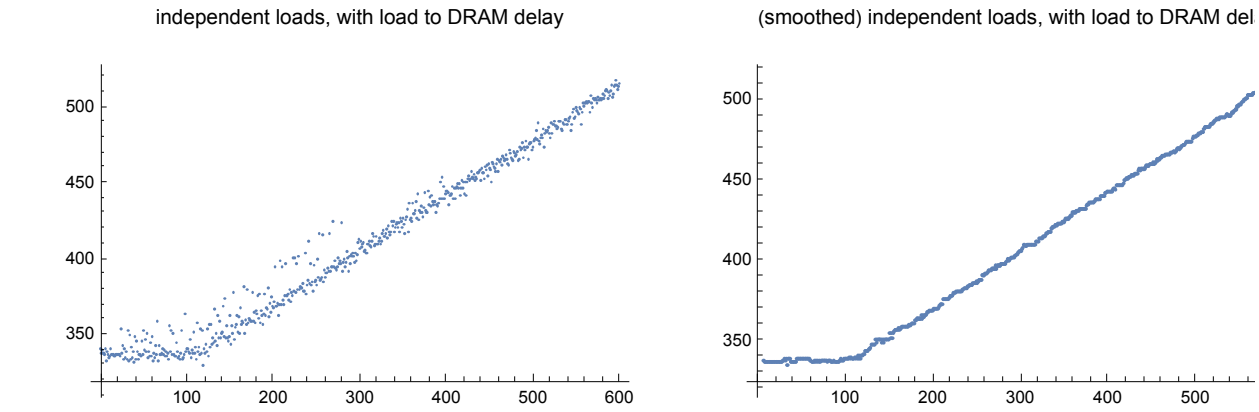


We see exactly the same pattern for loads and store (ie
- same jump at ~330,
- same running at about 4-wide till N=~420).
The difference in slopes after that arises from M1 having
- 2 load units
- 1 store unit
- 1 ambidextrous unit
so that it can run either 3 loads/cycle or 2 stores/cycle.
These results seem ...unlikely... They indicate a massively large load-store queue, same sized for loads and stores.

second attempt at testing queue sizes (miss-to-DRAM based)

Let's try a different tactic. Instead of using our trusty FSQRT as a delay block, we'll use a delay block based on loads that continually miss to DRAM.

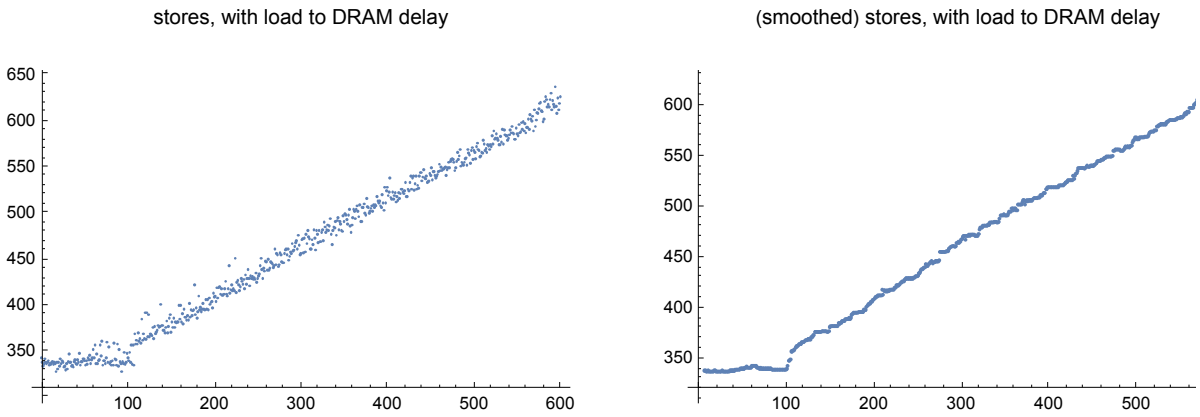
Out[156]=



As you can see, the problem with using a load delay is that there's a lot of noise in the signal, and while we can reduce this (at the cost of much longer runtimes) we'll be cheap and just use a smoothing filter (moving median) to make it easier to see the point. And the point is that, with this delay, as opposed to with an FSQRT delay, there's a discontinuity at N=~125!

We see the same pattern with stores, in this case with a discontinuity at around 100 stores:

Out[160]=



So we find ourselves with the following conclusions:
• there appears to be load queue, of size ~125 loads
• there appears to be store queue, of size ~100 stores
• but these do not behave like our previous constrained resource tests!

Clearly we need to understand exactly what the load/store queue(s) do, how they work, and how they can be optimized.

Theory of the load store queue

The load/store queue is a large, complicated subject. Even more so than the rest of the CPU, there are technicalities in this area that are specific to x86, or specific to Intel, but which are not universal constraints on how things have to be done. Be open-minded!

why do we need a store queue?

First the store side: Why do we need a Store Queue at all?
Once we have a speculative CPU, this means that we are going to be executing *speculative* stores. But we cannot allow such stores into the cache for two reasons, one obvious, one slightly less so.
- The obvious reason is that if you overwrite a value in the cache, then discover that the path of execution was misspeculated, what can you do? You can't undo the write and recover the previous value that you overwrote!
- Less obvious is that once you write to the cache then, (more or less), that value, although still speculative, becomes visible to other CPUs via snooping, and once again, you can't undo whatever those CPUs

do in response to that snooping, even if you want to undo the write.

So we need somewhere to hold each pending store until that store becomes non-speculative.
It is traditional to make this storage a queue, with the oldest stores at the end and newest at the beginning. The reason for this temporal order is that the code may very well generate multiple stores to the same address, so you need to be careful about the ordering of stores (and also, as we will see, of loads).
Remember that our CPU is out of order...
So imagine code that looks like this
store xM into xA
...
store xN into xB+xC
...
It is possible that xA=xB+xC. It's also possible that one of these stores may *execute* in reversed order, eg maybe xA is generated by a slow load, whereas xB and xC are already available in their registers. If we don't get ordering correct, we'll store xN, then overwrite it with xM, ie the reverse of correct program order...

So the basic idea becomes clear. To ensure correct store behavior given both speculation and out-of-order execution, we want something like
- a time ordered queue to hold stores, for which
- each slot looks something like (store address, store data, various flags [valid, free, etc])
- queue slots *allocated at Rename* (ie at a point where the instructions are still in order)

And the most basic flow looks something like
1) allocate a store queue slot at Rename
2) store sits in the issue queue for the LS unit until *both* its (store address and store data) dependencies are satisfied
3) (store address, store data) are written to the queue slot
4) ...at some later point... the store address has to be tested against the TLB to make sure there are no permission issues, no VM fault, etc
5) the store instruction reaches the head of the ROB. Assuming the TLB issues didn't generate a fault, the store can complete.
6) ...at some even later point... the store data is written to cache, and the store queue entry is freed.

Pretty much every step here has interesting non-obvious wrinkles, and we'll return to them, but for now accept this framework.

why do we need a load queue?

Now think about loads. Why do we need a load queue? Well think about this:
We have a bunch of pending data in the store queue, and loads whose addresses match those earlier stores have to get their data from the store queue, not the stale data in the cache.

And they have to get the *correct* version of the data, the *latest version that was stored just before the load*, from the store queue.

It is for this reason that we generally talk about the LSQ as a single queue that holds loads and stores. This becomes clear when you think of loads, and remember that loads can also happen out of order relative to each other, and relative to stores.
The idea, then, in this most basic model, is that a single LSQ has slots that hold both loads and stores, allocated in program order at Rename. Then,
- when a load executes, it scans the LSQ backwards in time from its position, looking for the *nearest* store slot with a matching address, and reads that data.
- when a store executes, it scans forward in time, looking for *every* load slot (there maybe more than one) with a matching address, and feeds that slots its store data.

This model has to deal with various dependencies you may not have thought of that have to resolve before load or store execution can proceed.
For example:
Suppose that (once again, out of order!) you're a load looking along the queue backwards in time to find a matching address. Maybe some of those stores
- have not executed yet,
- so there are slots that you know are store slots,
- but the address has not yet been recorded.
What to do? In the model we have described so far, we have to delay execution of the load until we can be sure that *every* earlier storage slot has its address attached, so that we can be sure we read the correct data from the store slot if there is a matching earlier address.

So, then, the basic load instruction cycle is something like
11) allocate a load slot, in order, at Rename, from the common LSQ
12) load sits in the issue queue for the LS unit until its (address) dependencies are satisfied
13) load scans backwards in time along the LSQ looking for matching addresses.
14) if there are any holes in that scan (ie stores with unresolved addresses), wait till they are filled
15) if there are matching addresses, grab the data from the nearest match, otherwise from the cache
16) ... at some point in the previous 3 steps, validate TLB/faulting/permission issues...
17) load instruction reaches head of the ROB, and its LSQ slot can be freed

Note also that we now realize we needed to include a (3a) step for stores, involving scanning the LSQ forward for matching load addresses.
And we're just getting started!

speeding up the basic design

So with this pool of ideas in place, let's now consider various issues.
Of all the complications, the biggest is the problem of loads and stores possibly having the same

address. Forcing loads to delay for every store loses us a hefty fraction of our OoO performance. And, of course, most of the time loads and stores are to different addresses! We are making the machine run a lot slower for a case that's very rare – but has to be handled correctly when it happens.
Specifically:
• if we know a load doesn't match an earlier store (and the same in reverse, a store doesn't match a later load) then we don't have to worry about this business of making sure data goes to the correct additional slots, we can just do the obvious thing – for a store, hold onto the store data in the store slot; for a load, read the data from the L1 cache.
So we'd like to know about a load/store address collision ASAP.

split store address from store data

So this leads to the first, easiest part of a a better solution: realize that a store may be able to resolve its *address* long before its store data is known.

If you can attach an address to a store slot, OK, any load that reads from that address will have to delay until the data is available. But all the other loads (and this is most of them) with different addresses can go ahead, all they care about is that *an earlier slot is not relevant to them*, and won't acquire an address *that becomes relevant to them*.
You are probably well aware that, for example, x86 splits stores into two operations, one corresponding to the store address, the other to the store data, and each is separately scheduled. Every performance CPU now does the same thing conceptually, though they may implement it rather differently.

predict load/store dependency

Alternatively, or in addition to, separating the store address from the store data, you can build a **predictor** that predicts whether a load is likely to depend on a recent earlier store. Then what you can do is run step 13 as before, but if step 14 shows missing store addresses, consult the predictor and either wait or just assume there won't be a relevant store that later occurs, and go ahead.
This is, of course, a new form of speculation ("load/store" or "address alias" speculation) but fortunately it can be handled by our existing machinery like the ROB. What's necessary is to validate that when every relevant earlier store slot has its address eventually filled in, that address did not collide with the load that we (LS-speculatively) executed and whose data we (LS-speculatively) pulled in from the L1 cache.

If there *is* an address match, we
- mark the load in the ROB as misspeculated,
- update the LS predictor,
- flush the bad load and every subsequent instruction,
- restore state, and start again; much like a branch misprediction.

(Note that I was a little quick above – if we're clever, we can actually compare the later stored data to the data

used by the load from the cache. If they are the same [and you'd be amazed how often they are, many many stores are the same thing repeatedly stored to the same address] then no harm no foul! No need to create drama and go through a recovery process...
This is possible in theory. I'm unaware of whether any cores actually implement this optimization.)

It is a fortunate fact that LS dependence prediction is surprisingly easy, and predictors are remarkably accurate, way above 99% for most code most of the time; it's a much easier problem than branch prediction.

Let's now consider a basic LSDP (load store dependence predictor) in more detail; how it might work to see how it might be improved.
The basic flow is as we have described
- we have an early store A, and a later load B
- store A does not execute for a while because its store address is not available
- load B executes optimistically
- when store A actually executes, it notes the overlap of its address with load B and raises an alert
So what the predictor should warn us about is that load B depends on store A. But how exactly will it do this?
The prediction cannot be based on the store and load addresses because we don't know those! Specifically, if store address A had not been delayed (ie unknown), we would not have had to speculate on whether its address did or did not match load B!
How about we identify store A and load B by their PCs, PCa and PCb? Obviously this may not be perfect, but speculation is about what usually works, not what's perfect, and using PCs to track probably colliding load/store pairs works very well.
So we imagine the LSDP as essentially a table of (storePCa collides with loadPCb) pairs.

Now how would we use such a table?
The simplest idea is at some point in the pipeline we check the PC of each load against the table. In event of a match we mark this load as "pessimistic" and do not allow it to execute until every earlier store address is known.
That will work, and is already a good start, but now let's consider various sub-optimal aspects of this implementation.

- as described this mechanism does not allow for multiple loads that depend on the same store. This could be dumb code (multiple successive loads from the same address) or reasonable code (store a 4-element SIMD vector, then successively load each of the four elements as scalars for subsequent processing). This is an easy fix that you can imagine for yourself.
- there's also the possible reverse problem, where a load depends on multiple stores (so store four successive scalar values, then load them as a SIMD vector)
- we are being too pessimistic, waiting for every earlier store to execute, not just the store(s) that affect this particular load
- as described, once a load/store pair is in the predictor, it's there forever! There's no mechanism for

removing entries from the predictor once they become irrelevant.
- a silly (but necessary) concern is that essentially this mechanism as described above, specifically the recording of load/store PC pairs, is the subject of a patent by the very litigious WARF (University of Wisconsin).

splitting the single load-store queue into separate load and store queues

We now know why
- an LSQ exists,
- why it's (conceptually treated as) a single queue,
- the issues around why you want time ordering of the (load and store) addresses, and
- what to do about missing addresses.

So consider resource issues:
If we want to hold up to, say, ~600 pending speculative instructions (size of History File), and we expect ~half of them to be loads or stores, we need ~300 LSQ slots. When we run out of LSQ slots, like all resource allocation under the traditional model, Rename stalls until the rest of the pipeline makes some progress, so eventually a load or store reaches the head of the ROB, is retired, and its slot is freed.

But a large LSQ is not cheap!The problem is not so much the area of the queue, to hold all the addresses and store data; the problem is that
- every time a load or store executes, it has to compare its address against so many other pending addresses, and the logic structure that does that (called a CAM) is power hungry, growing worse as there are more and more addresses to compare.
- and with a single queue, we have to waste power checking every address against every other, even though stores only want to compare against load addresses, and vice versa.

What can we do to improve life?
The original model I described, before load/store dependence speculation, is essentially what we saw in something like the MIPS10000 in late 1990s.
The x86 equivalent at the time only required a store queue because all loads and stores were executed *in order* relative to each other. (Meaning loads occasionally pulled their data out of the store queue, but there was never a situation where store queue entries had not yet been filled in, or a case where a store could generate data that should have been read by an earlier load).

IBM POWER4 (2002) takes us a long way to the full model I have described here. They have out of order load and store execution, and they have prediction of whether loads will alias with earlier stores. What IBM do at this point, and what seems to be the standard going forward, is to split the model I have described into two parts.
Imagine separate load and store queues (now possibly of different lengths) for which each queue entry has a few bits that we can call an "age".

Rename allocates ages as a single (continually increasing) stream that's common to loads and stores, but stores get store buffers from the SQ, loads get load buffers from the LQ.
Each buffer, at allocation time, has its age stored in the age slot. We still require that loads scan the store queue looking for earlier stores with a matching address, but they now find where to start the scan by finding an appropriate age marker; likewise for stores looking in the load queue.
You've
- given up a small amount of the ordering info in the single LSQ,
- but you have less work
- also each queue (and so each pool of addresses) is smaller, even though the total number of Load and Store Queue slots together can be larger.
You can now also start to optimize each queue separately for its particular tasks, giving it a slightly different logic structure.

Non-standard ideas for improving the LSQ

use the store queue as a L0 data cache

At this point you can make the following observation:
- on every load you have to pay the price for accessing the SQ (to match the load address with possible store addresses).
- And if you hit a load in the LSQ you don't have to pay the price of accessing L1 (because the store data is in the Store Queue entry)
- Therefore you might as well try to store as much in the LSQ as possible!
This leads you to try to structure things so that after stores retire, and even after the data is written out to cache, you try to hold onto every store queue entry (marked as valid, but free) for as long as possible, so that you can maximize the number of load hits to the storage queue.

This idea (and the mechanics of how to implement it) are discussed here: (2019) https://www.diva-portal.org/smash/get/diva2:1316126/FULLTEXT02 *Filter Caching for Free: The Untapped Potential of the Store-Buffer*.
(This works surprisingly well. On Skylake sized machines, you can get 30 to 50% hit rate, saving around 15% of the energy used by the load/store/L1D subsystem!)
Is Apple doing this? As a pure energy savings measure, with no performance impact, it's hard to test. I have seen nothing in the patent record, but if there's nothing to patent beyond the basic idea...

two level LSQ

This is nice, but we still want larger load/store queues, and they are still expensive!
Haithim Akkary, in the paper I have already referenced a few times, suggests a two-level LSQ. This is an easy'ish boost, but uninteresting and probably far from optimal.

virtual LSQ

More interesting is a step IBM took a few year after POWER4, with POWER7 in 2011, which is to "virtualize" much of the load/store queue.

Just as we discussed with registers, the standard LSQ model uses early allocation (at Rename) and late release (at Retire) of LSQ entries.
This means that much of the time the LSQ entry is marked as not free, but is not being used productively;
- much of the time it's empty waiting until the load or store executes, or
- it's long after everything relevant to the load or store executed, but the instruction is being blocked by something at the head of ROB and until then the LSQ entry cannot be released.
And as before, this is because in the most obvious LSQ model, or even, the fancier POWER4 split LQ and SQ model, you need allocation *at Rename* to attach *ordering requirements* to the LSQ entries.

So how to work around this?
What I am calling age tags are still allocated at Rename, in program order, across loads and stores in increasing order.
But the actual load and store buffers are not allocated until the instructions are ready to execute as they leave the Scheduling Queue.
This means that you maintain can ordering across a large pool of loads and stores (use more bits for the age tag) while using a smaller pool of Load and Store Queue entries (and thus addresses you have to check). Most of the entries in the virtual Load Store Queue (ie load/store instructions ordered by age tag) will spend much of their time in the Scheduling Queue, not in the Load and Store Queues.

As with virtual registers, if you temporarily run out of physical backing store (load or store queue slots) the addresses will pile up in the Load Store Scheduler, not in Rename, so more of the machine can make progress (even while load and/or store are temporarily blocked, other instructions can still move on to the integer and FP pipelines).

You can see some of the issues involved in a design like this (along with a more careful explanation of the various scans of the load store queue for various purposes) here: (2007) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.3533&rep=rep1&type=pdf *Late-Binding: Enabling Unordered Load-Store Queues*.

Apple definitely seem to be using a virtual LSQ (evidence in the experiments section below). Different patents suggest that what Apple variously calls the GNUM (retirement group number), or the RNUM (reorder buffer number), or the LNUM (load queue? number) are used as the ordering property for load/stores, fulfilling the role that I called "age bits" when describing POWER4.

Late resource allocation is a nice resource amplifier, with the flip side of the idea being early resource release. For example if there is no older store ahead of a load with an unknown address, then there is

no possibility of the load having been mispredicted as not matching a store, and so having loaded stale data from the cache. In other words, under these circumstances, there is no need to continue to hold onto the Load Queue entry.
Again the experiments (with some patent validation) show that Apple is using early release of both Load and Store Queue entries.

Apple's implementation of load store dependency

2009 (load referenced by instruction count relative to store)

We start with a very basic 2009 https://patents.google.com/patent/US20100205384A1 *Store hit load predictor*. (If you look at the inventor name on that patent and think *Apocalypse Now* or, even better, *Zeroville*, you are clearly way too much of a movie geek!)

In this early version, noteworthy points are that
- stores are tracked by PC.
- problematic loads are tracked by *an offset* (number of instruction in the instruction steam) relative to the problematic store.
This is clearly an end-run around the patent, but it works!
- the scheme can track a load dependent on multiple stores, but not a store that affects multiple loads.

The essential idea is
- when a *store* (not a load) passes through Mapping, its address is compared with the store addresses in the predictor.
- if there is a match, the associated count in the predictor's entry for that store is read.
- we now start counting instructions and if we see a load <count> instructions after the store, then that's a load that needs to be handled carefully.
- we know the SCH# of the earlier store, and we add that SCH# to the dependency list of the load.
Isn't that cool? Another lovely case enabled by the dependency bitvector. So now the load delays execution not just until its address register(s) are available, but also until the store has executed.

So this handles the legal issue, and the matter of waiting on the exact problematic store.
How does this handle the other issues we raised?

- the LSDP table is a CAM, so it's fully associative, and it's allowed to hold multiple entries with the same store PC but different load offset counters.
This is how multiple loads dependent on the same store are handled.
When the PC enters the CAM, multiple entries are triggered, and multiple counters started with the appropriate load offset counts.

- Alternatively the table can be populated by multiple store PCs each with a different count that ulti-

mately references the same load.
In other words, the table can store a load that depends on two (or more) stores. And the bitvector mechanism can capture those multiple stores as multiple dependencies.

This use of the bitvector mechanism is no small thing. Simply having a predictor that tells us “this load probably depends on a recent store, be careful” is better than nothing, but how do you act on that information?
The bitvector gives a very precise answer – create a dependency on the SCH# of the earlier store, so that the normal scheduling of instructions prevents earlier execution.
Without this mechanism you’re forced to resort to messier, less precise, ad hoc solutions, for example *Store Sets* (1998) http://people.csail.mit.edu/emer/papers/1998.06.isca.storesets.pdf *Memory Dependence Prediction using Store Sets.*

- The patent points out this possibility of a load depending on multiple prior stores; but a practical matter will be how many of these "instruction N from now" counters is the Mapping stage tracking at any given time? Presumably simulations will show a number of dependencies that make sense, and for more complicated situations (16 byte loads followed by a single SIMD load?) we revert to marking a load as pessimistic and just waiting for all earlier stores to execute.
So one could imagine something like the described mechanism handling anything up to 4 simultaneous "pending problematic loads" (ie four counters) all of which could eventually resolve to the exact same load (with four store dependencies) and any load that somehow fails to be captured under these conditions (either too many dependencies; or just too many simultaneous load/store dependencies happening in this narrow stretch of code) being relegated to a separate table of "pessimistic loads", based purely on load PC with no test of store PC. At least that's how I would do it.

- One sub-optimal aspect of this scheme is updating the table. The CAM is treated as a FIFO, so that entries are aged out purely based on new entries entering, with no tracking as to whether any particular entry is still useful or anything like that.
This is a reasonable first start and I'm guessing that CAM is small, so adequate as a first attempt.

- A second sub-optimal aspect (not essential, but part of Apple’s 2009 implementation) is that the store PC held in the CAM is only a hash (Apple suggests about the ten low order address bits) of the PC. This means that, for example, there will be occasional non-problematic stores that match in the LSDP predictor and start a counter. If that counter expires matching a load, then that load will be delayed. Presumably the combination of both these events (store matching low address, then load matching the count) is not too common, but it’s not ideal that it can happen.

I raise this issue because keep it in mind when we later look at branch predictors.
The LSDP (as of 2009) is a CAM, so it’s a fully-associative cache, but the cache entries are based on a hash of the store PC.
Various branch predictor structures are N-way set associative (rather than fully associative) and

indexed by a hash of the PC, but the entries in the indexed set are compared with a tag comprising the full PC. In other words the branch scheme is
- less flexible (it cannot cope with more than N entries that want to match a single index)
- slower (two stages of first tag comparison, then data lookup)
- lower power
- will not alias (match in the address hash of two unrelated cases).

2012 (load and store both referenced by hash of PC and instruction registers)

This is updated in 2012 https://patents.google.com/patent/US20130326198A1 *Load-store dependency predictor pc hashing*. What changes?

We learn that (for this implementation) the size of the LSDP table is 256 entries.
It's still a fully associative CAM, but operated rather differently.
Now the stores are identified not by a hash (ie low bits) of the PC but by a hash of the low bits of the PC and the register(s) used by the instruction. There are many ways to do this, but one could imagine, for example, using the 8 low bits of the PC again, xor'd with the register (5 bits) describing the source data concatenated with the address base register (another 5 bits) in some way.
More of a change is that the loads are identified in the same way; no longer by an offset in the instruction stream, but by a hash of the load PC with some register identifier(s).

The register identifiers, like the PC, are convenient, somewhat variable, data associated with the load very early in the instruction processing. They may marginally contribute to increasing the hash entropy, though honestly I suspect the low PC address bits are already at maximum entropy; but they also serve to make the load and store identifiers not based on the PC, so provide a second way to work around the WARF patent.
The patent gives the exact details of the hash used (which, honestly, to my eyes looks more complicated than necessary, but what do I know? and which is clearly at least to some extent still skewed to ARM32 because it uses a 16-bit aligned PC bit which is only relevant for Thumb.)

The hash (and details in both this and the 2009 patent) refer to load/store micro-op sequences, but I'm ignoring the specifics, though they are a real-world detail. You may think load/store micro-op sequences are an obsolete relic of the ARM32 instruction set, still being used in 2012 but not longer relevant to us. But that's not quite true. While all the standard loads and stores (and even load/store pair) can be treated as a single, not even cracked, instruction, there remain the infamous `LD2..LD4` SIMD load instruction (used to convert AoS data structures to SIMD vectors), and these still pass through a micro-op sequencer.

Without the count-based mechanism of the 2009 patent, the new mechanism is that
- every store, as it flows through Mapping constructs the PC+register identifiers hash,
- probes the LSDP CAM, and
- sets an "armed" bit on every match.
- This armed bit is later cleared when that store is executed.

Meanwhile every load in the same way
- probes its hash against the CAM and
- in the event of a match *that is armed* (ie the store is present, but has not yet executed)
- marks the load as picking up an additional dependency for every store that is matched.

An additional improvement is that a confidence field is now associated with each entry. This is used in two ways
- under low confidence conditions, the entry is retained in the table but stores do not trigger an arming (ie the entry is essentially ignored)
- when a new entry is added, it will be a low confidence entry that is purged to make space.

The patent again raises the issue of a load that may depend on multiple stores but is unclear as to the preferred solution suggesting either
- multiple entries match the load (ie the load picks up multiple dependencies) OR
- an entry is marked as “multimatch” and if that bit is set, behave as I earlier suggested, waiting for all stores to execute before the load is allowed.

Possibly both mechanisms are used, the multiple dependency case for the most common situations (maybe up to two or three dependencies?) and the multimatch case for more than that.
Likewise the mechanism can accommodate stores that affect multiple loads.

The mechanism avoids the counters (which are conceptually cool, but one more complication).
I don’t know if the energy cost of the first scheme (having to decrement a bunch of counters every cycle) is worse than this second scheme (having to perform an additional CAM lookup on every load).
The second scheme is probably more accurate – neither patent exactly mentions it, but they both seem aware that a problem with the first (instruction count) based scheme is if you have conditional branches between the load and the store, which will vary the instruction stream count between instructions...

One issue worth noting is that both stores and loads probe the table as a CAM, meaning that it’s not trivial to convert from a CAM to a lower energy structure like a set associative table. More difficult, and requiring a multi-step lookup, but not impossible (basically one table for stores and ARMing, a second table for loads, and cross-pointers between the two).
And because the lookup is based on data available at Decode it can be be made multi-cycle if necessary. Maybe this has already happened, maybe it’s on the agenda for a future energy saving tweak.

The companion patent 2012 https://patents.google.com/patent/US9128725B2 *Load-store dependency predictor content management* tells us how the confidence field is maintained. One could imagine a few different mechanisms (basically you want to increment confidence every time this entry correctly caused a load to wait, and decrease confidence every time it caused an unnecessary wait).

The precise way Apple do this is to check whence a load that matches an *armed* store ultimately

acquires its data.
If it acquires the data from the store queue, then we increment confidence (we probably need to wait on the store since it was fairly recent).
But if the load acquires the data from the L1D, then we decrement confidence (maybe there was a recent store, but regardless, it is separated enough in time from the load that it's fully processed and present in the cache by the time the load executes).

This is both a better and worse scheme that might first appear, depending on details the patent does not describe.

- It is true that just finding the data in the store queue is not a very reliable indicator that the store was recent enough to matter (there are good reasons to hold store data in the store queue for as long as possible, even after the store is safely executed and even retired).
But the fact that only loads that matched *armed* entries in the LSDP means that the arming is restricting interest to stores that happened recently relative to the load. So better than you might at first expect.

- On the negative side (as described in the patent) this means that loads that match an unarmed entry (maybe just by coincidence in the hash bits; maybe en entry that used to be valid but now the store always happens too early to be relevant) are not aged out.
So the scheme needs to be augmented by some sort of aging scheme to remove obsolete entries from the table. Something like:
- every 10,000 cycles you subtract 1 from every entry's confidence level.
The patent mentions that you need such a scheme, but gives no details.

2016 (LSDP optimized for Replay)

The next evolution is 2016, https://patents.google.com/patent/US10437595B1/ *Load/store dependency predictor optimization for replayed loads*.
This patent confirms the obvious design point, probably already present in the A6, that Apple split store address from store data.

Assume we have a store then at some later time a load that matches the store address, and the LSDP has not been trained on this pair.
What can happen?
- nothing at all. The load is much later than the store, the CPU has no reason to link the two together.
- the load is forced to Replay because it sees the store address in the store queue, but there is not yet any associated store data.
- the load sees no matching address (the store address has not been provided yet), the load goes ahead reading from cache, and later we Flush.
Clearly

- the first case is easy, and clearly
- the third case is used to add an entry to the LSDP.

But what about the second case?

You could argue that Replays are not expensive (true) so just ignore it, and don't use up an LSDP entry. But Apple's choice, as of this patent, is something subtler. The thinking is "we got lucky with this Replay, but clearly we have a situation where there is a store happening close in time to a load from the same address, and we should keep an eye on this".

How should you handle this? Imagine we created a separate Replay Predictor for this case.
As before,
- we store a hash of the store PC,
- a hash of the load PC,
- we arm an entry when the store occurs, and
- if a load PC matches an armed entry, the load PC sets up a dependency.

After all, a Replay is cheap, but it's not free; it would still be better if we didn't issue the load until the store it depends on has executed, rather than have to issue the load twice (once fails because no data, then Replays).

Now if you think about it, everything about this new predictor is basically the same as the existing LSDP! So why not treat them as a single table?
That's basically what Apple does, with only two tweaks.

First
- an entry is marked with a bit that says if it was generated by a Replay vs a Flush.

Secondly
- If the entry is marked as initiated by a Flush, then we want to be really careful about delaying the load. A load delay might cost a cycle or two, but that's much better than a Flush.
So even a very slight belief (weak confidence) that this entry is legitimate will delay the matching load.

- On the other hand, for a Replay the balance of cost and benefit is much closer.

What's wrong with always forcing a dependency on the store? The load can't execute anyway until the store is done!
The problem is that load takes multiple cycles. Once the load begins execution it has to
- calculate the address (add two registers),
- perform a TLB lookup, then
- scan the store queue.

The earliest it can get the value from the store queue is within three cycles, just maybe two cycles.

The world we *want* is that the load picks up the value from the store queue the cycle after it is deposited there;
the world we *get,* if we force dependence, is that the load picks up the value with an additional delay of one, maybe two cycles.

And we can't improve this because we don't have earlier indication of when the store data might be ready than the fact that the store has completed!
In essence, the best we can do is speculate that this load will find its data ready in time. And that's exactly the tradeoff we have here; that is the predictor we have built!

In conclusion:
- The Replay cost is not that high; the cost of forcing a dependency on the store is an additional cycle or two that might have been avoided.
- If we're not confident that the delay is really required (maybe the timing between the store and the load usually works out OK, just occasionally the store data is slightly delayed) then why force a *guaranteed* delay?
- So for Replay entries, we require a rather higher confidence value before we force the load to depend on the store.

Apple give a difference argument for the same end point. The reasoning they give in the patent is that
- Store queue Replays happen occasionally for random reasons that do not repeat.
- If you immediately started acting on a Replay entry stored in the LSDP, then you would delay a lot of loads until the entry ages out, based on these random events.
- By forcing a higher confidence threshold before the Replay entry is acted upon, essentially you require the load to have to replay *twice in succession* before it's considered a real problem case that needs to be respected.
I think both explanations are probably reasonable ways to view why this modification is worth making.

BTW, an additional way to use this Flush vs Reply indicator bit is when you have to replace entries in the LSDP: preferentially replace Replay entries rather than Flush entries. The patent does not mention this, but it's an obvious extension

dealing with non-aligned/overlapping loads and stores

Having a load/store dependency predictor, and the speedup it gives you, is nice, but at some point you still have to actually compare the load and store addresses in the load or store queues, to make sure there is not an overlap. This is not as easy as it might at first seem, when you remember that we could be dealing with a misaligned load that just overlaps in one or two bytes with a different misaligned store...

One way to deal with this is to treat LSQ entries at the granularity of a cache line. A load or store is

recorded in the LSU by the cache line it would occupy, and with a bitmask indicating the bits that the load or store cares about. This uses a little extra storage but is easy enough to implement in terms of simply
- matching cache line addresses,
- and'ing bitmasks from matching cache lines, and
- seeing if there's a 1 in the result.
(What about loads or stores that cross a cache line? Easiest is to convert them into two entries, though other options are possible. That's a waste, but if such cases are common, well, dammit, write better code!)
need to match this to discussion elsewhere of LSQ overlap

Apple's implementation of Replay

what is Replay?

A second advantage of the bitvector dependency scheme is that it allows for Replay dependencies. I've alluded to these many times, but now let's consider them in some detail.

Replays are situations where an instruction could not complete because some detail wasn't ready in time. The standard case is that a load begins execution because its address registers are ready, but the calculated address misses in the TLB. Or it hits in the TLB but misses in the L1 cache. Or it matches an address entry in the Store Queue, but the associated Store Data isn't yet present.
Every one of these cases has the form that something isn't quite ready, but will be soon, so ideally we'd just hold onto the load and try it again in a few cycles.

The problem also extends beyond loads because of the concept of Speculative Scheduling. This is discussed in a paper I mentioned at the start of this document, (2015) https://hal.inria.fr/hal-01193233/-file/ISCA%2715 _Scheduling.pdf *Cost-Effective Speculative Scheduling in High Performance Processors*. The issue is that in high frequency pipelined CPUs, you have to schedule the next cycle of instructions based on a hope that the current round of instructions will complete correctly; you don't have time to wait until the instructions have confirmed successful execution before scheduling the next set of instructions. This means that you might schedule multiple dependent instructions that assume that, at the time they start executing, the data they are expecting from a previous load will be present at some expected location (like the bypass bus). If the load fails, those instructions will still read whatever random data is on the bypass bus, and that's not great!

Replay recovery (early 2016)

Our first concern, once we realize a Replay is necessary (eg the TLB, the cache, the store queue, have indicated that the data expected by the load is not present) is recovering from the invalid data. The easy answer is simply to Flush everything after the relevant Load, wait till the Load data has arrived,

then start again, but obviously that's awful! Slightly better is just flushing the instructions that are *in execution* after the Load miss is discovered, but that's still not great. The classic paper (2004) https://pharm.ece.wisc.edu/papers/hpca2004ikim.pdf *Understanding Scheduling Replay Schemes* discusses the known ideas, some of which inform Apple's scheme.

The patent is (2016) https://patents.google.com/patent/US10514925B1 *Load speculation recovery*. Essential ideas are

- when an instruction dependent on a load begins execution, it is not removed from the scheduling queue. Instead a "timer" attached to the instruction begins a countdown. These instructions are said to be in the "shadow kill window", or sometimes the "load recovery window"
- if the load behaves as expected, the timer is "cancelled" and the instruction removed from the scheduling queue; otherwise
- the instruction remains in the scheduling queue, to be rescheduled when the load replays.

The nice thing about this scheme is that
- it scales to multiple dependencies. If three instructions are waiting on the same load, they will all be cleared from the scheduling queues, or left to Replay, when that load indicates its success or failure
- it is recursive in the sense that an instruction dependent on an instruction dependent on a load inherits the "timer" and will likewise be cleared or left to replay.

The exact details differ from what I've said; specifically the "timer" is a bit that is shifted each cycle, but the idea is as I said.
You should look at the patent details, but it's really very clever, while also being rather simple! In particular it builds on the existing dependency bitvector scheme, and it handles all the complications you can imagine, like an instruction that depends directly on load A and indirectly on instruction B that depends on load C that started a cycle after load A.
It also has the marvelous advantage that it is trivially adapted to value speculation! I have found no patent proving that Apple uses it in this way, but it's such an obvious next step.

Replaying instructions

Given the above design, we see that after a Load fails to acquire the data it expected, it (and all the instructions that depend on it) will still be in the scheduling queue. That's good, it means the Load is ready to immediately re-execute (after which the dependent instructions will be scheduled).
But when should the Load re-execute?
The traditional answer is simply to retry the Load every *n* cycles, which is easy to do but wasteful of both power and performance (every cycle you retry, some other instruction is prevented from using that execution unit).

2006 (extra dependency bits)

(2006) https://patents.google.com/patent/US20080086622A1, *Replay reduction for power saving* first lists a number of different circumstances that might require Replay, then describes adding a few extra dependency bits to the dependency vector for all these different various circumstances.
The relevant bits are set true the first time the instruction fails, so that the instruction will not be re-scheduled [because a "dependency" bit is not resolved] , until the problem clears, at which point the fake dependency is marked as resolved and the instruction is scheduled. (As always, details in the patent).

This is a good start because the instruction does not waste resources retrying until retry makes sense. But it's not perfect because the Scheduling Queue is a small structure, and while a Load sits there waiting on eg a TLB or L1 cache miss, other loads cannot use that Scheduling Queue slot.
An alternative would be to move the Load to some sort of separate queue while it waits (and even, if possible, return or some how reduce the resources it is holding while it waits).

mid 2016 (move Replays into the Load Queue)

Apple has moved partially to this with the second generation (A7..A10) of cores. For example 2016 https://patents.google.com/patent/US10133571B1 *Load-store unit with banked queue* states (as an aside, apart from the main patent idea) that the load queue and store now hold loads/stores that are waiting on either TLB translation or a cache miss.
This means that these loads (and stores) that are waiting on a Replay are cleared out of the LS Scheduling Queue (which is progress!) and now occupy the rather larger Load or Store queues.

But it still leaves instructions dependent on those waiting loads clogging up the other scheduling queues :-(

Note that this also now moves some Scheduling functionality,
- out from the Load Store Scheduling Queue (where the Scheduler waits for dependent physical register values to be available, as required to calculate an address)
- down to the Load Queue (where loads wait on things like TLB values or data to be available in cache, and they are waiting on Replay events).
This is a repeat of what we have seen so often – split a generic structure (in this case the generic scheduling queue) into specialized versions. So now the generic queue depends on other instructions and physical registers, the specialized Load Queue now has as dependencies changes in Replay sources, like TLB entries or cache data becoming available.

The primary concept of the patent is to split the Load Queue (which now supports Replay) and Store Queue into multiple banks.

For the Store Queue, this split is probably a power-saving measure. We start by filling up the first bank in order,

then when it is full, switch to the second bank while Retires drain from the first bank, then we switch back. I expect the hope is that most of the time the Store Queue is not very full, and so only one of these banks needs to be powered on, apart from a small transition time when a few older values in one bank are waiting to retire even as the newer values are being placed in the other bank.

For the Load Queue, the split is driven by the need for Replay. The details are not given (we will see them in the next patent) but essentially each bank is associated with a Load Pipeline, and loads are allocated sequentially across banks so that banks are populated at an equal rate. This means all banks are powered up, and means some degree of co-ordination between them is required to maintain ordering; but it also means they can behave like Scheduling Queues with a fairly easy “choose one item that’s ready and, ideally, the oldest” every cycle for Replay.

2019 (split Load Queue into a Replay optimized queue and an Address validation queue)

The 2016 scheme means the Load Queue (which was supposed to exist and to be optimized for comparing against earlier Store addresses) has taken on an additional Scheduling task. Having one structure perform two tasks is never ideal so, once again, apply the standard design idea: hence 2019 https://patents.google.com/patent/US20200394039A1 *Processor with Multiple Load Queues*. We split the Load Queue into two parts, one that’s a continuation of Scheduling, one that’s the traditional “validate against Store addresses” queue.

- The first queue (the LEQ or the "fun" queue) is devoted to performing whatever still needs to be done to perform a load, and as fast as possible. This queue will hold load instructions in various stages of execution, from the initial address calculation, to TLB lookup, to cache access, to data value returned. Instructions in this queue can be Replayed if necessary, and can engage in various speculative shenanigans for the sake of speed.

This queue inherits the structure described above, one bank per Load Pipe. Each LEQ bank can be thought of as an extension of the associated Scheduling Queue. Both attempt to schedule an instruction each cycle; of course usually the Scheduling Queue will win because Replays should be rare, but when a Replay is ready to execute it is the preferred instruction.

Replay begins, in terms of timing anyway, at the beginning of the Load Pipe with AGU then TLB lookup. This is clearly not ideal, redoing work that has already been performed. It's unclear if the work is re-performed or if the instruction just runs through those steps to match timing, but does not actually waste energy on a second Address Generation and (especially) TLB lookup. Matching the timing is more or less essential to ensure that Replay does not collide with normally scheduled Loads – if the two timings were not matched then you could have things like
+ in cycle N the Scheduler fires off a Load, which spends a cycle in AGU and TLB
+ in cycle N+1 Replay fires off a load that bypasses AGU and TLB, and so
+ both loads arrive at the next stage (simultaneous lookup of matches in the Store Queue and the L1D) and so collide.

One could imagine a few ways to handle this, but Replays should be rare, so it may not be worth it. On the other hand, a simple bypass that prevents the Address Generation and TLB lookup (while still enforcing the timing) should be feasible.

Interestingly in this new scheme (and probably also the 2016 scheme) we no longer use the dependency bitvector for this Replay Scheduling. Replay is a more specialized Scheduling because each Load only has one dependency (a specific TLB entry, or cache line or whatever) so we can switch to a simpler Scheduler. Each Load has associated with it one ID that describes the event of interest, and that event is broadcast over some bus when it occurs, and checked against every LEQ entry, with matches marked as Ready. This should remind you of a reversion to the original tag-based Tomasulo algorithm!

- The second, larger queue (the LRQ or "boring responsible parent" queue) is devoted to ensuring correct behavior. This holds onto loads even after a data value has been returned, on the off-chance that a Load Store Dependency has occurred, and recovery will be needed. It doesn't have to be as low latency as the LEQ, so it can be substantially larger, or even use low power, slower technology.

The LRQ is, I assume, structured like the Store Queue described above, so it's split into some number of banks (perhaps two or four), that are filled sequentially, with the hope that under most conditions only one or perhaps two of the banks need to be powered up.

Note that Stores can also have Replay conditions (for example the Store TLB lookup can miss).
Store Replays are not performance critical, so perhaps Apple continues to use a single Store Queue structure for both Stores that are waiting for Replay and Stores that are waiting to Retire? (That's certainly what both the 2016 and this 2019 patent suggest).
An alternative could be to create something like the LEQ, holding Stores waiting to Replay. Such an alternative, not being performance critical, could be simpler, for example having only one instantiation, so that only one such Store could be Replayed per cycle.

(And an interesting aside also suggested by the patent is that it describes a little of how Apple allows for early release of the slots for either of these Load queues.)

2019 (convert Flushes to Replays)

Obviously we want to avoid Flushes (Load assumed that it would not match a Store, and acquired stale data from the cache) as much as possible, and a good LSDP is the first line of defence.
But in addition we have (2019) https://patents.google.com/patent/US10983801B2 *Load/store ordering violation management*.

The idea is (at an abstract level, stripped of implementation details) something like
- the somewhat obvious way of checking for load/store dependency is *in the cycle after a load address is calculated*,

BUT
- suppose we move the test instead to the cycle at which the data would be stored into a register/-placed on the bypass bus?

This delay (of maybe 3 cycles under normal conditions, more if there is a cache miss) gives a few more cycles for unresolved store addresses to possibly become resolved, meaning that there'll be a few more cases we catch where we can recover simply by preventing the load result from propagating to the target register and just forcing the load to Replay, rather than Flush!

(Another issue this resolves is that the usual load/store queues explanation seems to operate in virtual address space.
It's rare, but you could have collisions that only appear in physical address space as a result of the load and the store targeting the same physical address via different virtual addresses. This will also be caught by the mechanism described, since this later point at which the load address is checked against the store queue is now after TLB lookup.)
The 2019 patent has some nice timing diagrams that show how this works, if you are interested in the details and still somewhat confused; and it explains exactly how it is done rather than my grossly simplified explanation. (Rather than moving the test to the end of load execution, there's a primary test at the start of load execution, plus an additional test at the end to catch all changes that might have occurred during the intervening few cycles.)

The various patents I've listed suggest that Apple is well aware of how Replays are a lot cheaper than Flushes, and strives both to limit Flushes and to make Replay efficient. This is no small matter. Replay is traditionally considered to result in a *substantial* performance reduction; known to be so for both Alpha and Pentium 4, and believed so for later designs.
This paper, for example, (2019) http://uu.diva-portal.org/smash/get/diva2:1316465/FULLTEXT01.pdf *Minimizing Replay under Way-Prediction* ascribes a 7% slowdown just to Replay caused by Way Misprediction, even ignoring other causes of Replay like cache bank collisions, TLB misses, or cache misses.
But Apple seem to be in a position
- to minimize Replay (no Way Prediction? clever bank handling),
- to perform Minimal Selective Replay (only the precise instructions dependent on the Replaying load have to be replayed, and they will already be present in the Scheduling Queue, not have to be moved back there), and
- to perform it at exactly the right time (by means of the tag-based dependency added to the LEQ entry of the Load that failed)

Something of a companion to the above is (2011) https://patents.google.com/patent/US20120173843A1 *Translation look-aside buffer including hazard state*, although I suspect the details of the scheme have changed since then.
Honestly, I think this patent is completely obsolete, solving a problem that is now solved via the much more sophisticated machinery of the Store Queue, the LSDP, and the various Replay techniques indicated. But if you want to look at it, it considers the situation
- an earlier store begins to execute
- a dependent load begins to execute
No problem, right? We've said that the load simply tests its address against all the addresses in the store queue.

What the patent says is that (for this 2011 design) the store may be in some intermediate pipeline stages between it being finally properly settled in the “Store Queue” (actually a Store Buffer which is a rather less sophisticated concept than a Store Queue) and when the load begins execution; and so the load needs to test its address against these intermediate pipeline stages.

Ultimately Apple has a way to implement Speculative Scheduling efficiently because the following elements can be used together
- instructions waiting in a Scheduling Queue can read a result off the bypass bus before it is written to a register
- instructions are held in the Scheduling Queue (and results are held from being written to a register) until an instruction has completed
- instructions are scheduled dependent on a previous instruction (a SCH#) rather than dependent on the register that instruction produces.
That seems like a technicality but what it means is suppose I have a load that feeds a result to add which feed the result to a mul.
I can speculatively issue the add to pick up its input from the bypass bus, from the load; then the mul to do the same from the add. This chaining works even when I don’t write back the intermediate values to a register.
If the dependencies were based on registers, I would have to be flipping back and forth between marking a register as valid (so that an instruction dependent on it can schedule) then marking it invalid (because its data is incorrect because of Replay). Once again carefully considering the different aspects of an issue pays off:
+ for SCHEDULING I want to depend on prior instructions. If a prior instruction is marked as “executed” that means I can now Schedule (picking up the result of that instruction off the bypass bus) but this is idempotent. If I learn the data I pulled off the bypass bus was invalid (Replay) I
= don’t mark my result register as valid, not yet
= I don’t mark my instruction slot in the Scheduling Queue as completed, not yet
= meaning I can execute again, just as soon as the SCH# I am waiting on (eg the previous load) once again marks that it has issued.
+ for DATA PERSISTENCE I want to mark physical registers as having valid contents, but I can’t undo this if I realized I made a mistake.
= So I can’t mark physical registers as valid until I know their data is trustworthy.
= So I can’t use flipping a physical register to valid as a Scheduling signal, unless I give up Speculative Scheduling. (Or use a much more heavyweight Replay which involves restoring a lot more state at Replay.)

Because
- all the instructions stay in their Scheduling Queues AND
- no results are written to a register (more precisely, random junk may be written, but the register will not be marked as valid)
until the load is valid, Replay is not that expensive. It requires redoing every instruction along a dependency chain a second time, but only one more time, but no Flushing.

Back to LSQ measurements

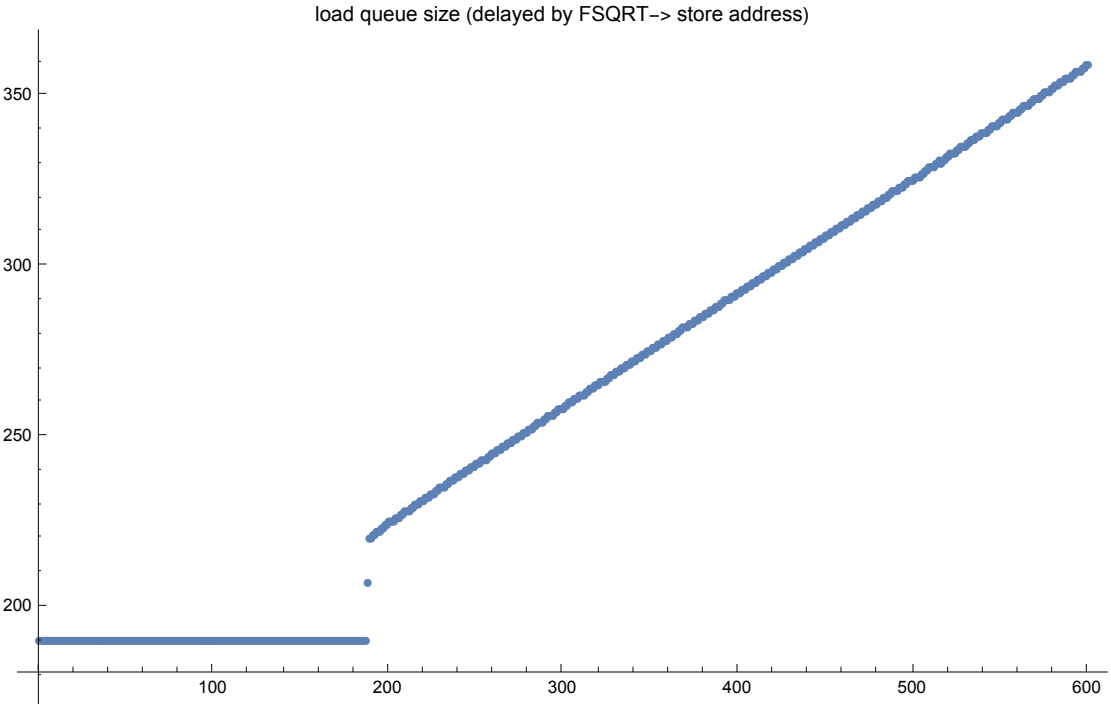
So recall where we started. We hoped to discover the size of the load and store queues, and instead we found what looked like sizes that varied substantially given different test probes.
So, given our new understanding, can we create a cleaner example of this issue, that we see what looks like a ~125/100 sized LSQ under one test but not another?

Yet a third different value for the “size” of the LSQ

Loads have to remain in the LSQ as long as they are in some way speculative. So imagine the following construct:
• long chain of `FSQRT`s to act as a delay
• convert the result of the last `FSQRT` to an integer using `FCVTAS x0, d1`
• store a result using that integer as an index, ie `STR x5, [x3, x0]`
• lots of loads

Under these conditions the loads will be scheduled and will, in fact, even execute speculatively (under the assumption that their load addresses do not collide with the *unknown* store address) but they cannot complete until the store address is in fact known. This gives us a much cleaner result:

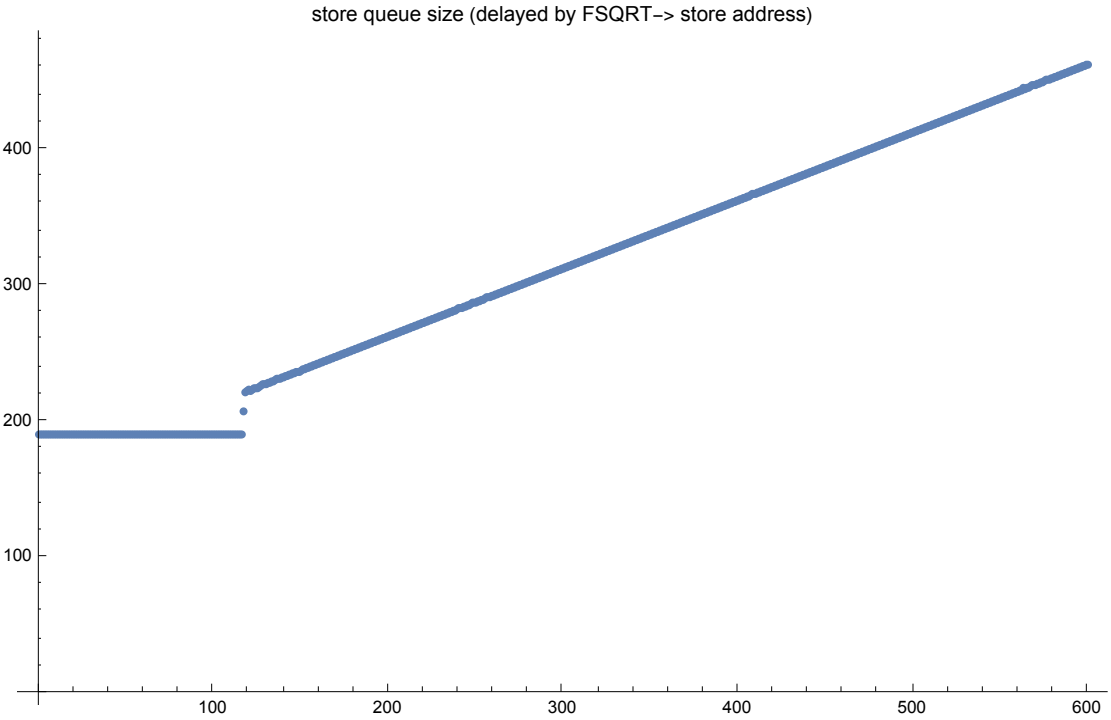
Out[162]=



We see a clear jump at N=188 (load queue can hold 188 entries) and this matches Dougall’s results achieved using somewhat the same idea (but with speculation from a branch dependent on the FSQRT chain, rather than on memory-aliasing dependent speculation).

What about stores?

Out[164]=



Again we see a nice clean jump, giving us ~118 store queue slots, again matching Dougall's result.
(You might wonder why a store would force subsequent stores to remain speculative. Surely even if the second store address matches the first, the second store will simply overwrite the first, so who cares?
In the generic case there could still be a problem because of alignment – if one of the two stores is not aligned, then it would be possible for a matching-width store to overwrite some bytes but not others...

If we force the alignments to match, I think that in principle the ARM model allows this to become non-speculative, but Apple doesn't catch that and special case it. This probably makes sense; they already have most of the win available from a good LSDP, and trying to optimize the very special case of this test is probably of little real world value.)

Explanations

So: We have seen three different results for different attempts to measure the size of the load and store queues:
- the most naive test (uses `FSQRT` delay) gives an apparent size of ~330 for both
- using a load miss to DRAM delay gives apparent sizes of ~125 (L) and ~100 (S)
- using an `FSQRT` delay generating a delayed store address gives apparent sizes of ~188 (L) and ~118 (S)

Why these various different results?
What appears to be happening is that Apple is using a virtual load-store queue. In other words
- while instructions are still in-order at the Map/Rename stage, load/stores are given a sequential ID

that describes their relative ordering, but they are not allocated a load/store queue slot.
- at issue time (ie at the start of execution, when the address is about to be calculated) the load/store queue slot(s) (two in the case of loads, for the LEQ and the LRQ) are allocated.

As counter evidence I will mention (2019) https://patents.google.com/patent/US20200394039A1 *Processor with Multiple Load Queues* which is very clear that
- entries in the LEQ are allocated at issue
- entries in the LRQ and STQ *could be* allocated at issue
- LNUMs (that track relative ordering) are allocated at Rename

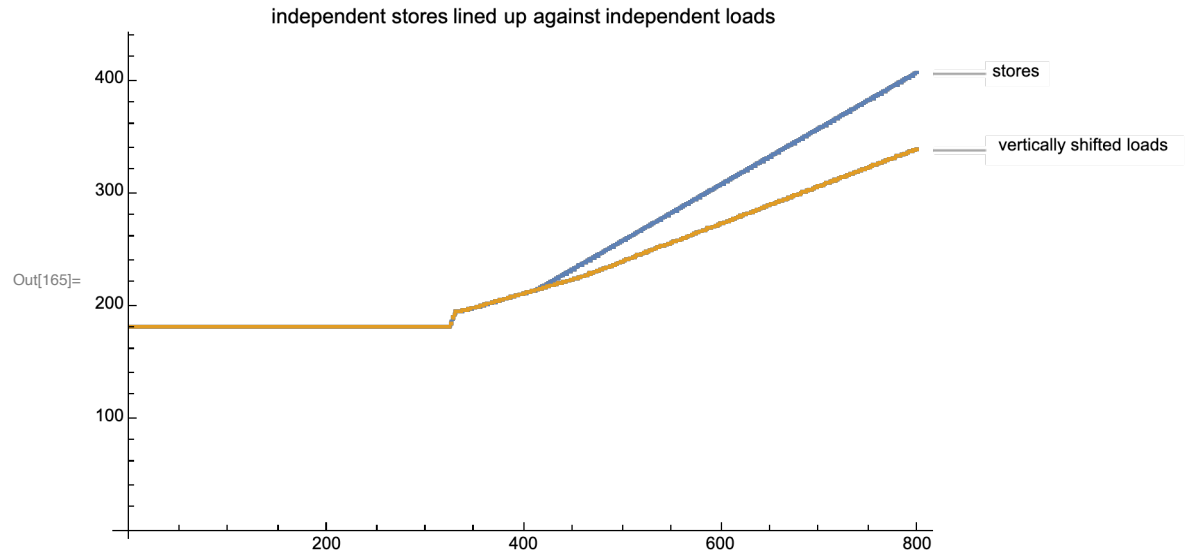
The most reasonable analysis is that as of the 2019 patent Apple was allocating LRQ and STQ entries at Rename, but was primed to switch this at any time, and it seems that by the A14/M1 that time had come.

We will first discuss the load cases, then see if there is anything different in the store case.

early load queue deallocation (simple FSQRT delay)

The easiest case is the first case, a simple `FSQRT` delay

Out[165]=



The salient issue in that case is that there is no question of load/store aliasing because there is nothing in the store queue against which an alias could occur!
Thus, while the loads occupy resources in the ROB, they don't have to hold onto their load queue slots (because the only reason for those slots was to ensure that older stores that have not yet completed do the right thing relative to younger loads, and there are no older stores).
More generally, in principle, once there are no older store that could, in any way, affect a load that load could release its load queue slot.

We see that Apple appears to be doing this, and as confirmation there is, once again, a patent: (2013)

https://patents.google.com/patent/US9535695B2/ *Completing load and store instructions in a weakly-ordered memory model*.
In other words, what we are seeing is proof of early resource deallocation for the case where the resource is Load Queue (ie LRQ) slots.

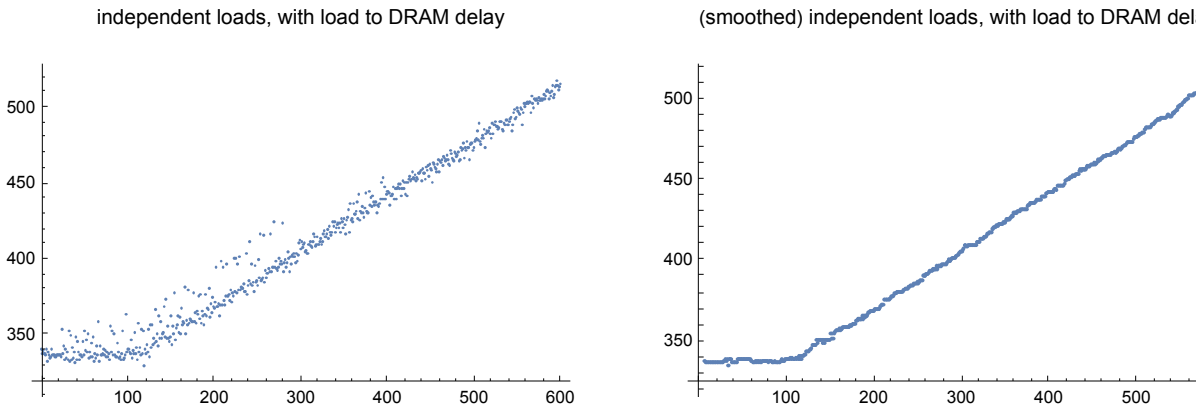
The resource limitation that causes the jump at around ~330 is something we have already discussed, that the ROB consists of ~330 rows, each row can hold 7 instructions, but only one "failable" instruction, and loads/stores (and branches) count as failable instructions.

We'll discuss the other interesting feature of the curve later, once the "sizes" of the Load and Store Queues is settled.

the size of the physical load queue (load to DRAM delay)

Now what about the case of the load-miss-to-DRAM based delay? ie this diagram:

Out[166]=



The most important thing to realize in this case is that the metronome that is determining the base ~330 cycle delay is a series of chained loads.
load A → load B → load C. If each load (to some random location that's hopefully never in cache) can execute immediately then (give or take a whole lot of noise in the response time of DRAM and the NoC) the time per loop iteration will be ~330 cycles.
But if something delays when load B can start executing, then the time per loop iteration will be more than ~330 cycles.
So what we see is that if we have load A (load to DRAM) followed by N local loads (loads to cache) there is no delay as long as N<~125. Once N>~125 we see a delay. So these intermediate 125 Loads are using up some resources that has to be freed before load B can begin execution.
What is that resource? It's not ROB related (that's N=~330, or HF related, ~630, or physical register file related, ~380). It is in fact "genuine", not virtual, Load Queue slots. load B cannot start executing until it has a Load Queue slot.

late load queue allocation (FSQRT delay with ambiguous store address)

Compare this with with our final case, the `FSQRT` delay generating an unknown address for a store, as in the nice clean plots that begin this section.
- while the head of ROB is blocked by multiple `FSQRT`s,
- loads (or stores) are fetched, renamed (ie given *virtual* LSQ resources) and send down the load store pipeline.
- They issue, at which point they are allocated a *real* load (or store) queue slot.
- Eventually those slots run out and the loads (or stores) pile up, first in the 48 LS Scheduling Queue entries, then in the 10 LS Dispatch Buffer entries.
- When those 58 entries are full (along with the genuine ~130 load queue entries, then Rename stalls and we get the classic jump, at N=58+130=188 for loads, and at N=58+60=118 for stores.

If you haven't spent much time with more traditional CPUs, this may seem obvious. But it's not!
A traditional CPU would stall loads and stores at Rename when it was not possible to allocate a Load or Store Queue slot. Hence it would stall at ~130 loads, or ~60 stores.
A CPU with a virtualized LSQ at Rename only has to allocate an age tag. The loads/stores can then proceed into the Dispatch Pool and Scheduling Queue, and can keep doing so even when all the LSQ slots are full, up to the point where the Dispatch Pool and Scheduling Queue slots are also full.
Hence late allocation allows us to enqueue 58 more loads or stores than traditional allocation before we have to stall. We get about 50% larger effective load queue size, about 100% larger effective store queue size, at the cost of a slightly more complex algorithm.

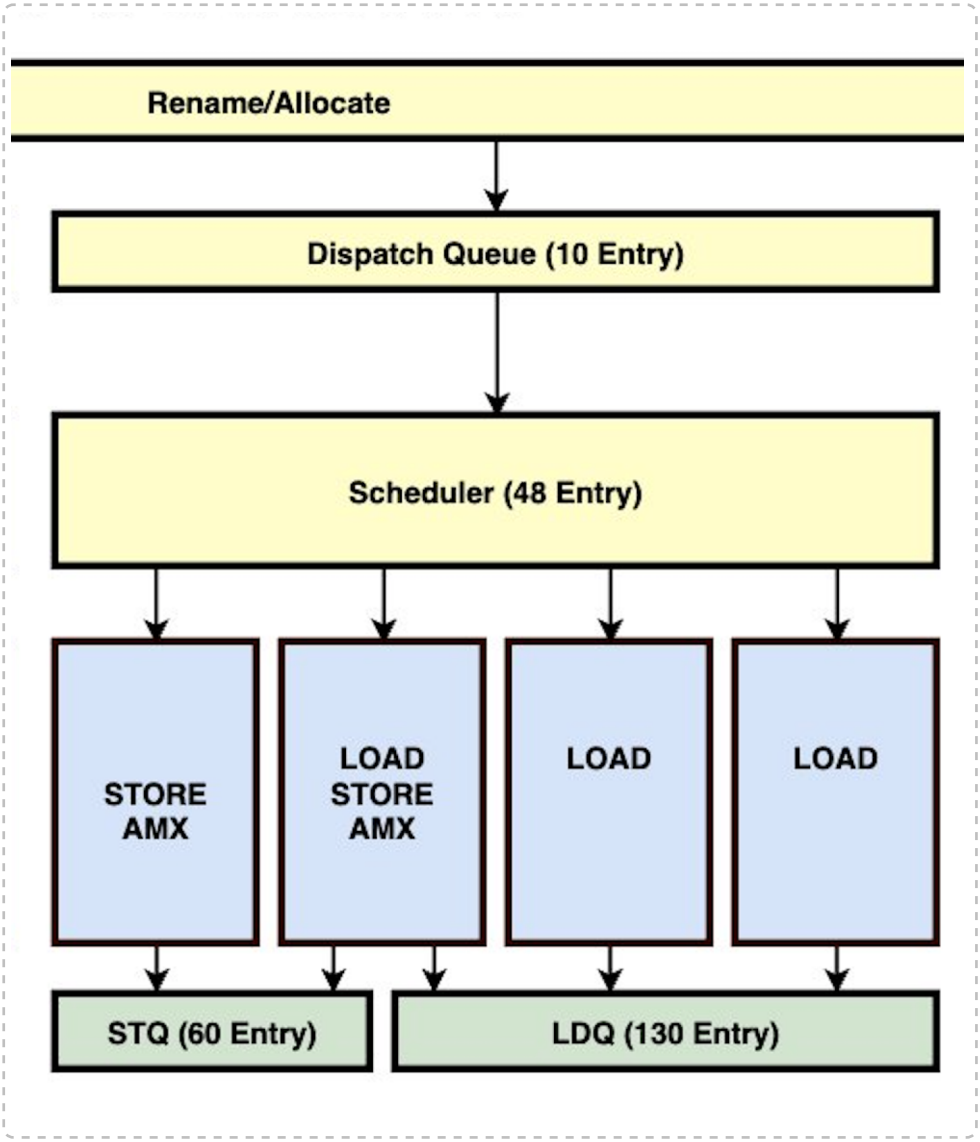
The difference between this case and the DRAM case is that
- for the DRAM second loop iteration to begin, the loop delay (the load B) has to proceed past the earlier stages of instructions storage (in the Scheduling Queues) to begin execution with possession of a genuine Load Queue entry
- for the `FSQRT`+Store second loop iteration to begin, the FSQRTs need to get past Rename to the Floating Point Scheduler, but they don't care if 58 extra Loads have been dumped into the Load Store Dispatch and Scheduling queue and are blocked there.

So what we see is that the M1 has a pool of ~130 genuine Load Queue entries (ie LRQ entries), but for most code patterns this pool is amplified by both
- early deallocation of Load Queue entries (if there are no ambiguous store addresses remaining ahead of a Load, that Load no longer needs to hang around in the LRQ)
- late allocation of Load Queue entries (so that excess loads, that might not be able to execute because all the physical Load Queue entries are busy, can still be shifted out of Rename into storage in the Load Store Dispatch Buffer and Scheduling Queues. Doing this means that even though they can't execute until a Load Queue entry becomes available, they will not block instructions behind of a different class, like the `FSQRT` delay instructions.

structure and evolution of the load-store Scheduling Queues (bonus discovery!)

What about the regions between about 330 and about 410 where we get what looks like 4 loads/cycle, not 3?
That's a reflection of the Dispatch buffer, and we saw a similar situation when we first discussed Dispatch Buffers. Before proceeding, you should also refamiliarize yourself with paired scheduling queues.

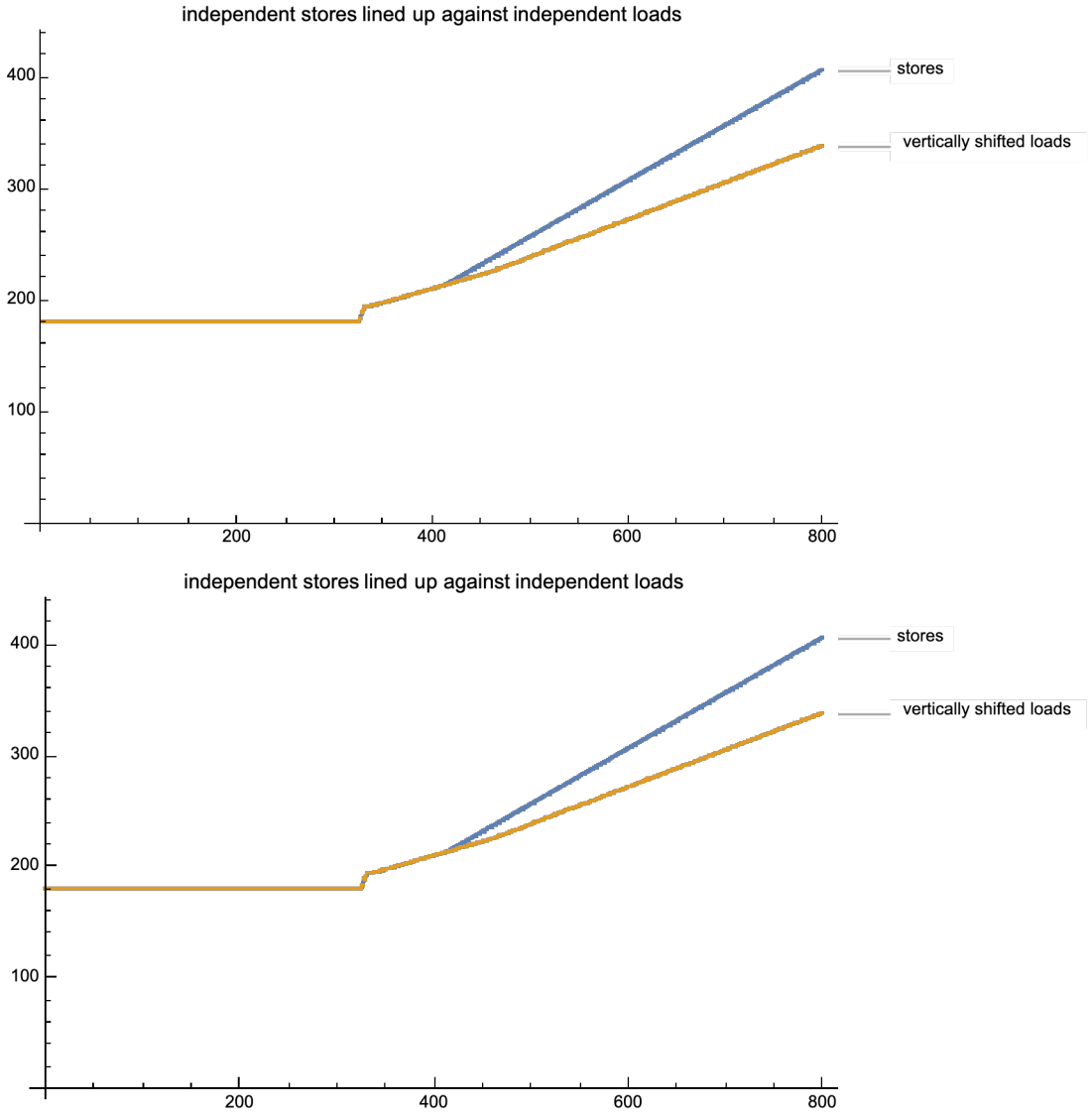
So recall the relevant part of Dougall's diagram:



This shows a single Dispatch Buffer holding 10 entries (probably correct) feeding what looks like a single, surprising large, Scheduling Queue (incorrect).
I suggest that the Scheduling Queue is split into 4 queue of 12 entries, and so somewhat like the integer and FP setups.
Below is the justification for the argument (which I don't claim to be definitive, but it's the best explana-

tion I can give for the phenomenon we see of the “4 ops/cycle for load or store, in the range of N from about 330 to 420” in the graph we have already seen:

Out[167]=



Let’s start by assuming the diagram is correct.
- Load/Store has a Dispatch Buffer of size about 10 instructions.
- This feeds into a single Scheduling Queue that feeds four Execution Units.
- The Dispatch Buffer, as usual, can accept 8 instructions per cycle, but can only output four instructions per cycle into the Scheduling Queue.
So suppose N=410. Then at the end of one cycle we have
- 330 loads have completed, and are sitting in the ROB
- 80 loads could not move pass Rename (because no ROB entries were available)

The last `FSQRT` completes, the loads in the ROB Retire, the new cycle starts.

- First thing to happen is that the 80 loads piled up behind Rename have to be handled. At first these are moved into the Dispatch Buffer at 8 per cycle, and moved out of the Dispatch Buffer into the Scheduler Queue at 4 per cycle. But that means a net of 4 in the DIspatch Queue, so that can only be sustained for about two cycles before we drop to a throughput of clearing 4 loads per cycle.
To clear out 80 loads will thus take ~20 cycles.

- As far as the Scheduling Queue is concerned, we are piling up an excess of one load per cycle, (4 in per cycle, from Dispatch; 3 out per cycle from 3 Load units) so we cannot sustain this indefinitely, but we can sustain it for quite a while, certainly for 20 cycles (at which point we have
+ an excess of 10 loads still in the Dispatch Buffer,
+ an excess of ~10 loads in the Scheduling Queue,
+ and have executed 60 loads).
- At that point the 80 Loads in excess of 330 are done, the `FSQRT`s are in Rename, and can propagate to the FPU, start to execute, and start the next delay cycle.

So we see that for this case of 410 loads, the time taken is the initial `FSQRT` delay, plus the time it takes to clear the 80 loads being able to do so at 4/cycle.
So this describes the initial flat region (the `FSQRT` delay), the jump (run out of ~330 ROB failable slots), and the high speed region from 330 to 410 (Scheduling Queue can take in 4/cycle, emit 3/cycle, and has space to sustain this for quite some time as the Scheduling Queue slowly fills up at one excess instruction per cycle).

That all seems plausible, but we are still left with the question of why we revert to 3 loads/cycle at N=~420? If the above explanation is the full story then at that point we should have the Dispatch Buffer full, maybe 12 instructions enqueued in the Scheduler, and space for an additional excess of 36 or so. We should be able to sustain 4-wide till a much higher N!

My guess is that this reflects some structure within what Dougall draws as a monolithic load/store Scheduler Queue of 48 entries.
XXX I don't have time to draw any diagrams, so you will have to make do with text diagrams for now but what I imagine is the following
- this apparently monolithic queue is actually four queues each 12 entries in size
- the Dispatch Buffer is able to feed one instruction per cycle to each queue (just like integer and FP)
- this works out OK (ie it's allowable to enqueue loads into a queue that looks like its attached to the STORE/AMX execution unit) because
- of the patent we described that pairs queues together and allows an instruction in one queue of a pair to execute if the other queue of the pair cannot find a runnable instruction.

If we accept this chain of logic, then what we see matches what we might expect – we can run 4-wide until one of these queue (the one attached to STORE/AMX) is full (because every cycle it is never chose,

the other queue always has a runnable load). And then we run three wide.

A reasonable objection is that this will lead to a drastic reordering of the loads – the loads that were dispatched to the STORE/AMX queue will (if we accept the way the patent described paired queues) always be considered second class, and will be delayed as long as their are loads in the other load queue, meaning a possible indefinitely long delay. Perhaps so, and perhaps Apple are aware of that, but figured it was acceptable? The A12 has two load pipes and 2 store pipes, I can't find data for the A13.

Here's what I imagine as the evolution of the design. (Timing details for what core got when may be slightly incorrect; this is a conceptual model!)
For the A11 we have a fully symmetric design. So we have the Dispatch Buffer feeding 4 identical queue, each 12 in size, arranged like
S0 S1 L2 L3

Now for the A12 we decide we will add paired scheduling queues. How should we pair?
The obvious choice is to pair the two L's together, and the two S's together; it's simple, and instructions from one queue can easily be shifted to the paired pipeline without even having to test the instruction type. The downside is that the total pool of Scheduling storage for Loads remains at only 24 instructions, likewise for Stores. If we have an imbalance of lots of Loads relative to Stores, we can't make use of the Store Scheduling Queue space.

What about if we pair (S0 L2) and (S1 L3)? The win is that now all our scheduling queue space is available to hold either an temporary excess of Loads, or (less likely, but happens sometimes, eg if you're filling a large structure with zeros, an excess of Stores). The downside is that (without substantially rethinking the details of how we choose the the oldest ready-to-execute instruction; and the mechanism of second choice from one queue feeding the other queue) we can get cases of imbalance. If the S queue is all filled with loads, and the L queue always finds a runnable load, then the S queue will stay blocked that way until something changes (eg a dependency eventually means the paired L queue cannot find a runnable Load); this is the imbalance we saw above.

So which of these two options is overall better is a question for the simulator, but I would not be surprised to see that the LS pairing is better. For normal code (with a mix of loads and stores, especially if Dispatch tries to place stores in an S queue, and only places a load if no store is available in the Dispatch Buffer) then it's probably rarely an issue. And for code that consists of a long stream of loads or stores and nothing else, it's not much of an issue the precise order they get executed, so if some loads get stranded in an S scheduling queue for many cycles, well, so what.

So let's assume that's the A12. With A13 we get AMX, and AMX instructions are executed (ie transported to the AMX unit) via the Store pipeline. So now we have (SA0 L2) and (SA1 L3). This works in the same

way as above, is still essentially balanced, and still gives us the ability to absorb long streams of pure loads, pure stores, and even pure AMX instructions, in a way that can devote all 4x12 Scheduling Queues to that single instruction type if that's all that is seen.

Finally with the A14 some bright engineer notices the following:
Suppose we define a third load pipeline. This means essentially we
- need an AGU (but can reuse a Store AGU)
- need a port into the TLB (details dependent on exactly how the TLB is structured, but reuse of the store port may be feasible)
- need to create a third path to the L1D (details dependent on exactly how the L1D is structured, but reuse of the store path may be feasible
- need to create a third bank for the LEQ
- need to create a third port to test load addresses against address in the Store Queue
In other words a lot of reuse of existing Store machinery is possible.
And because of the pre-existing paired Scheduling Queues, we don't have to do much real work at the Scheduling/Dispatch level.
So we land up, finally, with a structure that looks like
(SA0 L2) (SAL1 L3)
Loads that go into the second pair (either scheduling queue) can be issued as loads, and that part of of mismatch from the A13 has gone away. The only problem is if the SA0 queue is filled with a long run of loads, which become blocked because loads are also being Dispatched into L2, and those L2 loads are always preferentially issued.

At each stage we have given our Load Store unit a reasonable boost in capabilities while also, at each stage, never paying much of a hardware price. The cost is that at each stage we introduced some asymmetry so that while performance is usually boosted quite a bit, there are weird corner cases that will do less well. If I'm correct that every four years or so the design gets a complete clean start redesign, then we may soon revert to a symmetric design with better performance than what we have today, and without the asymmetries.
For example what we also converted SA0 to a load supporting SAL0? Well, do we want to do the full work required to support a fourth load pipeline? Maybe that's not a good use of resources.
Alternatively we could have that SAL0 and SAL2 take turns to feed into a single third Load pipeline (basically if only one of them has a load, they get the pipeline; if both have a load, one gets it and flips a bit so that next time it's the other one's turn). That sees like the sort of quick elegant hack that Apple uses so often, providing most of the benefit of giving a full SAL0 (in particular better load balancing now) at very low additional hardware complexity.

Alternatively, we could change the design so that the Scheduling Queues become virtual! I think this is eminently practical.
Consider how the Scheduling Queues are used. Their job is to hold instructions, test that an instruction

has become runnable, and check ages.
These are the same job regardless of whether the instructions are loads or stores. The details of load or store only matter at insertion (Dispatch preferentially inserts stores into an S Queue, loads into an L queue) and extraction (Issue from an S queue will only extract Stores or AMX instructions). Note also that both Dispatch and Issue are already cross-wired to both of a pair of S and L Scheduling Queues.
So:
- differentiate between physical queues (call them 0 and 1) and logical queues (call them S and L)
- Dispatch and Issue operate as above on the logical queues
- but there's a degree of indirection between the SL and 01 queues.
- So we start off with S tied to 0 and L tied to 1
- Whenever one of the queues becomes full, we swap the indirection at both ends. (A few addition tests are necessary to ensure that, under certain circumstances like once both buffers become full, we don't just keep swapping every cycle!)
- Meaning that, if eg, we were servicing a long run of loads, the 0 queue will become filled with loads that are never extracted, until the swap, at which point the the 1 queue will become filled with loads.

This mechanism allows two nice improvements
- ALL the LS Scheduling Queues can act as buffer during a long run of either pure loads or pure stores; we get a much larger effective buffer before we have to drop to 3-wide Loads or 2-wide Stores as far as the rest of the machine is concerned. Right now the effective excess buffer is 10 (Dispatch)+ 12 (one L queue) or 24 (two S queues). But what I'm suggesting would allow that to expand up to 10+48 for both L and S (and A).
- We do a better job (not perfect, but a lot better) of not holding onto very old instructions without Issuing them until the machine has been forced to stall.

However as it is, we see the design we we see. Rename can feed Dispatch can feed Scheduling 4-wide up to about 90 instructions (N=~420) before queue SA0 becomes full of Loads and never able to get a chance at Issuing a Load.
Note this also explains the Store behavior of 4 wide until N reaches ~420. The logic goes as before. but now every cycle both queue L2 and queue L3 are being filled with Stores (Dispatch of 4 instructions per cycle), while only SA0 and SAL2 are being drained of Stores (2 Stores Issue per cycle). Again at around N=~420 we have both L2 and L3 filled each with 12 stores which have never had a chance to issue, and throughput drops to 2 Stores per cycle.

what about the store queue?

What about the Store versions of these tests? Do they teach us anything additional?

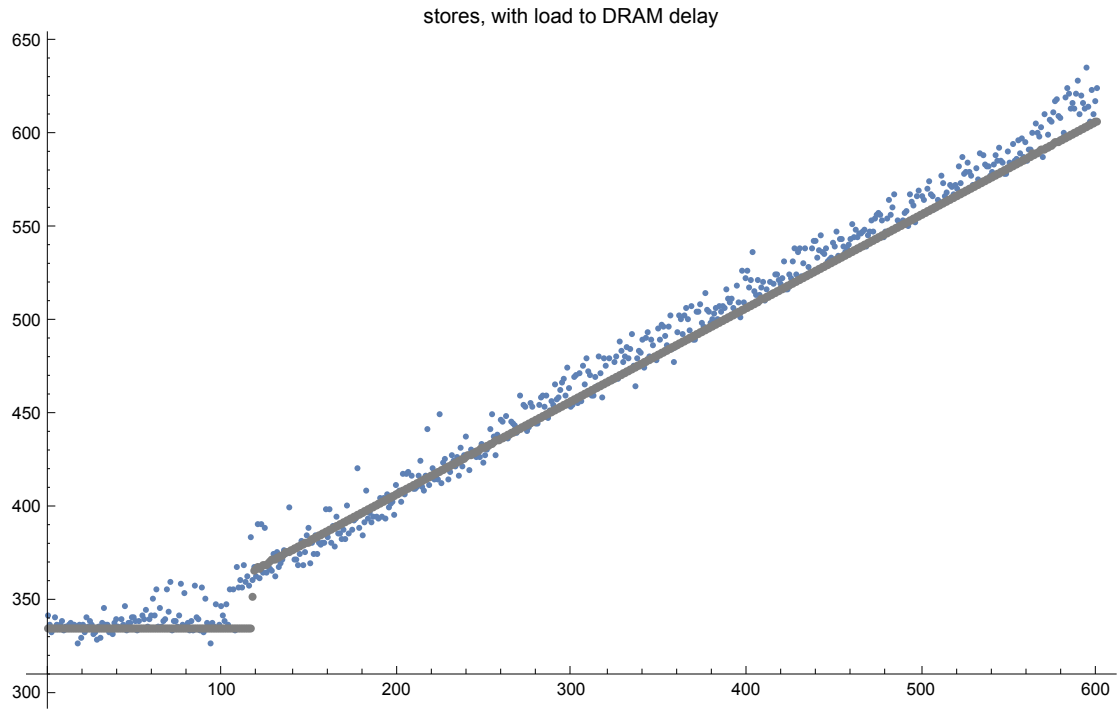
late allocation (miss to DRAM and FSQRT with ambiguous address delays)

Consider first the load-miss-to-DRAM case. That shows a jump at what looks like N=~100. Is that reasonable?

In this case we do not see a delay in how long it takes to start Load B until not only all 60 store queue slots are in use, but also the 58 Dispatch Buffer+Scheduling Queue slots.
Why the difference?
For the load case, the delay loads, load A, then load B, then load C, are fighting for the same Load Queue slots as the local filler loads. It does not help that the delay loads can move past Rename to pile up in the Dispatch Buffer and Scheduling Queue; that doesn't change the fact that the delay loop can't begin the next iteration until load B receives a physical Load Queue slot. ie the load case stalls at *Execute*.
In the store case, the delay loads have no interest in the Store Queue slots, so stall does not happen at Execute; rather now the common resource that both delay and filler store care about is the Dispatch Buffer+Scheduling Queue slots, and it is only when those are filled up that stall occurs, this time at *Rename*. Before that point, as long as a load can get past Rename, there should be a Load Queue slot available, and being Dispatched into a Load/Store Scheduling Queue should allow the delay load to get to it and start execution.

But I slipped a fast one past you in the above! Go through the argument again. The common resource that matters is in fact just the Scheduling Queue slots, not the Dispatch Buffer. As long as there is one space free in one of the four scheduling queues, then load B can be Dispatched into that Scheduling Queue, and can be Issued for execution in the next cycle; no delay. But if load B is placed in the Dispatch Buffer, then it cannot begin execution until a space opens up in a Scheduling Queue. Moving it to the Dispatch Buffer clears Rename and allows other types of instructions (int or FP) to make progress, but it doesn’t mean that the Load can make progress, not until a Scheduling Queue slot opens up.
This, and the fact that movement from the Dispatch Buffer to the Scheduling Queue is somewhat stochastic once the Scheduling Queues and Dispatch Buffer fill up (since the Dispatch Buffer makes no serious attempt to preserve ordering and Dispatch the oldest instruction) explains some of the noise we see in the DRAM Delay case. And it means that the N, number of stores, at which we should expect a jump is in fact 48+60=108, omitting the 10 storage units of the Dispatch Buffer. That looks (by eye, given the noisiness) more or less what we see.

Out[169]=



Certainly it seems reasonable to assume that, like Loads, we see delayed allocation of Store Queue slots, since both probes shows that many stores can progress past Rename even when the Store Queue (of size ~60 entries) is full.

early deallocation (simple FSQRT delay)

Consider this apparently simple case of a delay block of some `FSQRT`s, followed by a number of local stores (to the same simple in-cache address), which only jumps at N=~330? Here we have to tread more carefully.

It's undeniable that

- execution does not seem to be hindered by any resource limitation before the 330 entry ROB limitation
- there are not enough ways (either actual store queue slots, or "virtual slots" by holding store instructions in the LS Dispatch Buffer+Scheduling Queue) to get us close to 330 storage elements

- BUT! we said that stores have to be retained until they are non-speculative; we cannot allow speculative stores past the LSQ storage to make it out to the L1$!

If we equate “non-speculative” with “Retired”, then we have a problem because these stores have not Retired – that’s the whole issue of them still present in the ROB and using up the ~330 available ROB slots they can occupy.

Conceptually one might imagine two ways out of this problem.

One possibility is that we are, in fact, overwriting the same address every cycle. So *in theory* there's no need (once you establish that there are no intervening loads or anything else of interest between two

stores) to care about the older store value, all that matters is the newer value.
Under the right conditions Apple might cull older stores as being idempotent (ie they don't change the state of the machine).
This is easily understood, but I'm unaware of any core that attempts exploit this fact (which, ideally shouldn't happen in most code!)

Can we test this?
I tried changing the storage address so that every store stored to a successive location, and saw no difference.

But that's still not *absolutely* definitive, because how large is the amount of storage attached to each Storage Queue slot? It could be as large as a cache line! Maybe successive stores are aggregated somehow as a single unit in a single Storage Queue slot? Ambitious! But not impossible.

So I separated successive stores by 64 bytes and again, no change, the jump in the curve is at ~330 stores.
The issue is not related to idempotent stores.

The second possibility is that remember the whole point (the only point) of the store queue is to hold onto stores as long as they are speculative.
Not until they retire, not to force some sort of program order, *only* so that they don't escape out to cache until they are non-speculative.
Which means that as soon as we know they are non-speculative, the Store Queue can be released, and the store allowed to proceed to cache!

And this is, in fact, what happens. Essentially for every instruction in the ROB, Apple is tracking two things, both
- when the instruction Completes and
- when the instruction becomes Non-Speculative (ie every earlier instruction that was speculative or could cause a fault of some sort has successfully completed).
At the point that stores become both Completed (ie their TLB lookup has indicated that they are not problematic) and Non-Speculative, the store data can be sent off to L1$, and the Store Queue entry freed for re-use, regardless of how far from retirement the store is.

As usual, there's a patent confirming this hypothesis: (2015) https://patents.google.com/patent/US10228951B1 *Out of order store commit*.

2006 (historical interest, early release of store queue data for a strong memory model)

To understand the full flow of thought, we begin with (2006) https://patents.google.com/patent/US20080086623A1 *Strongly-ordered processor with early store retirement*, a patent from the PA Semi

days, and not especially relevant (because it's about a strongly ordered memory model -- perhaps PA Semi thought they might be forced to make x86 designs?). The patent says, essentially, that it can't think of anything to do about early release of load queue entries, but that there are circumstances under which store queue entries might be early-released. (Well, not exactly. What they seem to have in mind is almost the reverse of our concern. Assume a Store that is Non-Speculative and ready to Retire. At this point it tries to store its data, but the Store misses in L1. The idea seems to be that we will perform the Retire anyway, we will store the Store Data in some Cache buffer (to be merged with the cache line when it is delivered to the L1D), and we will free the LSQ entry.

2013 (early release of non-speculative loads)

This is followed by the more optimistic (2013) https://patents.google.com/patent/US9535695B2 *Completing load and store instructions in a weakly-ordered memory model* which we've already described (early release of loads once they become non-speculative).
The early release of loads when non-speculative is, perhaps, obvious (at least for a weakly ordered memory model); the patent is mainly about how you can implement separate load and store queues that maintain ordering relative to each other, while allowing both loads and stores to be (in-order) removed from one end of the queue while new instructions are added to the other end. It's worth looking at to see how these sorts of queues are implemented without having to move data between slots, and how wraparound is handled – but it will make your head hurt!

2015 (early release of non-speculative stores)

Finally we get the above-mentioned (2015) https://patents.google.com/patent/US10228951B1 *Out of order store commit* which gives us more aggressive store commit.
The difference here compared to the 2006 case is that 2006
- allowed stores
+ to release resources, and
+ commit to cache,
+ before retire,
- but this had to happen in ROB order.

2015
- allows the resource releasing out of ROB order.
Specifically, if the oldest store to be committed is a cache miss, then, while that cache miss is being served, we allow younger [but no longer speculative] stores to be written to cache.

It's interesting that they don't tell us how you can operate a low-power queue if you are allowed to remove items from the middle of the queue – obviously the circular queue techniques of the 2014 patent won't work!
Presumably they use ideas like those of the non-shifting reservation station, like we saw in (2015) https://patents.google.com/patent/US20170024205A1 *Non-shifting reservation station* when discussing

Scheduling .
(And it gives one confidence in Apple's design process if a good idea invented for one part of the CPU is immediately adapted to solve a similar problem in a different part of the CPU.)

2018 (how the machine tests that a load or store has become non-speculative)

This series of patents ends, for now, with (2018) https://patents.google.com/patent/US10628164B1 *Branch resolve pointer optimization*. In all the above work we refer to stores (or loads) as becoming non-speculative, and we know conceptually what that means, but not exactly how the machine implements it. The patent describes a way of implementing this. The problem is not quite as simple as you might think, because any given store might depend on multiple branches ahead of it, and those branches can execute out of order.
One way you might solve this is through a variant of our good old dependency bit vector scheme, with something like a bitvector that holds all the predecessor branches that have not yet executed. But there's an easier solution, the subject of the patent.

Simplifying to convey the point, imagine the stream of execution as it passes through Decode, where every instruction gets a sequential instruction number.
- Then any given store can know the number of the last branch before it in program order.
- If the Decode unit also propagates that number down to the Load Store Unit which propagates it to the Store Queue, then every Store in the Store Queue also knows the number of the branch after it.
- Finally as every branch is executed (possibly out of order) the number of the *oldest* branch that's still in the Scheduling Queue is noted. (The patent is somewhat vague about this, but it's the only interpretation that makes sense.)
- This oldest not-yet-scheduled branch is sent to the Store Queue.
Every store now knows that it lives between the branch ahead of it with sequentialID M and the branch behind it with sequentialID N.
When the oldest not-yet-scheduled branch sent to the store queue matches sequentialID N, then every branch earlier (ie branch M and forward) has been executed, and so the store (or load) is no longer speculative.

This sounds reasonable, but I don't understand why the behind branch sequentialID is required; why not just compare the branchID that's propagated to the Load Store Unit with the sequentialID of the store itself? I suspect there are a lot of details (how much can you trust that the Scheduling Queues and Dispatch buffer can precisely identify the oldest branch that has not yet been scheduled?) that ultimately rely in some way on this second behind branch sequentialID, but honestly I cannot understand the patent as it is presented, even when I try to fix up the parts that seem clearly wrong.

Testing the LSDP

Let's see what we can learn about the LSDP. The idea is to create varying types of store/load pairs and see how performance varies.
We build up complexity gradually.

independent addresses

To calibrate, start with a basic load and store to two different, unchanging, addresses, using two different registers:

```
STR x10, [x2]; LDR x11, [x3]
```
(x2!=x3)

There should be absolutely no interference here between the store and the load, and that's what we see.
We see nothing unexpected, and we see a throughput of 800 store+load pairs takes essentially 400 cycles, so two pairs (four memory ops) per cycle.

Now a slight variation:

```
STR x10, [x2]; LDR x10, [x3]
```
(x2!=x3)

What's different here? Note that the load now feeds into the store.

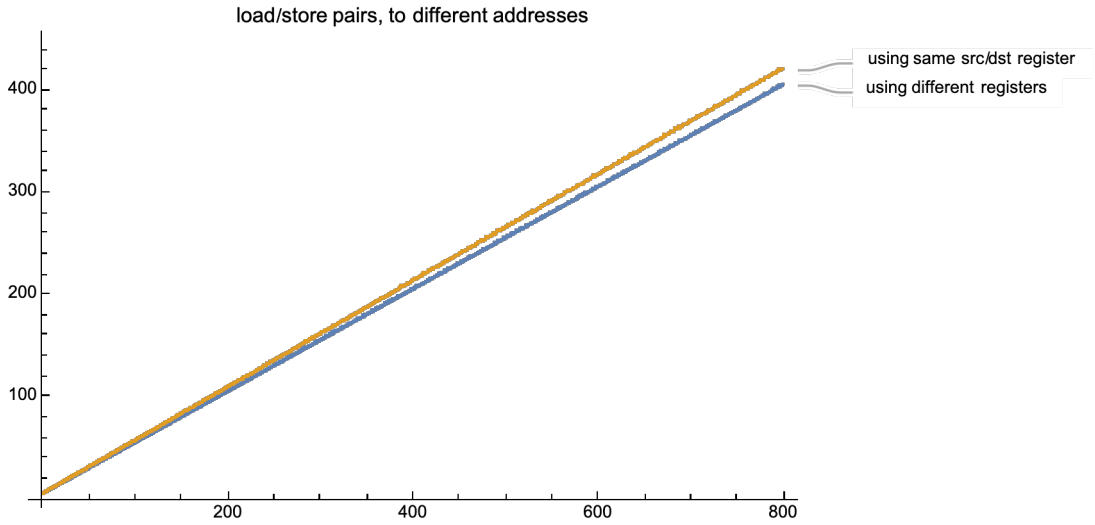
Your first thought might be that this has to run sequentially, each instruction waiting for the next. But be careful.
Write the code as

```
STR1 x10

LDR1 x10
STR2 x10

LDR2 x10
STR2 x10
```

etc

`LDR1` and `LDR2` are independent (they will load to different physical, renamed, registers).
What a common register enforces is that the store (one store) has to follow the load. That's all.
Hence there is nothing preventing each pair (as drawn) from executing in parallel, two per cycle (two loads, two store per cycle).
And once again that is what we see.

Out[172]=



The case of a chained dependency through the same register (gold line) is very slightly slower than the case of no chained register (blue line) but essentially the same speed of two pairs per cycle!

Given this understanding, we can now see why slight variants on this, using two different register widths, behave the same way.

`STR x10, [x2]; LDR w10, [x3]` (x2!=x3)

and

`STR w10, [x2]; LDR x10, [x3]` (x2!=x3)

also run at full (two pairs per cycle) speed.

same address (no prediction required)

Now we do the same thing, different registers but load/store to the same address.

`STR x10, [x2]; LDR x11, [x2]`

Once again your intuition is that we have forced a dependency, but again let's be careful. So write the code as

```
STR1 [x2]
LDR1 [x2]

STR2 [x2]
LDR2 [x2]

STR3 [x2]
LDR3 [x2]
```

Now think about how this code has to execute (for correctness) and how it will execute.

For correctness we require that

- every load (appear to) execute between the two store and

- every store (appear to) happen sequentially
But our OoO machinery is very flexible in how this is actually implemented.

The stores can occur out of order, just as long as the (address, data) pairs are placed into the Store Queue in the correct order (which is achieved by giving each store a Store Queue location at Mapping/Rename (while the stores are still in order).

The loads can also occur out of order, the only constraint being that each load of a pair must occur after its matched store.

So, conceptually, what now happens is that a whole lot of these loads and stores are thrown into the Dispatch Buffer and the Load/Store Scheduling Queues. Ideally their age is preserved in this process, and they are scheduled in order, but the Dispatch Buffer doesn't promise to preserve order, and the ambidextrous Load/Store unit may occasionally have a load placed in it the Dispatch Buffer rather than the perfect balancing of always stores. So ordering will occasionally be imperfect.

Assume perfect ordering. What should we expect? Some of this I have to guess, because no-one gives full details, but we have something like:
STR1 has been split into two instruction, StoreAddress1 and StoreData1, both targeting a particular slot in the Store Queue (slot allocated already).
STR1 (both parts) and LDR1 issue together.
Store Data places its data in the Store Queue slot.
Store Address and Load calculate their address, and perform TLB lookup in sync.
At that point, the next cycle or so, we have something like
- the store will place its address value in the Store Queue slot
- the load will send its address value to the L1D, and to the Store Queue
- if the timing is set up correctly, we can have something like the store places its value in the Store Queue slot in the first half of a cycle; the load performs a comparison against all the Store Queue slot addresses in the second half.
- so the load sees a match, the store data is already present, and the load completes (acquiring the data from the Store Queue rather than the cache).

If we have imperfect ordering (the load occurs one cycle or more earlier than the paired store) then what will happen is
- the load looks in the Store Queue, finds no matching entry, and gets the data from the L1D cache

For most cores that's the end of the story, at some later point the Store will execute, will compare its address to the addresses in the Load Queue, will see that its paired load has already executed (and acquired data from the L1D) and that load will be marked as having to be Flushed (or some similar recovery procedure).

For Apple (the 2019 *Load/Store Ordering Violation Management* patent we have already discussed) what will happen is that every store *in progress* will also compare its address against every load *in progress*. This means that, for example, right after address generation (even before TLB lookup) the store will have some sort of indicator of where its data is going. This can be compared with the in-progress load's virtual address and if they match, then that's simply an early version of the load's eventually matching the address in the Store Queue! (Of course this mechanism is not perfect; it won't catch weird cases like the store written to a physical address that's the same as the load, but via a virtual address that is different. And it may not catch weird partial overlap/alignment cases. But it will catch *most cases early*, with all cases being caught by the later tests.)

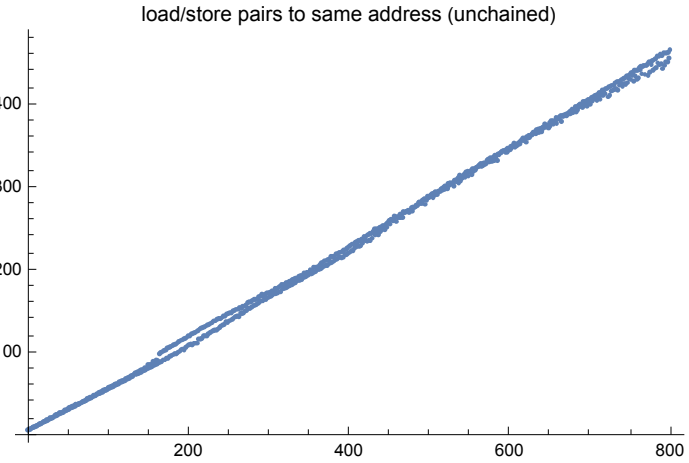
This means that even if the store started two or three cycles after the load, the there may be enough of an overlap in time for the load to detect that it has a matching store in progress and should not get its data from the cache. In theory the load might be able to get the data from the Store Queue (the store that matched its virtual address with the load knows its Store Queue slot, and the Store Data instruction may already have dumped its data there). I don't know if Apple does that. Even if they don't, the load can be marked as a Replay, and so it re-executes a cycle or three later, at which point the Store has fully completed, and the load can match its address against the Store Queue and pick up the data from there.

This is what we see in the curve. There's clearly some messiness (the occasional Replay) but overall our performance has not slowed down too much.

Note that what we have seen so far can be summarized as:
- if a register is shared, store has to happen after load
- if an address is shared, load has to happen after store
but in both cases, with only one of these two (register and address) shared there are no *serial* dependencies. There is only the dependency of each individual load/store pair, and the pairs can all execute independently. Hence we run at essentially two pairs per cycle in all these cases.

- no LSDP is necessary because the addresses (of the load and store) are known early enough. Ideally the loads execute at the same time as the stores (or a cycle later) so when the load looks up an address in the Store Queue, it finds its matching address.
Even in cases of a few cycles of slip between the load and the store we are still OK without an LSDP because our ability to compare an in-progress store address against every in-progress load address catches these cases and handles them without much overhead.

Out[174]=



same address but delayed data (prediction required to avoid replay)

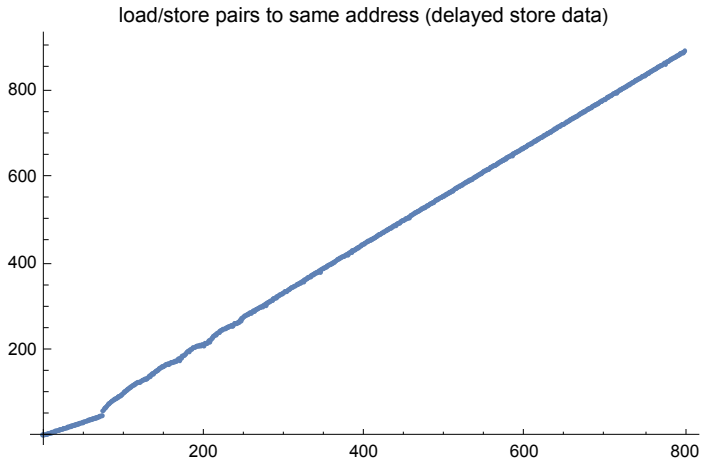
Now suppose that the data of the store is only available very late, but the address of the store is easily available. We would expect now that, just as above, mostly the loads and stores execute in sync, and the load hits a point where it has matched in the Store Queue, but the store data is not available. So it will have to Replay once the Store Data part of its associated store executes.

We can force this be using a slow instruction like FVCTAS to generate the store data, so our probe looks like

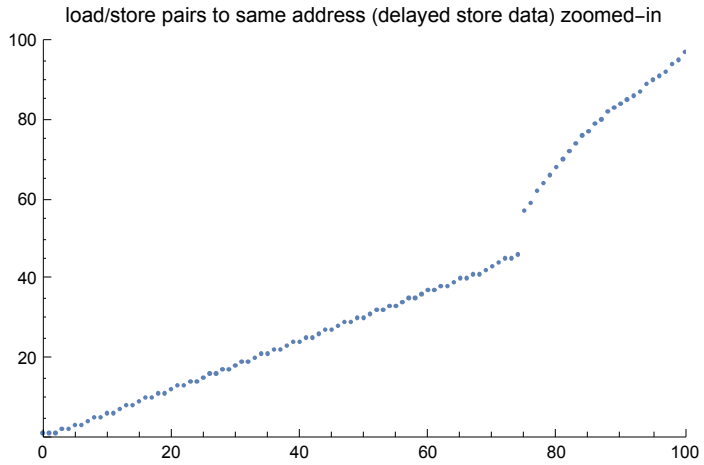
`FCVTAS x10, d1` (with d1=0.0) this instruction has a latency of ~13 cycles and converts d1(=0.0) to an integer

```
STR x10, [x2]; LDR x11, [x2]
```

Out[176]=



Out[177]=



Honestly, this is better than I expected!
There are clearly two regimes.
Up to 74 pairs, the machine processes about 1.6 pairs per cycle. Not quite two pairs, but close'ish.
After 74 pairs, the machine processes about .8 pairs per cycle.

So let's think about this.
Ideal execution would be that each load is delayed long enough that it only issues just before the x10 value is ready to be stored.
The machine is not psychic enough to do that!
But what it can do is record each time a load issued too early compared to its store, and record that in the LSDP.
Then the next time the load is placed in the Scheduling Queue, it will wait around with the store as a dependency, and will not execute until the store has executed (address and data).
If the machine could do that perfectly, then once the predictor is trained we shouldn't have to lose any cycles, we'd just have every load waiting around (for a long time) in its scheduling queue until its store was ready. In the real world the delay between the load and the generation of the store data is long

enough that scheduling queues will fill up, instructions will slip out of order, cycles will be lost while no appropriate instruction can be found in a particular queue, etc.

What we see is that we get close-ish to that ideal case for up to 74 load/store pairs. This suggests
- that the LSDP can hold ~74 entries
- when everything is working, load is delayed by the LSDP to the correct time required by the store data, so the load does not have to Replay.

Once we get past the capacity of the LSDP we switch to essentially one pair per cycle. In interpret this as meaning that the execution of every paid proceeds like
Store Address + Load (execute same cycle)
Load - sees no valid store data, sets a Replay dependent on the Store Data arriving for that particular Store Queue slot
Load - Replays successfully.

Now that each load of the pair has to execute twice how long would we expect a pair to take?
If 100 pairs take 50 cycles (no Replay), then naively 100 pairs take 100 cycles (Replay, each load executes twice).
This isn't quite correct because in principle three loads can execute per cycle if there are no stores (but can this happen for Replay conditions?), but let's accept it. Then we'd expect ideal throughput to drop to 1 (or a little over 1) pair per cycle.
We have all the imperfections of before, plus new collisions that can occur in the LEQ, so let's just round up the .8 per cycle we see to close enough to 1.

I think it's reasonable to conclude that
- we have seen the LSDP make its appearance
- it can hold about 74 entries
- when working correctly, we don't lose much load/store performance to the severe out-of-order execution generated by the slow production of the store data
- even when the LSDP can't help us, this particular case (address known, just data unknown) is handled adequately by the Replay mechanism. We lose half our throughput (since every load has to execute twice) but no more than that.
And as always what we are testing here is extremely unbalanced code! Normal code may well have store data that is very delayed, but it is unlikely to have a load of that store data immediately after the store, or to have such a high density of loads that the cost of the extra Replay loads is a noticeable additional burden.

same address, same register

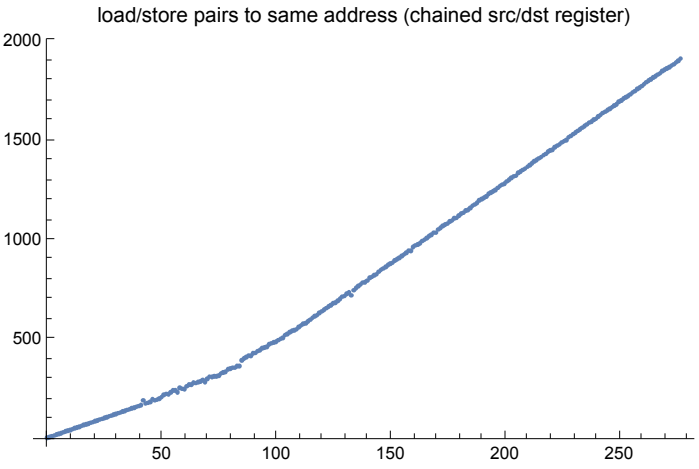
Now use a chained register:

```
STR x10, [x2]; LDR x10, [x2]
```

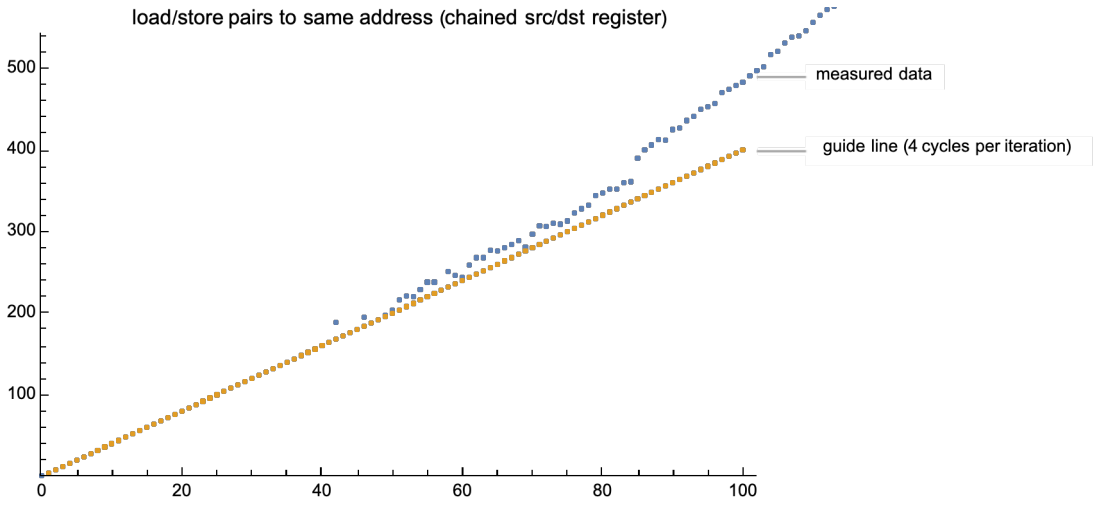
Unrolled this looks like:
STR1 x10, [x2]
LDR1 x10, [x2]
    STR2 x10, [x2]
    LDR2 x10, [x2]
        STR3 x10, [x2]
        LDR3 x10, [x2]

Now we stated that a common register required the store to follow the load. And a common address required a load to follow a store.
So we're basically stuck.
Unlike the previous cases, we can no longer separate the chain of instructions into independent load/store pairs that are run in parallel.
The best we can hope for is to run STR1 and LDR1 in parallel, meeting after three cycles at the STQ entry, another cycle to get that data value up to the bypass bus (and stored in the physical register corresponding to LDR1's x10); then we start the next pair of STR2+LDR2.
So we expect best case throughput of one pair per four cycles.

Out[179]=



Out[180]=



Clearly there are two regimes.
The initial regime can support about 70..80 load/store pairs. While in that regime the cost of a single load/store pair is about 4 cycles (slope of the initial section).
Once we transition to the slow regime, the cost of a single load/store pair jumps to about 8 cycles.

So this matches our analysis.
One thing it confirms is that loads and stores to the same address (ie the pair `STR [x2] LDR [x2]`) do indeed behave optimally. They can issue together, and the store address can be placed in the Store Queue early enough for the load to detect that address when it probes the Store Queue.
For the load to acquires its data from the Store Queue rather than the Cache is called *Store Forwarding.*

Secondly it confirms that the size of the LSDP seems to indeed be around 74 elements.
My assumption is that the replacement policy for the LSDP is not especially sophisticated. Suppose you want to add an entry to the LSDP? What slot do you use?
If any slot is marked invalid, that's an obvious choice.
If there are no invalid entries, then low confidence entries are the next obvious choice.
Then entries that are used to avoid a Replay rather than to avoid a Flush (since Replay's are much cheaper than Flushes).
But when all entries are essentially the same in these respects, what do you do?
- Random is one choice. This requires remembering no state.
- Another is LRU, which, in exact form is difficult to implement.
- Another is MRU, also difficult to implement and probably optimal for streaming situations.
- A fourth option is to track the entry number you replaced last time and just increment that with wraparound.
For "normal" code with a mixture of memory references of all sorts, this will behave like Random, skewed towards LRU (good).
For streaming code, it will behave like LRU, which is pessimal.

This fourth option seems to me to be what we are seeing.
In the earlier case where we had delayed store data, and a clear break in the curve, that suggests LRU (behaving pessimally, with essentially zero reuse).
This current curve has a much less pronounced break at the transition point, so it's behaving more like Random replacement of the LSDP, I'm guessing that's because entries are being added to the LSDP in a somewhat more random manner as execution proceeds. (Before the LSDP is populate, the stores will all be held back relative to the loads, because the x10 dependence is immediately visible to the scheduler; which means multiple loads will initially start too early compared to their subsequent stores. Throw in the ambidextrous load/store unit, and pretty soon you have a major mess in terms of the order in which the load store dependencies are detected and then recorded in the LSDP.)
Under Random (or Random-equivalent) replacement we'll see some entries being removed that are still being used, even as other entries would have been a much better choice for removal, so we should start to expect that even at only N=50 or so, we start to see some occasional slowdown because some loads that should have been held back by the LSDP have had their entries replaced prematurely and sub-optimally.

For most code the choice of LSDP replacement policy probably doesn't matter much. If you are writing code that is streaming (so it has a controllable structure and it's constantly generating loads that match the address of recent stores, you're probably doing something terribly wrong!) For normal code, Random is a reasonable choice, and if Apple do what it looks like to me, they get something probably a little better than Random because it skews to LRU.

The second regime is as we discussed before. In the second regime We are still trying to execute like
STR1 x10, [x2]
LDR1 x10, [x2]
    STR2 x10, [x2]
    LDR2 x10, [x2]
        STR3 x10, [x2]
        LDR3 x10, [x2]
but we do not have the LSDP to enforce the precise synchronized scheduling of each store with its matching load. So most loads execute early relative to their stores, don’t see the data available in the Store Queue (but are caught as Replays) and are forced to Replay. So our minimal cycle becomes
STR1 x10, [x2]
LDR1 x10, [x2] (a cycle or two off earlier than the store)
LDR1 x10, [x2] (replay)
    STR2 x10, [x2]
    LDR2 x10, [x2] (a cycle or two off earlier than the store)
    LDR2 x10, [x2] (replay)
        STR3 x10, [x2]

LDR3 x10, [x2] (a cycle or two off earlier than the store)
LDR3 x10, [x2] (replay)

The minimal timing of one loop becomes not the latency of one load but the latency of a load+replay.

Experts might ask Zero Cycle Loads in this context, and if that changes anything. Zero Cycle Loads will be explained towards the end of this load/store section.
The short answer is no, because the ZCL still has to be validated, and that takes the full timing cycle described above.
But to check this, I ran the code three times.
The second time used LDR x10, [x2, x0] (with x0 set to 0). An indexed address of this form is not susceptible to the basic ZCL but is still susceptible to the strided address ZCL.
Any simple attempts to bypass the strided address ZCL by varying the common address used by the load and the store is strided, so will still be captured by the ZCL! So let's use a quadratically increasing address stream!
initialization

MOV x0, #0; MOV x20, #1

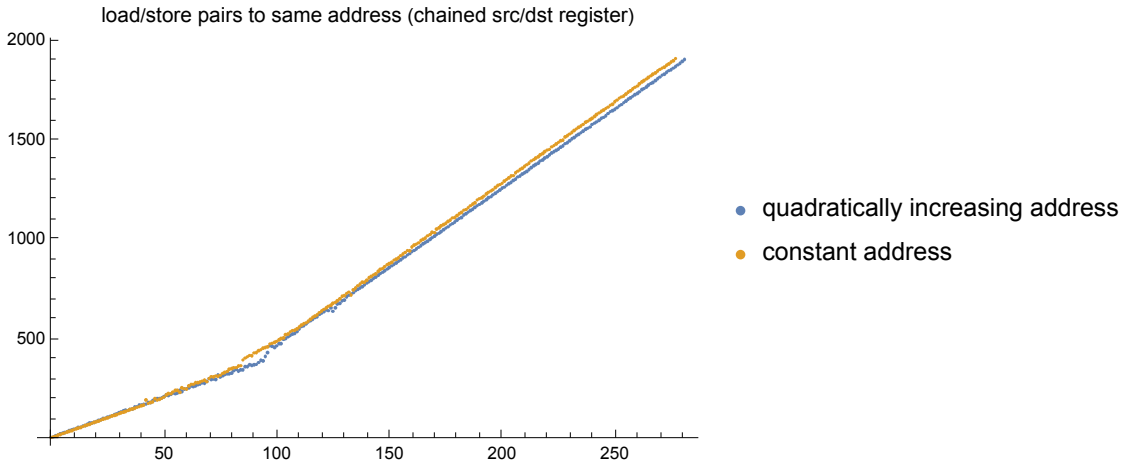
loop

STR1 x10, [x2, x0]
LDR1 x10, [x2, x0]
ADD x0, x0, x20
ADD x20, x20, #1

but even this quadratically varying index stream (no ZCL!) has essentially identical results.

Remember that the LSDP matches a store PC with a load PC. The whole point is that it does not know that actual load and store addresses (if we knew those, we would not need a predictor!) So apart from the ZCL issue, we should expect (and we see) no change even when the addresses being shared by the loads and stores change.

Out[182]=



force address prediction (delayed address)

Now we' re going to repeat these tests, but forcing the store address to be unknown at the time of the load. The plan is the same as how we generated delayed store data:

• generate an integer (slowly!) using
`FCVTAS x0, d1` (with d1=0.0) this instruction has a latency of ~13 cycles and converts d1(=0.0) to an integer
follow this with
STR x10, [x2, x0]; LDR x11, [x3] (x2!=x3)

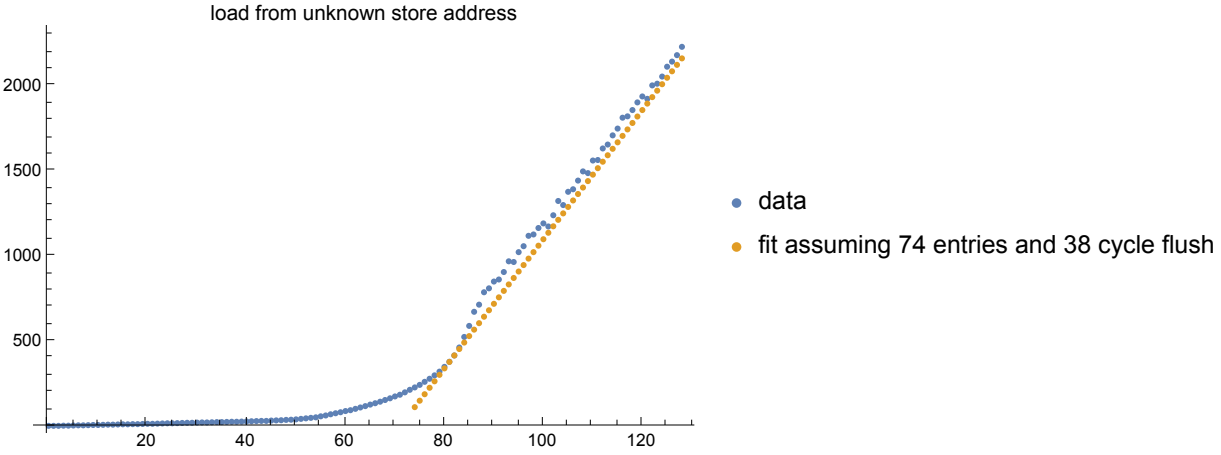
So the probe looks like repeated (`FCVTAS STR LDR`).
Once again the load and store are completely independent.
If the LSDP is working correctly, they should not interfere with each other in any way, and we should see no slowdown.
Even when we do overflow the LSDP, as long as loads are predicted by default never to match unknown store addresses we should still see no interference

And we see nothing unexpected, nice smooth two pairs processed per cycle.

load from (unknown) store address

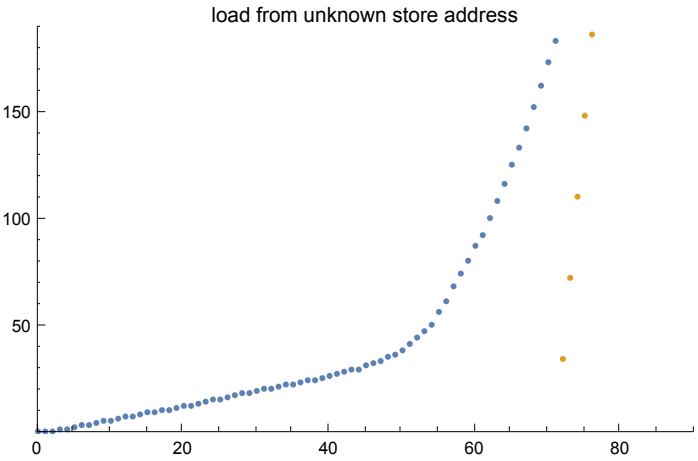
But now one small change, force the load and store to share an address:
`STR x10, [x2, x0]; LDR x11, [x2]`
Note that these correspond to the same address, but we are not forcing the load to delay on the x0, so it should be able to execute *well in advance of the store*.

Out[184]=



Yikes! Let's zoom in:

Out[185]=



So up till about 50 pairs we get adequate throughput. Not the 2 pairs per cycle I would expect, but something like 1.4 pairs per cycle. Honestly I can't see why the gap between what we expect and what we get is quite as low as it is. Each load will have to delay until its store executes, but in theory these should all be able to just wait in the various Scheduling Queues until the stores are ready, then immediately execute, with few lost cycles.

However we are more concerned with what happens once we exceed the capacity of the LSDP. The slope of the second half of the curve is about 38 cycles/pair!
I think this splits into something like 25 cycles as the cost of a flush caused by a load/store aliasing., and 13 cycles as the cost of performing the `FCVTAS` after the flush, either way the cost of a Flush is clearly phenomenal compared to either correct execution or a Replay.

The execution pattern is:
- the load executes (far in advance of the store, which is waiting for the `FCVTAS` result).
- It isn't stopped by the LSDP (which has overflowed and so has no matching PC) and so
- it executes assuming it matches no address in the Store Queue.
- Later the store address becomes available and is compared to the address in the Load Queue,
- we see that the Load picked up stale data (either from the L1D or an earlier entry in the Store Queue),
- and so the machine forces a Flush after that load.

(In principle doesn't have to be this way! Suppose the value that was loaded were stored in the Load Queue next to the Load Address. This could be compared with the Store Data and if they match, why Flush?
No-one does this, but given that repeatedly storing and loading the same data is not that uncommon – sometimes for good reasons, sometimes not – it seems worth at least running a simulation to see what sort of a win this could buy you.)

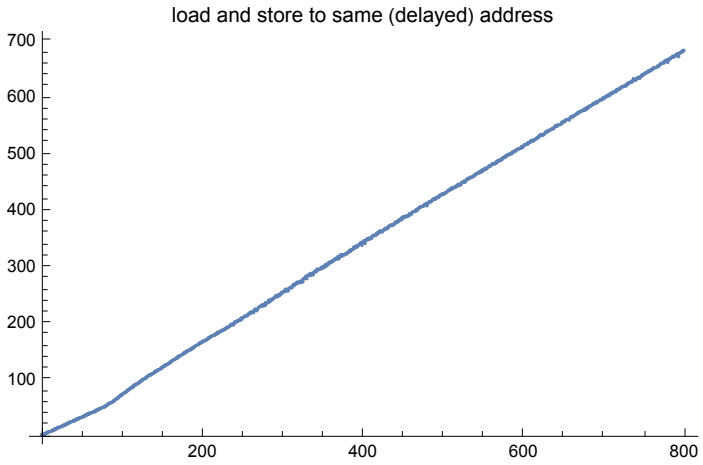
The curve suggests the LSDP holds only about 50 elements. But that's an illusion caused by the massive increase in cycles when even just a small fraction of the load instances cause a flush. I added the gold line to both curves to show how we're still really dealing with ~74 entries in the LSDP.

force the load to be delayed like the store

What if we also force the load to delay, by making it also depend on the x0?

```
FCVTAS x0, d1; STR x10, [x2, x0]; LDR x11, [x2, x0]
```

Out[187]=



Now we see something like the first stage of the LSDP curve.
We see an initial slightly faster regime, about 1.5 pairs per cycle, for N<~70..80.
Then about 1.2 pairs per cycle.

So performance is adequate. Not great, when we compare with the 2 pairs per cycle when the load and store addresses are both known, but clearly we're just getting some sub-optimal scheduling (at N<74) and the occasional Replay (at N>74), not many Flushes.

As I've said earlier, I wonder if at least some of this glitchiness is the result of the ambidextrous load+store Execution unit?

Loads and Stores will be placed across the load and store Scheduling Queues as the Dispatch Buffer sees best, but because that one unit will occasionally switch from serving loads to servicing stores, the various queues will slip out of perfect synchrony...

shift the timing of the load relative to the store

Now let's tie these ideas together. Suppose we have a probe that looks like

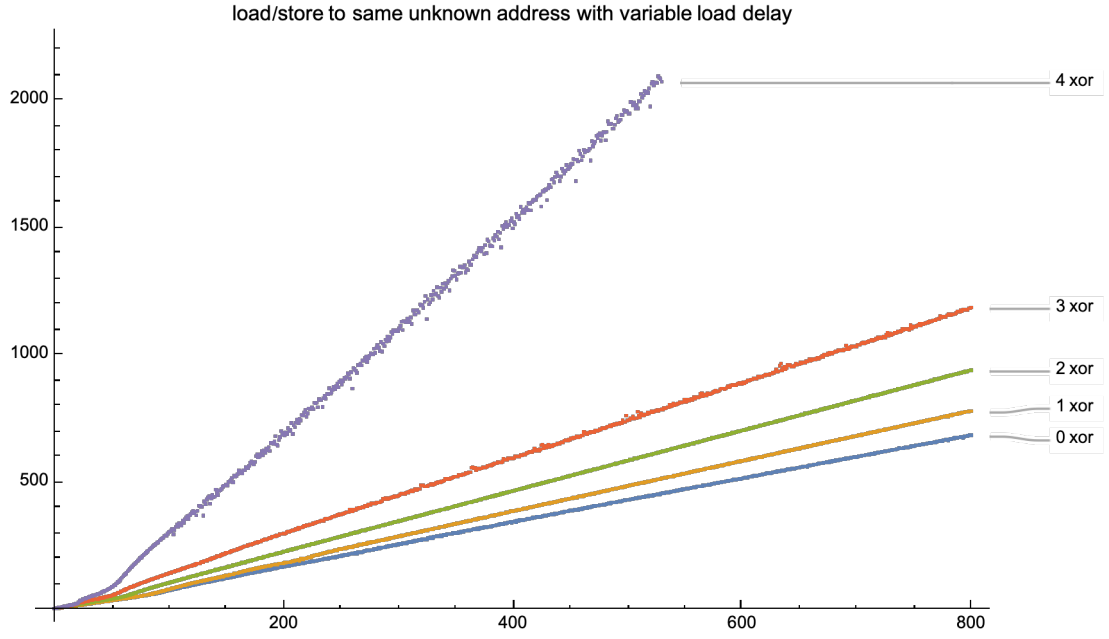
```
FCVTAS x20, d1
EOR x0, x20, x20; (followed by some number of EOR x0, x0, x0);
STR x10, [x2, x0];
LDR x11, [x2, x20]
```

Remember that on M1, unlike x86, xor is not a zero'ing or dependency breaking idiom, so `EOR` takes one cycle, and the successive EOR's are chained, so each one adds a cycle of delay. Of course the EOR generates a zero value, so x0, like x20, continues to remain at value 0.

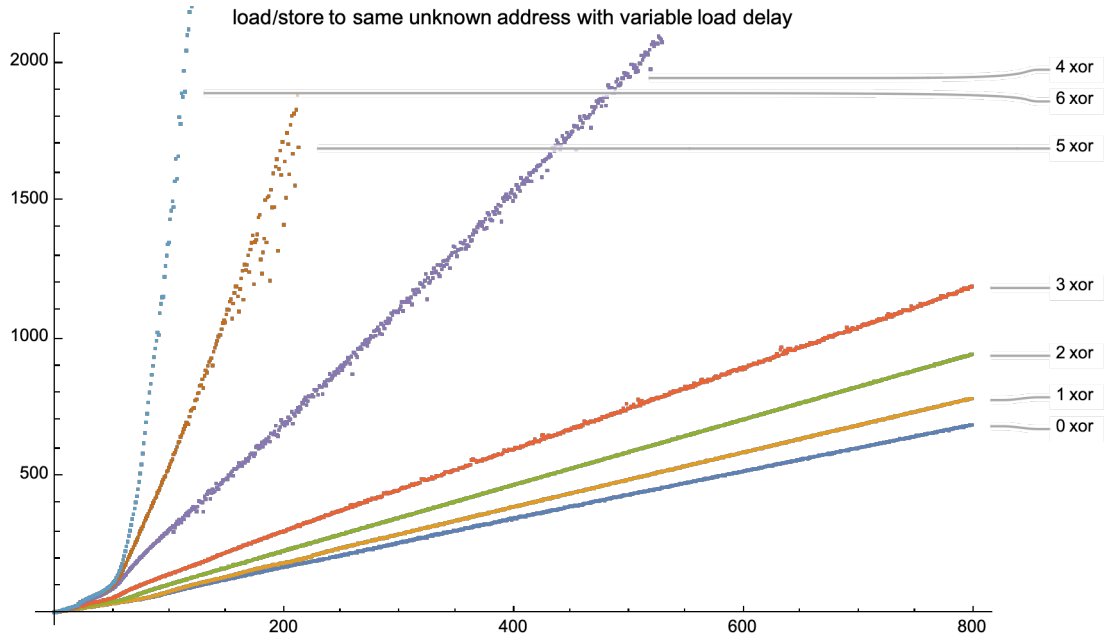
This means that (with just the first `EOR`) we have now forced the store to be one cycle later than the

load, so higher chance of collision.
And we can vary the number of EOR's to move the distance to two, three, ... cycles later than the load.
What do we see?

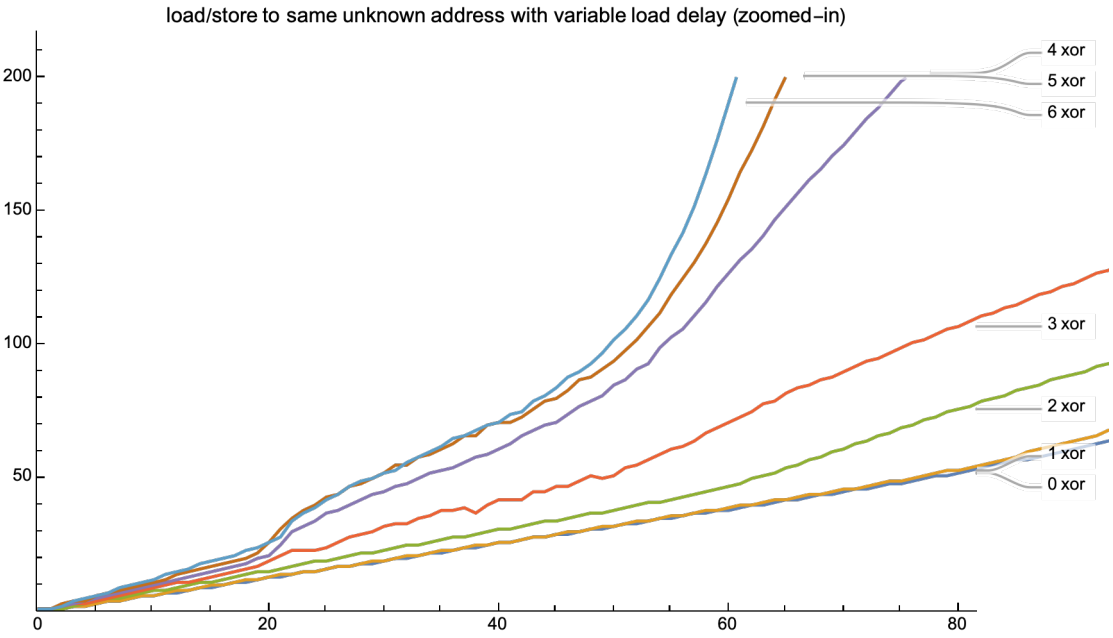
Out[194]=



Out[195]=



Out[196]=



So as we said, the baseline (no delay) gives us about 1.2 iterations of (FCVTAS, store, load) per cycle, so about .8 cycles per iteration.
As we add 1, 2, 3 cycles of delay between the store and the load, this increases to about 1, then about 1.2, then 1.5 cycles per iteration.
Obviously we're doing slightly more work (the xor's, but that basically trivial); more important is probably that ever more cases are being caught by late detection of load/store matching and being converted into a Replay.

By 4 EORs, each iteration is taking a little over 4 cycles, and pretty much every load/store pair is being run as a Replay. We start to see a small effect at N=~50.
At 5 EORs, each iteration is taking 10 cycles, and we clearly have a mix of some Replay cases with a fair number of Flush cases;
By 6 EORs (ie delay between the store and the subsequent load of 6 cycles) we're essentially at the case where every load/store pair generates a Flush, as soon as we exceed the capacity of the LSDP.

Examining the zoomed-in plot for small values of N, I think the case for N<~20 corresponds to enough temporary storage (STQ and Scheduling Queue) to hold all the pending loads and stores over multiple loops; enough so that a Replay doesn't slow us down because it can happen in parallel with other load/store pairs performing various parts of their executions.

Past N=20, without enough storage to hold all this state, we begin to have to delay processing later loads/stores because all the temporary storage is occupied by items in various stages of partial execution
- waiting for the value of an address index (x0 or x20),
- waiting for a chance to Replay.

And delaying those subsequent loads/stores make the time taken to process them visible because it's time not overlapped with other stages of the load/store processing of other loads/stores.

So we have that effect (exceeding temporary storage) hurting us up until N=~50, but even up to ~50 performance is not catastrophic, because the LSDP is preventing mostly preventing Flushes. We spend a few cycles with every load and store waiting in all the various temporary storage, but we don't Flush. Past N=50 we begin to Flush, and then everything goes terribly wrong terribly fast.

various nasty cases

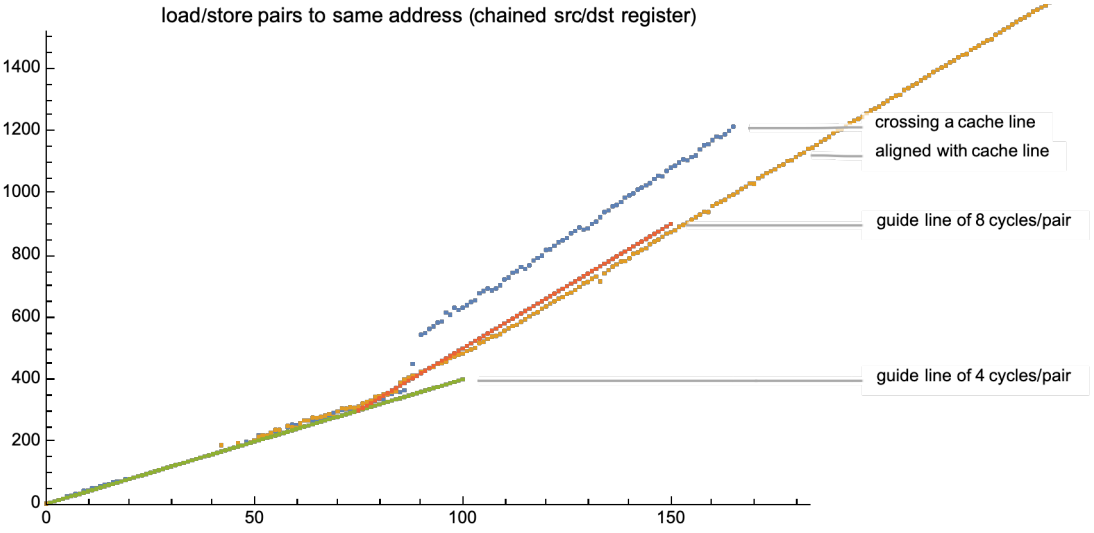
Let's now try a few nasty cases to see how Apple copes.

unaligned loads and stores

Till now we have had x2 and x3 perfectly aligned, at the start of a page boundary.
Let's add 127 to each one, so that now load/store of anything but a byte crosses a cache line boundary.
This has no effect on throughput of the basic
`STR x10, [x2]; LDR x11, [x3] (x2!=x3)`
case, which is fairly impressive! Somehow we are getting four *cache-line-straddling* accesses per cycle, even if we accept that Zero Cycle Loads are shouldering some of the burden.

The easy variants on this behave in the same way. The more difficult variant
(same register, same address, latency has to be at least 4 cycles and is frequently 8 once we overflow the LSDP) shows similar behavior.

Out[198]=



For few enough (~74) pairs, behavior is identical in both cases (load/storing at the beginning of a cache line, vs straddling two cache lines).
And there is a similar slow regime, where every load has to Replay once (the slope is slightly higher, about 8.5 cycles per pair rather than 8 cycles per pair, but essentially the same)

But there is a pronounced jump (by about 130 cycles!) between the two regimes.
I have no idea what's causing this. The only difference between these two cases is that the blue case performs its load/stores crossing a cache line.
This equivalence of the earlier, faster, regime (where we assume the LSDP is ensuring that Replay's are never required) suggests that the LSDP doesn't need to anything strange about loads or stores that cross cache line boundaries (which makes sense, since it is blind to the actual values of the load and store addresses).

How about the following theory:
- At the point where the LSDP stops being a useful predictor, the Store Queue is full of retired but not yet completed stores (split over two cache lines).
- One of the loads at this point generates a Flush.
- The Flush forces that entire Store Queue to be written out. Maybe this isn't normally done at Flush, but is a special case for stores split over two cache lines or something?
- Forcing out all that data generates the one-time obvious jump? (
+ 60 Store Queue entries,
+ each one has to be read twice (for each of the two cache lines), reads on separate cycles because each STQ entry is single ported,
+ each write to cache has to happen serially in sequence because they're all to the same address.

But I will admit this is special pleading, the result of the similarity between the size of the gap (~130 cycles) and twice the size of the Store Queue. Why don't we see a similar 60 cycle gap for the aligned case?

If we really engage the LSDP with our delayed address calculation
`FCVTAS x0, d1; STR x10, [x2, x0]; LDR x11, [x2]`
we again see the same behavior as before, with no serious differences.

vector load/stores & load pair/store pair

Now let's try vector load/stores. All the different variants behave as expected, including LSDP behavior, Zero Cycle Loads, and no problems with mixed used of q0 and s0, one for the load, one for the store.

Same is true for load/store pairs.

partially overlapping loads and stores

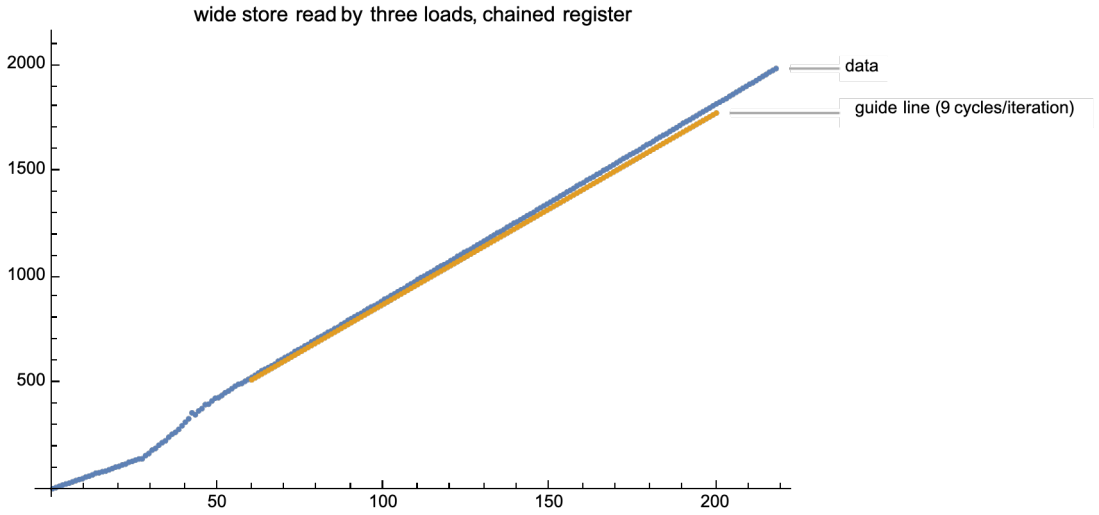
Now let's try overlapping loads and stores.
Start with
```
STP x10, x11, [x2]
LDRB w12, [x2, #0]; LDRB w13, [x2, #1]; LDRB w14, [x2, #2]
```
This has no real need for address speculation , but does involve store to load forwarding.
The basic unit involves one store and three loads, so (assuming everything is timed correctly) we should be able to perform one iteration per cycle.
And that's what we see.

(If we are correct that each Store Queue Entry is single ported, then the three loads will have to execute sequentially as far as loading their data is concerned. But they manage to co-ordinate this smoothly enough that there's no serious delay.)

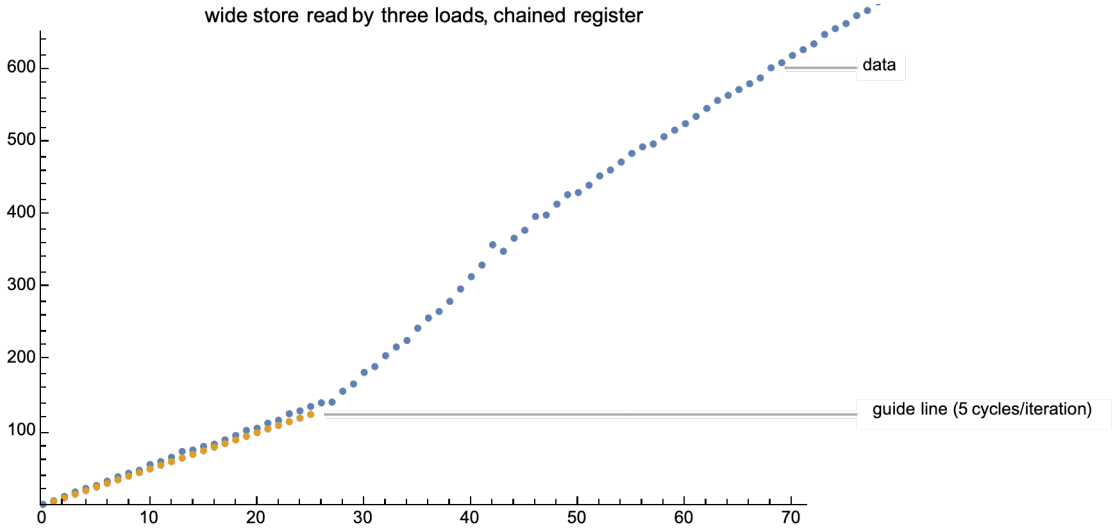
Now let's introduce register chaining so:
STP x10, x11, [x2]
LDRB w10, [x2, #0]; LDRB w11, [x2, #1]; LDRB w14, [x2, #2];

Now we do see something slightly different, though hardly a disaster.

Out[200]=



Out[201]=

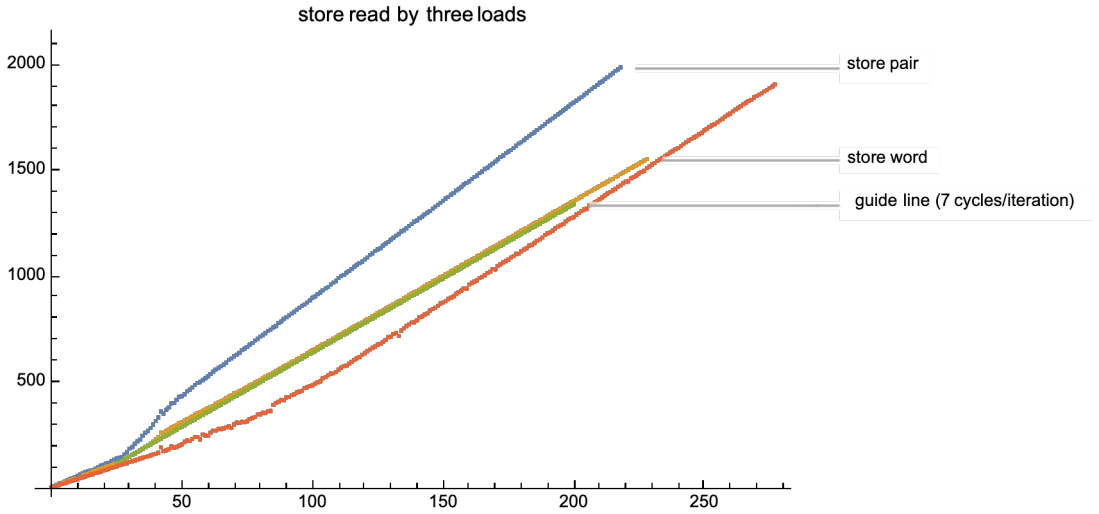


Overall much what we expect, but with a few differences in the details.
The largest difference is that we move to the slower part of the curve at around N=25 rather than 74. Presumably each pair of (initial store, one of the loads) occupies an LSDP entry, so the effective size is reduced to a third.

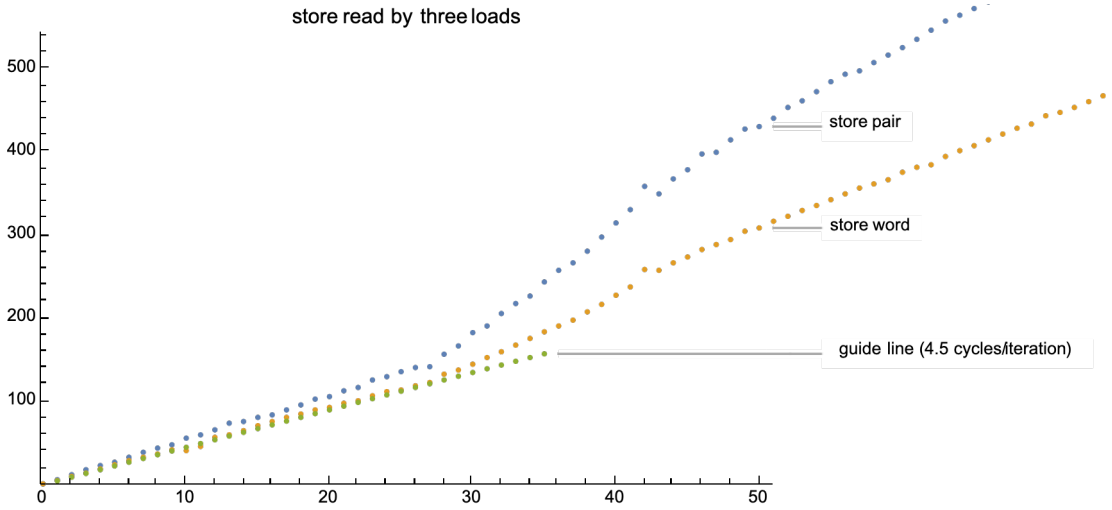
The fast regime runs at 5 rather than 4 cycles/iteration. And the slow regime at 9 rather than 8 cycles/it-

eration.
Let's try to figure out where these differences come from.
First let's drop the STP and switch to storing just a single register.

Out[203]=



Out[204]=



So it seems that the primary problem was the Store Pair. Even in the case where the LSDP is fully functional, the primary loop
STP x10, [x2]
LDB w10, [x2] (+ two extra loads)
is one cycle longer. In principle there's no reason why this is necessarily so that I can see. In principle the STP could be performed as a single 128b store that has no interest in the fact that the 128b were created from two registers.

If we replace the store pair with a store word we see that in the fast case an iteration now takes 4.5 cycles per loop. That's a consequence of the three loads, but stays at that value for two loads. (If we use only two loads, of course we now get the LSDP as useful up to N=74/2=37, but the slope remains 4.5

rather than 4). Likewise the high slope remains 7.

Now this is honestly rather strange! Why would the more complicated probe
STR x10, [x2]
LDB w10, [x2] (+ two extra loads)
run faster (once it has to continually Replay) than the simpler probe
STR x10, [x2]
LDR x10, [x2]
?
Even
STR x10, [x2]
LDR x10, [x2] + LDB x11, [x2,#1]
runs faster, at 7.5 cycles per iteration rather than 8!

The only thing I can imagine is that the presence of the extra load(s), once the LSDP is no longer helpful, breaks up sub-optimal scheduling patterns. The extra load (and the delay it engenders when it Replays to access the Store Queue slot) serves to occasionally delay the LDR x10, [x2] enough that it is not required to Replay.

Finally recall that
STR x10, [x2]
LDR x12, [x2]
took half a cycle per iteration, because the fundamental units are the STR/LDR pairs, and each of these can execute independently, so the only real constraint is we can perform 2 loads and 2 stores (ie two pairs) per cycle, ie half a cycle per pair.

Modify this to
STRB w10, [x2, #0]; STRB w11, [x2, #1];
LDR x12, [x2]
The change to storing a byte is not, by itself, a problem. A single store byte runs at the same half a cycle per pair, for the same reason.

We now have two stores in our triplet, so the maximum speed at which the triplet could execute, with everything independent (independent registers, independent addresses), is now one triplet per cycle (ie two stores per cycle). That's what we see.
But what if we now have one of the bytes overlaps with the load, as in
STRB w10, [x2, #0]; STRB w11, [x2, #100];
LDR x12, [x2]
Now we take ~1.33 cycles per triplet, likewise if both byte stores match the LDR address.
Obviously there is extra work that's involved in this second case, in that the load has to acquire values

from two store queue entries and the cache, then stitch them all together. It looks like this (specifically the stitching together) takes an extra cycle. There's no reason, in principle, why this extra cycle should be visible to us; it's not part of any dependency chain. My guess is that the speculative scheduling is done on the assumption that each load will take four cycles, and when they take five it's not a catastrophe but one cycle in three two loads, or a load and a store collide in some stage and one of them has to be delayed by a cycle. In principle, with more resources in the LSU, this doesn't have to happen.

If we convert the stores to store the lower and upper halves of a word, the cycles per triplet remains unchanged, which suggests the stitching together does take an additional cycle (since this case requires no cache access, only the two store queue accesses).

There's a whole lot more that could be investigated here to investigate precisely all these various unusual cases, but the broad contours are clear enough that I think it's time to move on, past Replay and the LSDP.

conclusion

So I think we can conclude that this validates
- there is an LSDP
- it can hold about 74 pairs of (storePC, loadPC)
- it prevents both Flushes (catastrophic) and Replays (not catastrophic, but a few wasted cycles)
- there is also late checking of whether loads might have collided with stores (catching cases where the load is two, three, even four cycles early; and converting them into Replays), and this is probably extremely helpful in real code, converting Flushes to Replays even on first time code that isn’t part of loops or otherwise is not present in the LSDP.

We have seen the evolution of the LSDP from preventing just Flushes to trying to prevent both Flushes and Replays, with a few compromises made to the Flush part of the design to better accommodate the Replay part.
I wonder if, following the standard Apple pattern, it’s time to disaggregate the LSDP into two predictors, one handling the Flush case only, optimized for accuracy but not needing to be very large; and a second predictor handling the Replay case which does not have to be as accurate but should be much larger (eg based on a direct-mapped or two way set associative design rather than a CAM-based design). Replay is not that expensive, so it’s not a catastrophe if a few Replaying Load/Store pairs are not captured because of collisions in the hash of the PC to an index, as long as most are; and these two issues can be balanced with a more cache-like design.

Continuing the theme of disaggregation, there are suggestions for how to split the Store Queue by function (in the same way we saw the Load Queue split by function). Probably the best match of these to Apple is (2006) https://zilles.cs.illinois.edu/papers/baugh_lsq.pac2.pdf *Decomposing the Load-Store Queue by Function for Power Reduction and Scalability* which suggests a small high speed structure, tightly linked to the LSDP, which handles forwarding stores to loads, and a second large, banked and

slow structure that handles testing that loads and stores do not conflict. (Obviously we hope the LSDP will catch most such cases, and any that are left are not performance critical given that we will be Flushing anyway.)
Of course this philosophy (use as minimal a store forwarding structure as possible) is in direct opposition to a different paper we have already seen, *The Untapped Potential of the Store Queue*! Is the energy win from using lots of store queue forwarding (and so avoiding L1D lookups) more than the energy win from minimal store queue forwarding and a lower energy structure to hold most stores? Well, that's what makes microarchitecture interesting!

Load Accelerators

Because loads are common, and high latency, we do whatever we can to make them faster.

Obviously one track, with which you are familiar, is multi-level caches.

A second track, one we have discussed in immense detail, is ensuring that loads are not slowed down by store any more than is absolutely essential (the LSDP) and are not slowed down much by slight glitches in timing at any stage of load execution (Replay).

But a third track is to make loads provide a result faster than it takes to access the L1D.
To understand how this could be done, we need to think abstractly.
Consider Loads to be an instance of "matching" the load with a "source" via some "matching mechanism".

Traditional loads find their source data in the L1D, and perform the matching via the physical address.

Acceleration via the Store Queue

Any modern machine (store forwarding) will have loads that match recent stores find the data in the Store Queue, and the match is via the physical address. One could imagine an alternative to this.
Suppose the match were by virtual address. This means that the comparison of the load address with the Store Queue entries could happen right after address generation, in parallel with TLB lookup.
This could provide the data one, maybe even two cycles earlier, for a reasonable fraction of loads. (The *Filter Caching for Free* paper we have already mentioned suggests about 18% of loads hit in the Store Queue for a Skylake-sized Store Queue of 56 entries, similar to M1's 60 entries).
The big problem with this idea is that it hurts Speculative Scheduling:
- do you schedule dependent instructions assuming the load will provide data in two cycles, or in four cycles?
Presumably one can (as always) build a predictor, and then it's a question of the speedup vs the energy cost of the predictor.

But we can also look at this from a different direction.
Consider the LSDP. This mostly succeeds in tying a particular store (by PC) to a particular load (by PC). In other words, we are matching source and load by the PC of each.
What if we generalized this? Imagine a table much like the LSDP but the way it works is:
- every entry in the Store Queue records the store's PC
- if a load hits in the Store Queue, that load's PC and the store's PC are recorded in the table
- in future, when a load matches a load PC in the table (comparison performed eg at Mapping time) then instructions dependent on that load can be speculatively scheduled assuming a load latency of 2 rather than 4 cycles. Failure will result in a Replay.
- of course we can add the usual bells and whistles as necessary, eg confidence tracking, aging out entries, arming when we see a matching store, ...

This might allow us to capture not just the lower energy savings possible by reading the Store Queue rather than the cache, but also the latency savings.

Acceleration via Registers

2012 ZCL patent (register renaming based on common base register)

A second source for load data is values in registers. Every value that is stored to memory by the CPU was stored there as a register store. That value may still persist in a physical register. This is again a matching problem – how can we match a load (that will ultimately use a particular address) with the physical register that was earlier stored to that address?

A different way to think about this is consider the sequence
define x0
```
STR x0, [x2]
```
do some stuff
```
LD x1, [x2]
```
use x1
What we want to do, effectively, is remap the store register, x0, to the load register, x1, bypassing/augmenting the actual store to DRAM.
For this reason the idea is called Memory Renaming (by analogy with Register Renaming), 1997, https://web.eecs.umich.edu/~taustin/papers/MICRO30-mren.pdf, *Improving the Accuracy and Performance of Memory Communication through Renaming*.
In spite of the idea having been around for so long, and in spite of Apple having multiple patents in this space going back to 2012, I have to tell you (spoiler alert) that I could find no evidence for any of these techniques present in the M1. Perhaps they were present in earlier cores and considered unnecessary on modern designs? Perhaps some detail of how they handled misprediction no longer works on the

M1?
Regardless, let’s look at them because, presumably at some point they will (re?)appear.

Let's start by considering the simplest possible version of a solution, then improving it. So we start with simple stores (a single register) using simple address modes (a single base pointer, no index or immediate offset).
Suppose we store in a table that base pointer x2 stored physical register p100. There will be multiple such entries, in principle one for every possible xn base pointer.
Now suppose I execute `LDR x10, [x2]` (which is known at Decode time, unlike the load address). I look up my table, see that the appropriate value is p100, and so I rename x10 to p100, and mark x10 as already valid, so it can be used by any other instructions.
In other words, just like Zero Cycle Move, the work is done at Rename, and this acts as a Zero Cycle Load.
(Note that this implementation stores the value to be forwarded from the store to the load via the register file, as opposed to the initial paper which suggested storing it in separate storage).
This design is covered in 2012 https://patents.google.com/patent/US9996348B2 *Zero cycle load*, where they call the idea the RF-LSDP (Register File Load Store Dependency Predictor).

So what can go wrong?
Suppose I change the value of x2. That's easy to catch, and I simply mark the entry in my table invalid.
I need to track if p100 is overwritten, but that's also easy and handled in the same way.
What if the load accesses the effective address of x2 via some different combination of registers, eg [x2]=[x3, #64]. Well, in that case we will not get a match and we will not get a Zero Cycle Load. But nothing will go wrong. Realistically, high level code uses pointers (or the equivalents of pointers, even if they are not language visible) for loads and stores, and the same register will be used to write or read from a common structure under most conditions. So this is not a serious concern.

The real concern is that a different store, using eg address [x3, #64] will overwrite the value stored at [x2]. Or even another CPU might overwrite it if the address is somehow being shared by more than one core.
This tells us that, whatever we do to accelerate the load, we will also have to validate it. I have some thoughts on that (building on the existing Replay mechanism), but given that the mechanism doesn't even appear to be present, these speculations seem pointless for now.

2018 patent (extend to load after load, extend to stack pointer)

So at this point we understand the basic idea of the most basic sort of ZCL load accelerator. How could we improve it?
In the initial patent Apple provided two linked upgrades:
- they allow addresses that are not just a base pointer, but also a base pointer+immediate, ie [x2, #64]
- they also track addition or subtraction of immediates to the base pointer.
So suppose the table records

[x2, #64] → p100; and now we add 10 to x2. Then, in essence, the table entry is updated to
[x2, #54] → p100.

These two changes mean that
- a much wider range of stores will be eligible for being captured in our ZCL table
- and entries can persist after common modifications to base pointers (like incrementing them to walk along an array)
But of course they also make the implementation of the table a little harder! Apple don't describe how they implement the ZCL lookup table, or how they track which entries to update when a base pointer is added to or subtracted from.

Note that this mechanism as described does not handle load/store pair, and Apple are unclear about whether it handles FP/SIMD registers (ie can the entry for [x2, #64] point to say f100, a floating point physical register rather than only integer physical registers?

Apple made one neat improvement to this mechanism a few years later when someone realized that stores aren't the only way a register is attached to an address; this is also true for loads. If a few cycles ago I loaded p100 from [x2] and then I load from [x2] again, I can use the ZCL mechanism to return p100 as the destination register of the second load. Ideally code is not constantly re-loading from the same address! But realistically this is common in various scenarios, including non-optimized code (eg when debugging) and code generated by JITs.
This idea is covered in 2018 https://patents.google.com/patent/US10838729B1 *System and method for predicting memory dependence when a source register of a push instruction matches the destination register of a pop instruction*.

However, as the name of the above patent suggests, it is primarily concerned with stacks.
Now that we have this idea of reading from registers, is there any other way we can usefully associate a register (loaded or stored) with some sort of pattern that is known at Rename time? Apple provide a second variant that behaves something like the stack engine of x86 designs.
The idea is essentially the same as the previous RF-LSDP, but specialized to the case where the base pointer is the stack pointer. This specialization, called the SP-LSDP (Stack Pointer LSDP) allows the mechanism to capture load/store pairs, and to be an especially good fit to function prolog/epilogs, hence making function calls even lighter weight.

Both the RF- and SP-LSDPs are nice zero-cycle accelerators, but they are also both clearly not yet optimal and one can imagine a variety of ways to improve them. For example if you read the patents you will see that entries in both predictors only last until the "register-producing" instruction. a load or a store, is retired.
It's obvious why this is (after retirement the register is freed, so it's value could then change at any point) but this is clearly not optimal.

One could imagine a restructured set of predictors taking the best of zero cycle moves, zero cycle immediates, and zero cycle loads (SP and RF), perhaps even some other cases (value prediction?), and

based on a mechanism whereby predictor entries remained valid until at least a physical register was actually overwritten, rather than being cancelled when the physical register is freed.
Likewise one would want to tweak all the predictors (not just SP-LSDP) to behave appropriately with load/store pair and fp/SIMD registers.

BTW patents comes with another nice pipeline diagram showing how some of this fits together. Note the RF-LSDP kicks in at Decode, the SP-LSDP one cycle later at mapping, and the traditional LSDP at Dispatch.
(Note that this is the 2nd gen A7-class pipeline; it's clearly not the M1 pipeline because it still references the RDA [Register Duplicate Array] as the mechanism for handling multiple references to a register, not the current 2019 mechanism I described earlier.)

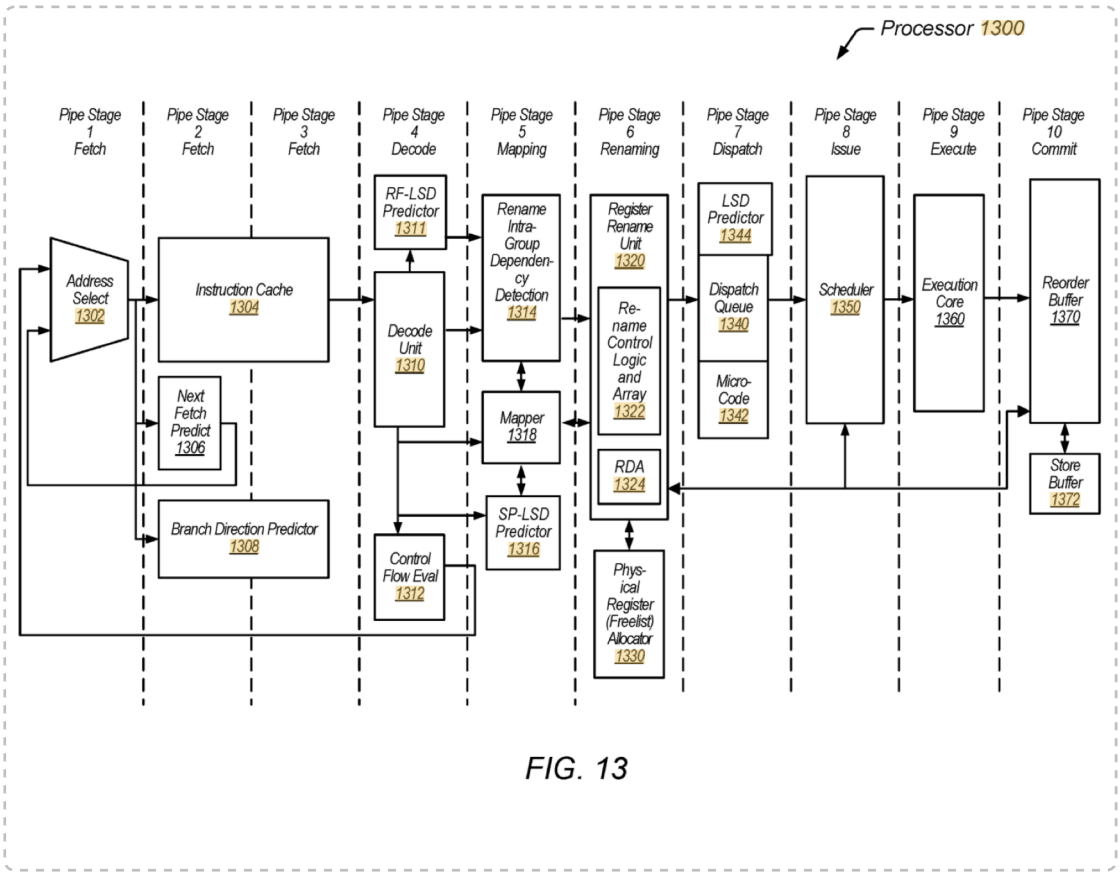


FIG. 13

2019 patent (store and dependent load in same decode group)

Do the RF and SP-LSDPs cover everything? Of course not! There is a technical detail in how they are implemented that means that they both can only cover a producer operation decoded in one cycle that feeds a load in a subsequent cycle. But what if you have a store followed immediately, or just one or two instructions later, by a load, so the store and load are decoded in the same cycle?
Who would write such dumb code, storing a value then immediately reloading it? Well, the example Apple suggest is interpreters in general, and JS in particular, especially when running for small code

sections that haven't yet been aggressively JIT'd.
For these cases Apple has (2019) https://patents.google.com/patent/US20210173654A1 *Zero cycle load bypass*.
This adds that as part of Decode, every appropriate possible pair of store/loads in a decode group (ie all the instructions decoded in a single cycle) is tested for matching address patterns, and if so we use the usual Rename trick.

As I said, this is a remarkable set of patents that all seem to build upon each other, and that include what at least look like careful thought as to how to fit the ideas into an existing design. But zero evidence than any of them are implemented!

Acceleration via Faster Address Calculation

The mechanisms discussed so far make loads faster by retrieving data from unexpected places (the Store Queue or the Register File).
A second way we might make loads faster is if we can more rapidly construct the address used by the load.

2013 patent (pointer chasing)

Our first technique ties both these ideas together, 2013 https://patents.google.com/patent/US9116817B2 *Pointer chasing prediction*.
Consider pointer chasing code, ie code that looks like node->node->node->data, which will compile to something like
LDR x3, [x2]
LDR x4, [x3]
LDR x5, [x4, #8]
Performance is limited by how fast the address of the next load is acquired by the previous node.

Consider the scheduling of the second load. We want to send the load to the LSU at the earliest possible moment that its dependencies are satisfied (ie it has access to the value of x3) but no earlier.
The most cautious solution is to wait till the value of x3 is in a register, read it from a register, and base the scheduling on that. That makes sense if a physical register, in this case for x3, is the *only* available source for a value (eg for values that were calculated many many cycles ago) but not otherwise.

A better solution is to pull the value of x1 off the bypass bus. Remember that there's a (very wide!) bus connecting every execution unit with the register file to transfer calculated values to their destination register. If these values are appropriately tagged, and the scheduler appropriately matches tags, it can grab the values required for an upcoming instruction off the bus at the moment it's needed.

But that's still not optimal in the pointer chasing case! The fundamental insight is that there is one

more place where data could be made available to the load – internal to the LSU.
The value is available in the LSU, but then has to be transported to the bus, moved over the bus, read, then transported back to the LSU.
So even better is to guess that the value will be available in the LSU at the appropriate time, and simply fire the second load into the LSU. If everything works out correctly, the request to execute the second load (which begins with an adder in the LSU that adds an offset, in this case 0, to the value of x3) will arrive at the adder just as the value of x3 arrives at the adder, fresh from the L1 cache; and the secondary load begins its execution one cycle earlier than it would have been if it had to wait for the value of x1 on the bypass bus.

Presumably the fundamental scheduling trick here (although Apple gives no details) is to track in the LS Scheduler that the three most recent loads are each generating register pA, pA, and pC; and to test if any of those three registers match the base register of the next load; if so the load can schedule a cycle earlier than normal.

A year later in 2014 (so now A7 generation) https://patents.google.com/patent/US9710268B2 *Reducing latency for pointer chasing loads* we we improve this in a few simple ways
- we also allow a store that's dependent on an immediately prior address load to use the mechanism. Not that essential for performance, but easy, so why not?

- we now understand that we are using the LSDP in this additional (pointer chasing) way and we allow pointer-chasing failure cases to also train the LSDP predictor
We also get this nice LSU pipeline diagram that helps clarify timing of the different components (remember, again, this is A7 generation; M1 certainly differs in some details because it takes one cycle less).

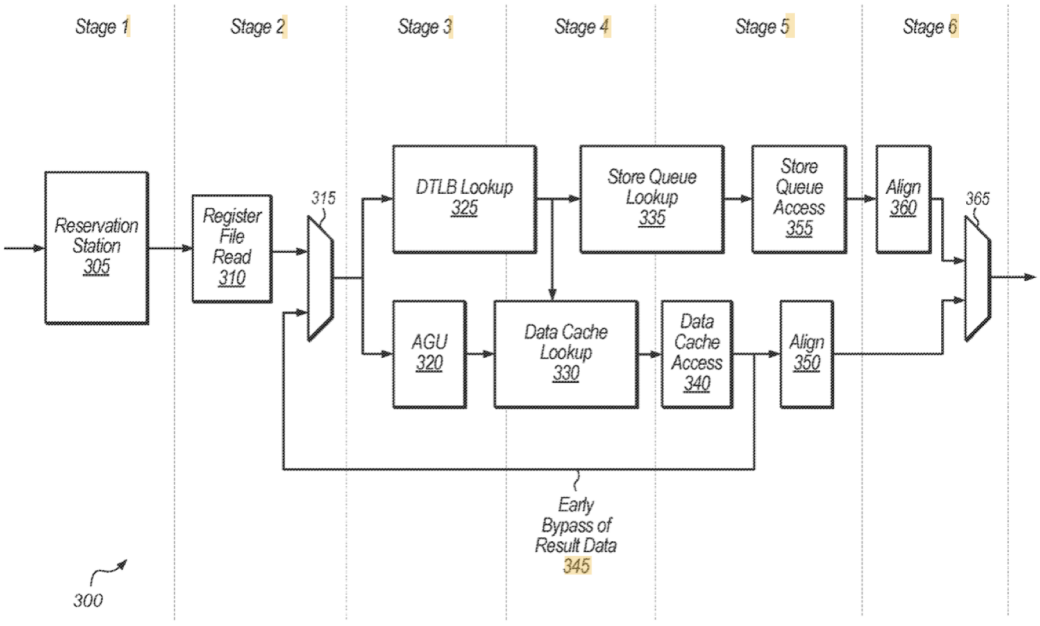


FIG. 3

Good news! This is the one patent that is implemented. Though not perfectly.
As far as I can tell this particular scheme does not yet work (as of M1) with `LDP`. You might think this is an unimportant detail, but it's not uncommon to have a pointer chasing loop that loads, from each node, the next node pointer and some node payload data to be utilized. This pair of loads would naturally be coalesced by the compiler to a `LDP`, at which point you lose the optimization 😢.

The M1 version of this accelerator does also work with the first register (base pointer) of indexed loads, like `LDR x0, [xBase, xIndex]` but not with the index. If you're writing manual assembly (or a compiler), and you are loading the index just before the lookup, (this can happen, for example, in certain types of sparse array lookups), then if it's feasible (no shifting of the index required) be sure that you swap the order of xBase and xIndex in the `LDR`.

2017 patent (strided load address prediction + pre-execution)

A second way we might know the address faster is to guess! In other words, once again, we use a predictor. This class of predictor is called a *Value Predictor*, more precisely an *Address Predictor*. A recent discussion of the idea is here (2017) https://www.researchgate.net/profile/Rami-Sheikh/publication/318283615_Load_Value_Prediction_via_Path-based_Address_Prediction_Avoiding_Mispredictions_due_to_Conflicting_Stores/links/59b26eea0f7e9b37434e7036/Load-Value-Prediction-via-Path-based-Address-Prediction-Avoiding-Mispredictions-due-to-Conflicting-Stores.pdf *Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions due to Conflicting Stores*.

Apple's idea, (2019) https://patents.google.com/patent/US20210049015A1 *Early load execution via constant address and stride prediction,* is to track the address that is generated by any particular load (ie note the stream of addresses generated by a load with a particular PC, presumably in a loop). If this stream of addresses forms a linear sequence then we can reasonably predict what the next address will be. And once we know what that next address will be we can
- execute the load early behind the scenes
- record the value in a physical register
- rename the destination register of the load at Rename (the usual ZCL)
- validate by re-executing the load at the correct time, based on the actual values at the correct time.
(Do we have to re-execute the load? In principle perhaps we can use the Load Store Queue mechanisms and the Poison mechanism to detect if the load address has had its value changed. If that's possible, then all we need to do is validate that the load address matches our speculated load address.)

This may sound like a standard stride prefetcher, but remember the prefetcher gets data into the cache early; we are interested in the next level of getting data from the cache into a register early.

If you looked at the previous diagram for the RF/SP-LSDP pipelines, you saw that, for whatever reason, Apple wants the different predictors to run at different pipeline stages.
They're close to running out of stages at this point, but, no fear! We can move this prediction all the way back to where the Fetch Address is predicted!

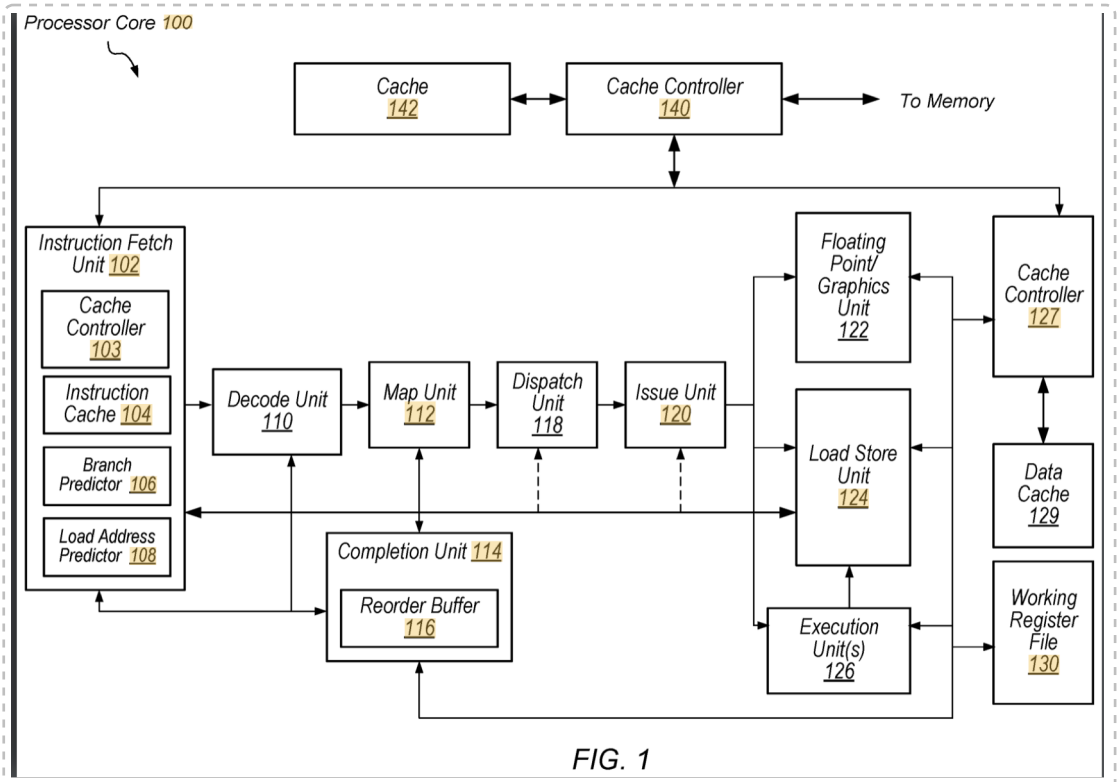


FIG. 1

Experiments

testing pointer chasing (success)

The easiest case to test is pointer chasing loads.
Initialize with
`STR x2, [x2]`, and `MOV x0, #0`;

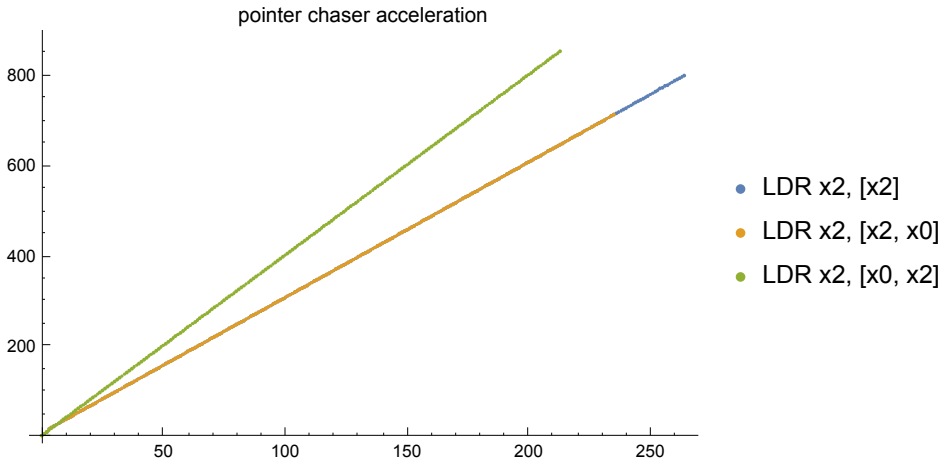
then compare repeats of:
`LDR x2, [x2]`
with
`LDR x2, [x2, #0]`, `LDR x2, [x2, x0]` and `LDR x2, [x0, x2]`

All of these perform the same task of repeatedly following a pointer (that points to itself). But some versions of the instruction are (we believe, from the Apple patent) susceptible to pointer chasing acceleration and should take three rather than four cycles per iteration.

The second case (adding an immediate to the basePtr) is not completely trivial to code because an immediate of 0, as I have written it, encodes to `LDR x2, [x2]`

To test an offset I'd have to use a more elaborate setup that pre-initialized an entire array of pointers. But I'm pretty confident what the result would be; if you don't believe me do the test yourself!

Out[208]=



We see the accelerator in action. Pointer chasing that feeds the pointer into the first argument (the base pointer) runs at 3 cycles per load, as opposed to normal load loops (and pointer chasing done by someone who doesn't understand ARMv8 assembly) which takes 4 cycles per load.

testing ZCL's (abject failure)

That's fine, but testing ZCL's is much less obvious!
The primary problem is that the machine is so complex! You have to be very careful to
- eliminate all other ways that code can run fast (OoO and very deep queues)
and confounders like
- the stores still have to happen, and the loads still have to be validated, so if these are too close together the validation steps are what lands up slowing you down, or you land triggering Replays.

When I first read the various ZCL patents I was very excited, tried some tests, and believed I had validated their existence. But after every test, I would think up some way in which the machine could have achieved the speedup via some other OoO mechanism, or I found myself drowning in a sea of uncertainty about exactly how timing worked inside the LSU for various special cases like Replays or stores and loads to the same address that issued in the same cycle.

I'll omit most of the experiments, leaving only the last because I think they are both definitive in showing no ZCL, and they show something else interesting.

Our problem is that the machine can do so many things in parallel, *but it is finite*.
Suppose that I create a block of 100 successive EOR x0, x0, x0. This will take 100 cycles. If chain two of them, 200 cycles, etc. No surprise.
Now suppose I change the code to
`LDR x0, [x2]`
Repeat (100x) `EOR x0, x0, x0`.

Note that the Load *breaks* the dependency between the successive chains.

If we had an infinitely sized machine, we could queue up (`LDR 200x EOR`) then a second (`LDR 200x EOR`) and run them in parallel, for an average throughput of two blocks per cycle.
But in reality queues have a finite size.
So suppose the size of the queue that can hold `EOR`'s is 50. Then as execution proceeds:
- the load happens
- the queue fills up with `EOR`s (at 8 per cycle)
- the `EOR`s execute (at one per cycle)
- when the queue is full everything pauses at Rename and one `EOR` per cycle moves into the queue.
- *in particular* the second load is blocked behind a whole lot of `EOR`'s, so it cannot start the next chain until after 150 cycles or so.
- in other words (with the limits I have given) we can get some parallelism, but not full parallelism.
In other words
- 150 `EOR` instructions run one-wide; while 50 `EOR`'s are stuck in the Scheduling Queue.
- The next load can execute; at this point the value of x0 beginning the next `EOR` chain is known
- the last 50 instruction of the previous block and the next 50 instructions of the new block can now run in parallel two-wide
- when those are done, we are back to one-wide for 100 `EOR`s, with 50 queued, then repeat.

So long term what we get is approximately:
2-wide parallelism for (size of the queue) and 1-wide parallelism for (size of block - size of queue).
Or to put it differently, rather than 200 `EOR`s taking 200 cycles, they will take 100 cycles (1-way parallel) + 50 cycles (2-way parallel)
Likewise 300 `EOR`s will take 250 cycles. Etc.

This is, in fact, what we see.

Out[210]//TableForm=

100	42
200	117
300	215
400	315
500	415
600	515
700	615
800	715

The "effective queue size" for the EOR's is about 85 operations.
We can perform variants like
`ADC x0, x0, x0` (which can only three of the six integer execution units) and which gives an effective size of 35 operations,
or (`MSR NZCV, x0; MRS x0, NZCV`) which only uses two units and gives an effective size of 26.

For purposes of measuring Scheduling Queue size we have the complication that Queues can, as we have seen, be cross-connected so that one Queue can feed two units, if the primary feed queue for unit has no entries. It's possible that this effect confused Dougall's automated attempts to measure the queue sizes, so each of his integer and FP queues was measured as ~twice as large as it is?
This would go a long way to reconciling these rough heuristic numbers (which otherwise seem much too small) with his numbers.

OK, so we have a theory about how long chains of instructions behave, either with or without an initial dependency-breaking load at the beginning of the chain. Now suppose we change the code to
`STR x5, [x2]`
`LDR x0, [x2]`
Repeat (100x, or 200x, or 300x or whatever) `EOR x0, x0, x0`.
Note that
- we are adding one additional instruction at the beginning of the chain
- we are not storing x0, we are storing x5 (so each chain begins fresh with x5, there is no carried dependency)
- we only have one store and load. No matter how small and limited our various prediction structures are (LSDP, ZCL, whatever) you'd hope they can handle *this* situation!
- we have a massive amount of time after the load/store pair for these two to whatever cleanup they want after initial execution (storing to cache and whatever)
So what do we see?

Out[212]//TableForm=

100	40
200	115
300	213
400	314
500	414
600	514
700	614
800	714

In every case the timings go down by one or two cycles. Once again this is absolutely repeatable, there is no noise in these numbers.
This suggests that there is something happening with ZCL. The reduction is not much, a cycle, but seems to be real.
But don't get too excited! This is precisely the sort of godawful *almost* signal that appears in every test I tried.

Let's change the code slightly.
`STR x5, [x2]`
`LDR x0, [x2, x9]` (with x9 initialized to 0)
Repeat (100x, or 200x, or 300x or whatever) `EOR x0, x0, x0`.
This should not hook into the ZCL mechanism (which is not supposed to handle an index), but we get

the same results.
But maybe it is hooking into the stride prediction/early load mechanism with a stride of 0?
OK, so we add some code to increment x9 by x4. (x4 is the loop counter so it changes by 1 every cycle), so end result is x9 forms a quadratic (not a linear) sequence. Still same results, but clearly a stride predictor will not capture our sequence of loads.

So why does adding the Store speed the code up by one, even two cycles? Who knows? There are so many moving pieces involved...

Final attempt was to begin each block with an `ISB` instruction. ISB is a synchronizing instruction that ensures that after the ISB, every subsequent instruction is refetched from the I-cache. It's meant to be used after situations like when you change the permissions of a page table, so that every instruction executed after the page table change sees the new permission; there are no instructions queued up in various places that were loaded earlier and slip past the new permissions.

If we use the basic loop with only EOR's and add the ISB, we take 826 cycles (for 800 successive EORs).
This suggests the cost of a flush (ie basically restarting the pipeline) is 26 cycles, closely matching our estimate when we investigate the LSDP and flushes caused by a load/store ordering failure.
If we then add in the load, the time rises to 830 cycles (four additional cycles for the load), no surprise.
If we then add in the store before the load, the timing does not change, same 830 cycles.
This suggests that indeed the store and load are matching their timing precisely in the LSU.
But it *also* suggests that there is no zero-cycle load happening by having the store'd physical register renamed to the load's physical register (which should have given us a reductio in time, ideally all the way back to 826 cycles).

So my best guess (I'd be thrilled f someone proved me wrong) is that
- the LSDP and Replay all work well and as we have suggested
- pointer chasing works (though could be improved to handle both pairs and "dumb" indexed addressing)
- all the ZCL variants (RF-LSDP, SP-LSDP, strided address prediction) are not present. To be honest I did not test SP-LSDP, by that point I was so depressed at the lack of any clear proof no matter what I tried, that I figured it would just be a waste of time to even try figuring out a test.

Some aspects of the L1D cache

Bandwidth and latency basics

Before we start the serious investigation, let's get a few simple tests out the way.
You should know by now that we can perform 3 loads per cycle. What if we want to maximize load throughput?

The best solution appears to be a loop that uses LDP x0, [x2, #16] or LDP x0, [x2], 16, ie one of the autoincrement modes.These can run at essentially three operations per cycle. Any alternative like manually incrementing the x2 register will introduce a chain dependency (tremendous slowdown) and if you try to use multiple address registers which you manually increment, you won't do better than the autoincrement modes, while writing a lot more code and using a lot of temporary registers.

So using this optimal form, within L1 we can load 48B/cycle. There are various wider load instructions, eg LDP of vector registers, or LD4. But the width from L1D to the LSU for each load appears to be 128b, so something like a vector LDP takes two cycles for the data transfer. My guess is widening this bus to 256b is something we will see as part of SVE.
So, for now, the maximum load throughput from L1D is 48B*3.2GHz=153.6GB/s.
We can now test various types of either load latency (from beyond the L1D) or throughput (from beyond the L1D). The numbers you will see depend in detail on exactly what your test code does (is it a simple stream that can activate prefetchers for the the caches and TLB, vs does it bounch around randomly?)
If that's what you want to know, the best resource is to read up Andrei's summaries
for the A13 (which gives bandwidth numbers, which you should be able to project via commonsense extrapolation to M1, remembering that the M1's DRAM bandwidth is about twice the A13's bandwidth.), and latency numbers, all for a variety of patterns.
A13: https://www.anandtech.com/show/14892/the-apple-iphone-11-pro-and-max-review/3
M1: https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2

For at least some definitions of "random lookup that mostly misses to DRAM for every load", M1 appears to provide a bandwidth of around 64GB/s. This is remarkably good compared to other CPUs (look at eg the i9 results, and remember that M1 has double the DRAM bandwidth of A13).
 Apart from things like the tight integration of the DRAM with the SoC (lower power per bit) and smart frequency variation of the DRAM (run it slower when high memory bandwidth is not required) at least part of this success is probably due to the SLC.

The SLC is sometimes referred to as the M1's L3, but while true'ish that misses some important points. One is that it's a system cache; so its primary jobs are
- to facilitate communication between blocks on the SoC, eg memory that has been worked on by the CPU and cast out of L2 into SLC can then be transferred to the GPU
- to save energy by holding onto these sorts of inter-block transfers rather than paying the cost of moving them out to DRAM.
But just as interesting is the fact that it's a *memory-side* cache. In other words you can think of it as being a cache that is tightly associated with the DRAM (and more specifically with the DRAM controller), rather than with the CPUs or any other block on the SoC. This tight integration with the memory controller provide a few benefits. Two are
- streaming reads or writes (detected by the prefetcher or the CPU or even indicated by a DMA engine) can bypass the SLC to go straight to the NoC and thence to their target device. And the memory con-

troller can know that these are streaming and behave appropriately in terms of the DRAM page opening and closing policy.
- the SLC can be used as a *virtual write queue*. Rather than the controller having just a smallish write queue and being forced to write to DRAM when that queue hits a high-water mark; the memory controller can treat the whole of the SLC as a write queue, choosing to buffer writes and not write for a very long time as it services a long stream of requests, then switch to writing out a long stream of writes when there's a let-up in the reads. This reduces time lost to switching between DRAM read and write modes.
This is described in (2010) https://lca.ece.utexas.edu/pubs/ISCA_2010.pdf *The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies,* and is known to be present in the IBM POWER8 (and presumably successors, so maybe also recent z/ series).

It's probably also the case that the tight coupling between the DRAM and the SoC allows better characterization of the minimum number of refreshes required, so less time is also spent on refresh. The details of this (and many other ideas for how to improve DRAM) are associated with the name Onur Mutlu; you can find a biography here: http://people.inf.ethz.ch/omutlu/projects.htm, for example (2018) http://people.inf.ethz.ch/omutlu/pub/VRL-DRAM_reduced-refresh-latency_dac18.pdf *VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency*.

Introduction

Be warned that of all my investigations this was the most difficult, in terms of understanding the data,and in terms of trying to find useful guidance either in the literature on in Apple patents. This material starts to get very low-level, and no-one seems very interested in discussing it publicly. Even amongst the competition (eg Intel or IBM) useful explanations ended about fifteen years ago.
Meaning a lot of this is conjecture and best guess; I'd be happy to be corrected if anyone knows better.

On the one hand we know, from the general tables of M1 instruction latency/throughput that M1 can sustain 3 loads tor two stores per cycle. But how exactly is that split?
First consider just the type of operation. Then the best way to summarize is something like there are
- 2 load-only pipelines
- 1 store-only pipeline
- 1 ambidextrous pipeline
This implies that we should be able to run 3 loads+1 store, or 2 loads+2stores in one cycle, and we can; but we can't, for example, run 3 loads and 2 stores in a cycle.
This compares favorably with Ice Lake, which can handle 2 loads and 2 stores per cycle, (but cannot toggle to 3+1).

However, unlike an ALU where simply comparing "I have four adders, you have six" tells you almost everything useful, there is a massive amount of detail below how these 3+1 or 2+2 instructions are executed which in turn has massive implications for performance and power.

I'm going to assume right away that you know basic cache issues and terminology - sets, ways, tags, way-prediction, the role of the TLB, etc.
If you don't, then read something like https://web.eecs.umich.edu/~twenisch/470_F07/lectures/13.pdf (for the basic terminology), then
http://web.eecs.umich.edu/~twenisch/470_F07/lectures/15.pdf (starting at page 11), for the issues related to any cache sustaining more than one load/store per cycle.

Experiments on the TLB

To get started, let's clarify the capabilities of the TLB. We will not explore some basic issues, like capacity, which have been covered by Andrei F at AnandTech). Rather we want to provide an initial base line for unexpected behavior, to show how what you thinj is obvious is not essential; and so to justify why we need detailed analysis, and how the TLB behavior matches the cache behavior.

Consider the three probes below:

```
LDR [x0]; LDR [x0+8]; LDR [x0+16]       1 probe per cycle
LDR [x0]; LDR [x0+8]; LDR [x0+16K]      2 probes per 2.8 cycles (better than 2 per 3 cycles...)
LDR [x0]; LDR [x0+16K]; LDR [x0+32K]    2 probes per 5 cycles (better than 2 per 6 cycles...)
```

For all of these I don't indicate the destination register, it's not relevant. If you care, jut assume every LDR deposits the result in x10.
Also note that the x0 is not updated, we are not probing what happens when you exceed L1D or TLB that's a different concern.

The first probe is as trivial as it gets. Three loads, all on the same page, all on the same cache line. Unsurprisingly it takes one cycle.

The next probe results are perhaps unexpected if you are used to most CPUs. Sure, the probe requires different pages from the TLB, but aren't TLB's always multi-ported to as many load/store instructions as can occur per cycle? How else could the design work?

The second probe's result looks very much like the machine can "provide" one TLB lookup per cycle, but that TLB lookup result can service multiple requestors that all have the same page number.
This concept is known as a "piggyback" port – a single lookup is made into a data structure, but the result is then sent to two (or three, or four, different requestors). It has been part of the academic literature since at least the late 90s, but I'm unaware of it being used by anyone but Apple.

So it looks like the timing is essentially (after the system stabilizes)

```
LDR [x0], LDR [x0+8]          1st cycle
LDR [x0], LDR [x0+8]          2nd cycle
LDR [x0+16K], LDR [x0+16K] 3rd cycle
```

The model I am suggesting is that load/store addresses are effectively placed in a few very short queues for TLB lookup, more or less sorted according to a common page, so that lookups that match a common page can be shared.
This could be as simple as something like
- 2 or (more likely) 4 queues, based on the low bits of the page number,
- each holding perhaps four entries,
- so that (assuming some degree of locality) even when we have loads bouncing between different pages like here, mostly the loads to a common page will be binned by the low-page-bits to a common queue, and can then be serviced together.
This in turn suggests that the TLB is, in fact, single-ported.

Note that if we were willing to aggregate over three cycles, we could service three of the `LDR [x0+16K]` at once (at the cost of some delay), and so run at 1 probe per cycle. But the first `LDR [x0+16K]` would then be delayed by two cycles rather than one. However the system seems to prioritize latency over throughput. A one cycle delay cannot be avoided (if we have made the choice of a single-ported TLB), but we can choose either
- service the (non-empty) queues by oldest first, or something that approximates like round-robin
- or delay servicing queues for a cycle or two to allow them possibly to fill up (higher throughput, but worse latency),
and Apple seems to be choosing the first.

The third case confirms this model. Now, in principle, even if we only wait for a queue occupancy of two, rather than three, we could split servicing across three cycles, something like

```
LDR [x0], LDR [x0]              1st cycle
LDR [x0+16K], LDR [x0+16K]      2nd cycle
LDR [x0+32K], LDR [x0+32K]      3rd cycle
```

And if we did this, we should still be able to service 2 probes in three cycles.
But if we are aggressively moving between the various (four?) queues that are holding these addresses, then the timing is such that we alternate between one and two entries in each queue that are serviced each cycle, so that the time for two probes (six loads) expands from three to five cycles
- on average the queue occupancy that is serviced is about 1+1/6, so mostly 1 but sometimes 2.

The same analysis then explains the second case? The second case is servicing 2.14 loads per cycle (6 loads/ 2.8 cycles). So the average queue occupancy must be just over two, usually two but sometimes three.

What if we modify the code slightly?
If, for example, we make each address auto-increment, something like `LDR [x10], #200` (with the

base registers x10, x11, x12 separated by either small values or page size)?
You might expect this to be the same or slightly worse (note that the increment of #200 is large enough to cover a page in 80 increments, but small enough that we don't leave the L1).
But this substantially improves things! Giving a throughput of very close one one cycle per three loads (whether we use just one load offset by 16K, or two, offset by 16K and 32K).
I think what's happening in this case is we are breaking up sub-optimal patterns so that the three loads are spread evenly over four(?) queues, serviced in a way that's now usually collecting three (actually about 2.75) loads in each queue. So slightly worse latency, but substantially better throughput.
There might be scope here for a design tweak that introduces a small amount of non-deterministic jitter into the order of servicing of the queues, to break up sub-optimal patterns that simple code can get locked into?

If we introduce stores, the TLB can now sustain eg 2 loads and two stores whose addresses are all in the same page. (There's something interesting happening here slightly above the TLB. Presumably what's delivered from the TLB over the piggyback ports is not just the physical page number but a set of permissions, which are then matched to the load or store at each piggyback port, so that store might cause a fault even if that same page allows reading.)
As soon as we introduce a single load (or store) to a different page, we can get the same phenomenon as in our second case, of a slow down because of alternating between servicing different queues , but not nearly as pathological as the earlier worst case we saw. Numbers vary slightly depending on the exact details of whether we are dealing with two, three, or four different pages, but we still get close to one cycle per quad of (two loads+two stores), with an average TLB servicing per cycle (ie average queue depth) of 2.6 to 2.7.
You'd hope this might rise to an average of 3.x instructions serviced per cycle (since we are trying to execute 4 load/store instructions per cycle).
This is achievable if you line up everything exactly correctly, so that no later constraints down the pipeline will trip you. For example if you use the four offsets (0, 1, 2, 3)*(16*1024+128+16) you will sustain four operations per cycle. Over time we will see why these numbers were chosen. This means that the queues before the TLB must number at least four, each capable of holding at least four entries (and anything larger is probably pointless).

If you break up the perfect symmetry then, we revert to the sub-optimal case. One way this can happen is if you use three loads and one store; I assume this has to do with
- loads and stores are placed in both the store queue and ambidextrous queue
- since we can dispatch 8 per cycle but issue a maximum of four, the queues all fill up
- the ambidextrous queue holds some stores (though ideally it would not) just because of how the instructions were queued
- and that's enough to break up the perfect ordering (and the perfect scheduling of every step).
Another (minor) breakage is to auto-update each address by the same amount. This seems like it should still preserve perfect symmetry, but not quite. Auto-updating by #0 (so I guess just a slight rerouting of the address calculation) drops to us 1.07 cycles per quad (load/store pair). Still pretty

damn good, but no longer perfect.
Auto-updating by #200 (again not a random number; if you used #128 the results would be closer to #0) drops this to about 1.25 cycles per quad. Again still very good (most cycles we are processing all four elements of the quad!) but again not perfect.

What if we introduce a truly pathological stream, with a different page number for every accesses? (We only need to cover maybe 16 to 20 or so accesses to flood all the TLB queues, so we don't have to bust the TLB or cache and worry about those effects.)
Then we get a situation where no TLB lookup can be shared, and the number of accesses processed per cycle drops from the ideal of three or four down to to slightly below 1.
(The case I used for this to
- only use loads (we will see there are complications with stores able to delay, or to reuse load activity)
- have offsets of x10, x11, x12=(0, 1, 2)*(16*1024+128+16)
- have a basic block of

```
LDR x20, [x2, x10]
LDR x20, [x2, x10]
LDR x20, [x2, x10]
ADD x2, x2, #16384
```

repeated four times.
This will result in 3*4=12 different page accesses, in the inner loop; but we don't want to just keep incrementing x2 (by a large amount!) forever because that will exceed L1 and bring in other complications.
So after these four blocks, we terminate the probe with a

```
BIC x2, x2, 256K
```

which will essentially reset x2 after 256K. Note that 256K is 16*16K, so we actually run a little over 16 pages because of the mismatch between a block size of 12 pages, and a test only once per block. The actual number of pages at any given point of execution might be as high as 24 before we wrap.

With this structure (so if our model of essentially up to 16 queue slots in front of the TLB is correct) we should never be able to reuse a TLB lookup for two loads, and that's what we see. The time taken to perform 200 repeats (ie 200*4*3) loads is 2439, so basically 1 load/cycle plus noise.
If we drop the bit clear to 128K (so essentially now the number of pages in execution at any time varies between about 8 and 12 before we start repeating pages) then we immediately double our throughput: 200 repeats now take 1105 cycles, so slightly over 2 loads/cycle (ie average queue occupancy of *a load with the same page* [so that one TLB lookup can service multiple loads] or, if you prefer, number of loads serviced in a cycle, is just over 2.)

Another question we might ask is how Speculative Scheduling and Replay features in this. Presumably if we had immediately dependent instructions on the instructions that miss in TLB, we would have to Replay the Load and those instructions to get them to mutually match timing. Fortunately if there is no Speculative Scheduling (as in this case), we only have to eventually dump the result in a physical register, and if that register write is delayed by a cycle or two, no big deal.
My guess is that Apple can also be much more tolerant about Speculative Scheduling. The paper I listed discussed this in the context of four cycles from Issue to Execute, simply to move data across the chip. Given

that Apple is not pushing extreme GHz, they probably only need to schedule a cycle or two ahead, and this means they may be able to do so after perhaps after TLB lookup (and even some of cache scheduling, in particular tag lookup) so avoiding most cases where this queueing might introduce a few cycles delay

Bear this all in mind as we go forward. We'll see a much more complicated version of the same idea when it comes to the cache, along with an explanation of the underlying design principle.
Basically Apple has managed to provide almost all the performance of a 4-ported cache and TLB, while paying the power and area costs of a single-ported cache and TLB! This is a neat trick!! It works because
- most loads and stores demonstrate strong locality (both in the TLB, and then in the same cache line)
- use of queues before the TLB to aggregate multiple requests to the same page
- the cases where this scheme will introduce a cycle or two of latency are generally code simultaneously streaming over multiple large arrays or walking down multiple large pointer-based structures, where an extra cycle or two of latency is nothing compared to the latencies induced by the memory lookups (missing at least to L2). For most code it's almost pure win.

Is there any evidence for these ideas (beyond these experiments)?
I've found one paper (2013) https://eprints.soton.ac.uk/347147/1/__userfiles.soton.ac.uk_Users_spd _mydesktop _MALEC.pdf *MALEC: A Multiple Access Low Energy Cache*, which talks about a full design using these sorts of ideas, and it's confident that a single-lookup TLB can service up to a 6-wide LSU without losing much performance.

From the LSU, past the TLB, to the cache

Let's start with some basic, simple numbers, to show the amount of complexity, and hence to justify the explanation. Once we have the explanation, we'll use that to investigate other aspects of the LSU's interaction with the cache.
It's natural to start experimenting with the widest loads possible because hitting maximum bandwidth is sexy. But that's a bad experimental approach because it becomes difficult to disentangle addressing constraints (how many different cache line addresses can the cache exploit per cycle) from bandwidth constraints (how wide is the data bus from the cache to the LSU). So we start by limiting ourselves to loading bytes, nothing wider.

Next, to design our probes we need to know some basic facts about the M1 L1D cache. These facts are established by other techniques (including Apple telling us!) so we won't bother exploring them:
- the page size (under normal conditions) is 16kB
- the L1D size is 128kB

- the cache line length is 128B. But be careful here! What exactly do we mean by this?
The most fundamental meaning for a cache line is the number of bytes that are serviced by a single tag. This may differ from other aspects of cache line behavior.
For example:
+ the basic cache line has a few flags associated with it (most obviously a Valid flags and a Modified flag).

+ But a cache line can be sub-sectored, with separate copies of these two flags associated with the upper and lower halves of the line. This allows obvious bandwidth reductions like only writing out half the line if the other half is not modified, or only loading in half a line at a time.
Some quick tests I did (based on latency, not bandwidth, and rather rough and ready) suggest that the cache line is loaded in half a line at a time, so that it's sub-sectored at 64B granularity, certainly for the purposes of loading from L2, probably also for purposes of writing out changes. XXX This should be investigated more closely.

- it's assumed that the L1D is 8-way set-associative. I haven't validated that, but everything I saw is consistent with that.

Given all the above, here's what we will do:
We first initialize three base pointers x0, x1, x2. These will all point into a large buffer so that loads are safe without any protection issues, and x0 is always maximally (16kB page) aligned.
We can set x1 and x2 with an offset relative to x0 to get relative offsets within a line or across lines.
Likewise we initialize common index x5 to 0.
Then we execute a stream of

```
LDRB w10, [x0, x5];    LDRB w11, [x1, x5];    LDRB w12, [x2, x5];
ADD x5, x5, #
```

where the offset # added to x5 will move us around the address space.
(Note, if you are trying to reproducing this, that you have to be careful about how many iterations are in your inner loop. You never want your inner loop to spilling outside the L1D. This is easy to do once we start generating stress cases that limit all the loads to a single bank or a single set of the 8 sets.
Whenever the numbers seem to be much worse than you expect, that's the first thing to look at – have you inadvertently forced your "effective" cache size to much smaller than you expected, so mainly you are testing the cost of missing to L2?)
So with this framework, let's start by setting x1=x0+0, x2=x0+0. We will denote this by [0 0 0] meaning that within a single cache line each of our loads is loading byte 0.
We have spatial flexibility (to change x1 and x2 offsets within the 128B line) and we have temporal flexibility (how much we increment x0 and so how we move through the cache).

We start by setting x5 to 0. So, yes, we are reloading the exact same byte every cycle from the exact same address. This gives us a throughput of 3 loads per cycle (and honestly, we'd be rather worried if it didn't!)
This may seem uninteresting, but think what it means. We are hitting the exact same address (meaning same TLB lookup, tag lookup, line access, byte reads) three times per cycle.
Now if you're a SW person thinking about this vaguely, you will say so what? But if you are a hardware person this should put you on high alert.
Your primary takeaways from the next section, where we look at the theory of cache design, are:
- reading two (let alone three) simultaneous values from the “same” cache is *hard*. We can read multiple registers simultaneously from a register bank because registers banks are physically huge relative to the limited amount of storage they actually hold. That same multi-port technology is totally infeasi-

ble for caches because of area costs, because they would be horrible slow, and because they would burn immense power.
- we are replicating the same load over and over which, conceptually, means that each cycle each of three caches (the TLB, the tag storage, and the actual data storage) is generating three values – and these cannot be values coming from different banks because, by definition, we're forcing the same address and so the same bank!
So already this is non-obvious.
(I honestly don't know how Intel or a mainstream ARM core would respond to this test.
On the one hand it's dumb, not something worth optimizing for because no real code behaves this way.
On the other hand, it's an interesting start to a set of stress tests.)

So let's continue experimenting with same-line behavior while not incrementing x5.
Let's try [0 1 2] so now three different (but adjacent) bytes. This likewise runs at full speed, 3 per cycle.
What about [0 63 127] (so three bytes spread maximally over a single line). Again full speed.

Let's now try these same patterns with varying temporal locality. We'll start by increment x5 by 1.
(This is an increment of one byte, not one line!)
That doesn't seem very interesting but consider, eg [0 7 15]. Maybe those were all loaded in a single line access? Or maybe they were loaded one at a time and somehow cached in the LSU?
If we increment x5, then each cycle we're now loading three different bytes from the line, first [07 15], then [1 8 16] and so on. But we mostly stay within a single line, so TLB and tag access are not a complication; we are testing how data is extracted from the line while not varying either TLB or tag lookup behavior.

First [000]. So each cycle we are loading the same byte three times, but each cycle the address of that byte increase by one. Again full speed.
And once again no change as we use different byte access patterns within the line.

What if we increment x5 by 16 rather than 1? Unsurprisingly no change,
likewise for x5 increments of 32 or 64.
But at 128 something interesting happens... Now, each cycle, we are actually loading from at least two caches lines.
It's easiest to summarize the results in a table rather than keep giving textual explanations:

Out[216]//TableForm=

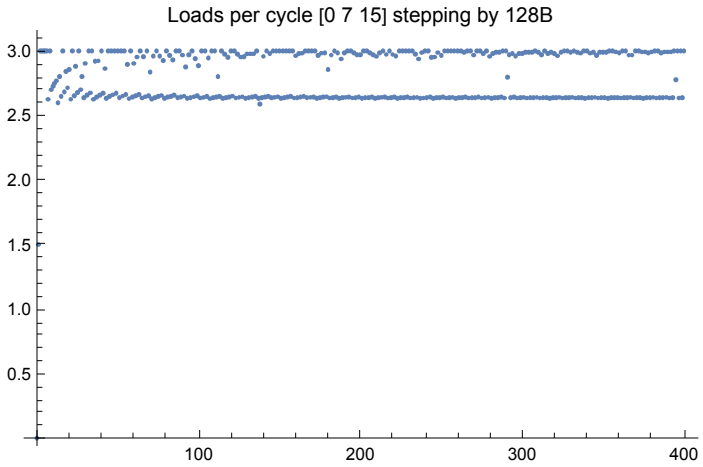
	[0 0 0]	[0 1 1]	[0 1 2]	[0 3 7]	[0 7 15]	[0 31 63]	[0
128*0=0	3	3	3	3	3	3	3
128*1=128	1.89	1.89	2.29	3.	3.	3.	3.
128*2=256	1.13	1.13	1.53	2.16	2.16	2.16	2.1
128*3=384	1.89	1.89	2.29	3.	3.	3.	3.
128*4=512	1.13	1.13	1.53	2.16	2.16	2.16	2.1
128*5=640	1.89	1.89	2.29	3.	3.	3.	3.
128*6=768	1.13	1.13	1.53	2.16	2.16	2.16	2.1
128*7=896	1.89	1.89	2.29	3.	3.	3.	3.
128*8=1024	1.13	1.13	1.53	2.16	2.16	2.16	2.1

Out[217]//TableForm=

	[0 64 128]	[0 65 130]	[0 129 258]
128*0=0	3	3	3
128*1=128	2.29	3.	
128*2=256	2.16	2.16	
128*3=384	2.23	3.	2.22
128*4=512	2.16	2.16	2.16
128*5=640	2.23	3.	2.16
128*6=768	2.16	2.16	2.16
128*7=896	2.23	3.	2.13
128*8=1024	2.16	2.16	2.16

Note that the cases marked as 3. (as opposed to 3), and meaning we can perform 3 loads/cycle, are in fact still somewhat variable depending on the precise number of loads in the inner loop; one sees results that look like this as one varies the exact number of load triplets in the inner loop:

Out[219]=



The most obvious pattern in this table is the difference between stepping by an odd number of cache lines vs an even number. (Recall that we have appropriately sized our inner loop so that this is not a number of sets issue! Yes, the even case restricts us to a smaller subset of the cache [either 4, 2, or even1 set, for incrementing by 256, 512, or 1024], but the question is why, even if we limit our activity to that smaller subset, do we get fewer loads/cycle).

There is a secondary pattern here whereby the loads from within the same twobyte ([000] and [011] are

slower than loads in the same word [012] which are slower than loads from different words (all the others).

Clearly caches are not as obvious as we expected! !!!! 😱!!!!
So our goal is to survey the issues involved in designing an L1D cache, and to understand (or, more accurately guess at/hypothesize!) as much of the M1 design as possible.

SRAM design

Consider the load pipeline. The classical version of the various steps is
1) construct the address (generally add a base pointer to a [possibly shifted by a small amount] index pointer)
2) compare that address with all the relevant (earlier) addresses in the store queue (virtual address compare)
3) look up that address in the TLB (based on address bits 14 and higher) to learn the physical pageID
4) use address bits 7..13 to form the setID, look that up in the tag store (which will be holding 8 physical pageIDs per setID)
5) precharge the eight lines of the setID
6) if one of 8 tags from step 4 matches the physicalPageID from step 3 that tells us the correct way (ie the lineID), and select the data from that line of the 8 lines of (5)
otherwise we have a cache miss

I have indented lines according to what can be performed in parallel.

This classical model is still what most people have in mind if you ask them to describe the L1D, even among supposedly informed individuals, but it has some major problems.
- it is power hungry, most significantly in step 5 which pumps current into eight lines, but only eventually cares about data from one of them
- it only supports one access (load or store) per cycle.

We can make this slightly more sophisticated and energy efficient by use of a way predictor, but before we get there, let's understand what all these different steps are doing and why they are required.

Going forward it's worth drawing a distinction between what we might call the cache at a logical level, and the cache at a physical level.
- Logical level concepts are n-way set associative, cache lines, and cache banks. You should know what n-way set associative means, and what cache lines mean.
- Physical concepts are arrays, sub-arrays, and words (or rows).
I bring this up now because there are very natural ways to map these concepts onto each other (you can get away with thinking a cache bank and a sub-array are "basically the same thing" for some time before getting terminally confused, likewise for a cache line and a cache row. But I want to reduce confusion, not increase it!

SRAM arrays

Baseline SRAM storage takes the form of a matrix of bits, threaded by horizontal word lines and (pairs of) vertical bit lines.
Some details here: http://users.ece.utexas.edu/~mcdermot/vlsi1/main/lectures/lecture_14.pdf and https://inst.eecs.berkeley.edu/~cs250/fa10/lectures/lec08.pdf are reasonable overviews.
You can look at the details of how an individual SRAM cell works if you like, but what matters for our purposes is how arrays of these cells work.
Going forward I'm going to be working through some examples. I urge you to actually draw the examples in your head, or on paper, whatever works for you; don't just skip over them at high speed. You will lose most of the value if you don't see for yourself how the numbers fit together.

The significant points for the standard SRAM model are:
- a "block" of SRAM (call it an array or sub-array) has a single set of address and data lines, an R/W line and maybe some other control+clock lines. It can support one operation (one read or write) per cycle
- it's usually close to square

- the bit-lines are maintained in a state of constant "tension" (ie they are pre-charged to a particular voltage) but (approximately) no current flows because access transistors present a high resistance.
This term pre-charge is unfortunate. It's correct in that this setting of voltage levels needs to be established before a read or write operation; but it's confusing in that (for the standard SRAM model) the restoration of correct voltage levels happens after a read or write operation, so can be thought of more a a "recharge" than as a "precharge".

- the baseline read operation reads an entire word (ie the entire width of an SRAM array). You can subsequently ignore whatever bits of this word you don't want, but as far as I can tell, you always pay the energy costs of reading an entire row.
This reading requires pumping electrons into the word line, an operation that requires energy and is called *activation*.

- reading takes time because it involves
+ "decoding" an n-bit address into one of 2^n lines (ie one of 2^n physical strips of metal), then
+ raising the voltage on that particular word line (ie activation), then
+ waiting for the voltages on every bit line to change slightly, then
+ amplifying those voltage changes to values for each bit in the selected row.

Writing is somewhat similar except that an additional line is involved for every bit that decides whether that bit will change or not. Speaking very roughly, the write energy costs are
+ we start like a read, with all the bits along the word line modifying their bit line voltages slightly, then
+for the selected bits that we want to modify,

+ we pay an additional energy cost by forcing the relevant bit lines to different values that reflect what we want to write.

- only one row of the array is ever activated at a time (which in turn means that you only get one read or write per cycle).

- what you probably want to do, unless you're creating a truly low-end system, is capture the data that was just presented on the bit lines in a given cycle into a row-width latch that can be read in the next cycle.
This allows you to pipelines the SRAM so that in any given cycle you are performing the operation of
+ "extract data from one row of the SRAM" and in the next cycle you are performing
+ "move that data off the SRAM; while simultaneously performing the activations of the next row that we want to access".
This means you have access to a different row every cycle but given an SRAM address, you don't get access to the data you want in the next cycle, but two cycles later.

We can add a very simple modification outside the SRAM array where we select for reading just the actual bytes we want out of the entire row returned.

So imagine a block of SRAM 128x128 (bits) wide.
We will interact with it via
- a 7 bit address bus (which will pass through a decoder to be turned into one of 128 vertical word activation lines)
- a data bus of the maximum width we wish to read or write in one operation (could be as high as 128 bits, but let's say it's 64 bits)
- some byte enables or whatever that tell us which specific bytes (or bits) of the 128 bits in a line are of interest
- some command signal lines
Operation will consist of sending the 7-bit address and a read-command then some time (a cycle or two) later using the byte enables to decide which of the 128 bits read from the SRAM to actually route to the 64 bits (or less) returned as the result of the load.
The point that matters is that the SRAM array has a minimum functionality width of 128 bits, and it's up to the rest of the cache or load-store unit to decide how and where to deal with this – one possibility is to throw away the excess bytes as close to the SRAM as possible, another possibility is to transport every bit read from the cache to the LSU and perform byte editing there.

The decoder is some logic that converts an n-bit address is one of 2^n lines. This logic is somewhat of a hassle, the more so the more address bits you have. For this reason (among others, like layout) SRAM blocks are generally fairly close to square (between about 1:1 and 2:1 aspect ratio).
Suppose you want storage very different from that, like you want to store 2048 rows each holding a word that's 16 bits wide. What you might do is "fold" this into an array that's 256 rows of 128 bits wide. For addressing, you

might do something like use the eight high bits to as the row select, and convert the 3 low bits to choose which 16 bits of the 8*16 bits that are stored in each row (ie you would treat those three low address bits as something like a byte enable).
This changes the problem of creating a decoder
- mapping 11 bits into one of 2048 output lines into two problems,
- one decoder mapping 8 bits into one of 256 output lines,
- the second mapping 3 bits into 8 output "lines" (which will each be split to active the 16 bits of the word of interest.
So we've converted a long skinny SRAM (usually more difficult to place nicely, and wasting a lot of logic in the decoder) into something that's more square shaped and has fewer transistors embedded in the decoder logic. Of course now we pay more energy in the word activation (each word is eight times wider) but we pay less energy in the voltage restoration of the bit lines after a read or write (each bit line is one eighth as high); and the physics of the problem mean that you may not be able to reliably detect the very small changes in bit line voltages once a bit-line crosses more than about 128..256 rows.

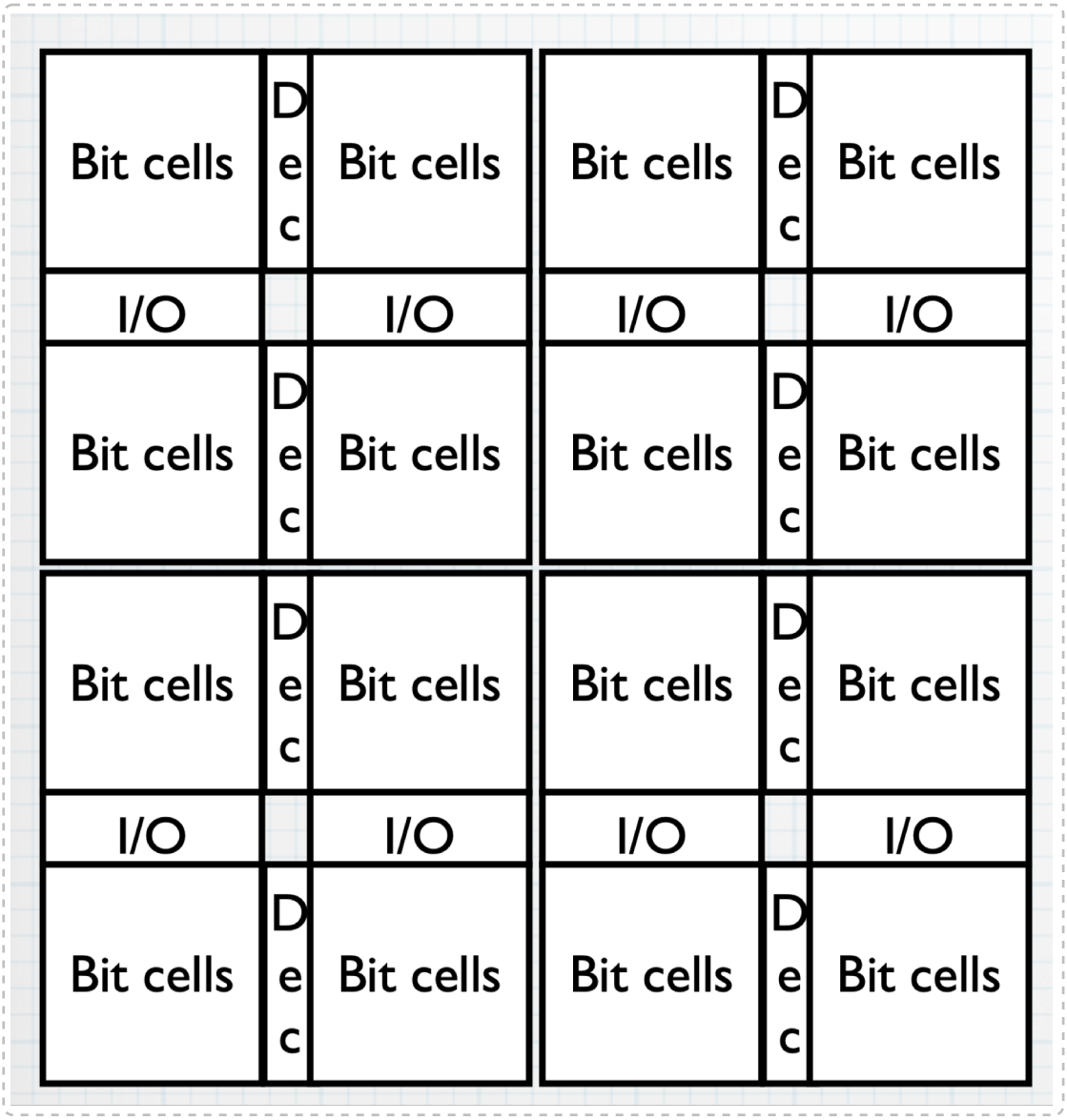
However I should note that the push for square arrays seems to be less important nowadays than it used to be. There's still a desire to limit large encoders (eg by folding, as described above) but beyond that people seem to be much less constrained now than they used to be in terms of SRAM aspect ratio. (Perhaps a consequence of smarter automated layout that can find a way to place rectangular SRAMs in a way that manual layout not good at?)

splitting an SRAM array into sub-arrays

So we have constraints on the height of an SRAM array and the width (as the array gets higher or wider it uses ever more power, and takes longer, for the word and bit lines to change their state so as to reflect the bit value read from the storage cell).

So an obvious next step might be to split this 128x128 array into four 64x64 sub-arrays. Since each column height is halved, it should take less energy and less time to activate just the bits of interest.
The downside to this is slightly more complicated design and routing (to feed all the lines appropriately between the four different blocks), a duplication of some of the machinery perhaps the decoder, probably the sense amplifiers). But those are small costs.
Once you have sub-arrays, of course the point is you route the signals of interest to the appropriate sub-array, and activate then read/write from only that sub-array.

There is a fairly standard layout for doing this that (shown below for the first round of splitting) that allows for reuse of decoder and sense amplifiers across sub-arrays:



So why not keep doing this to get even smaller sub-arrays (and faster, and lower power).?
The standard answer is that keeping all these sub-arrays in sync requires a clock H-tree, and that comes with its own costs in terms of power, design, and latency as you split it finer and finer. And so there's been a (fairly coarse) limit to how small you make your banks, the usual story being that you design down to about 128x(128 or 256) and no smaller.
The sorts of images involved look like (from (2017) https://sci-hub.se/10.1109/SBAC-PAD.2017.14 *Addressing Energy Challenges in Filter Caches*).

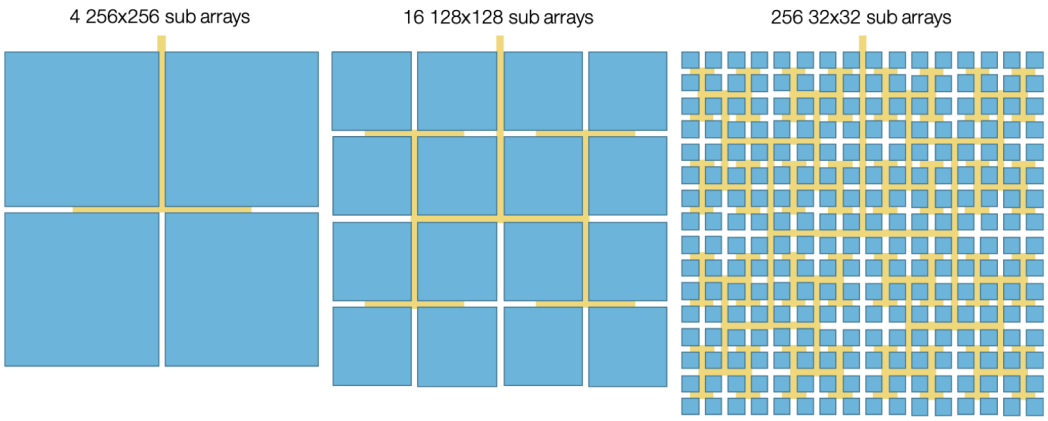


Fig. 3: Diagram comparing the same size cache with different sub-array sizes and corresponding h-tree

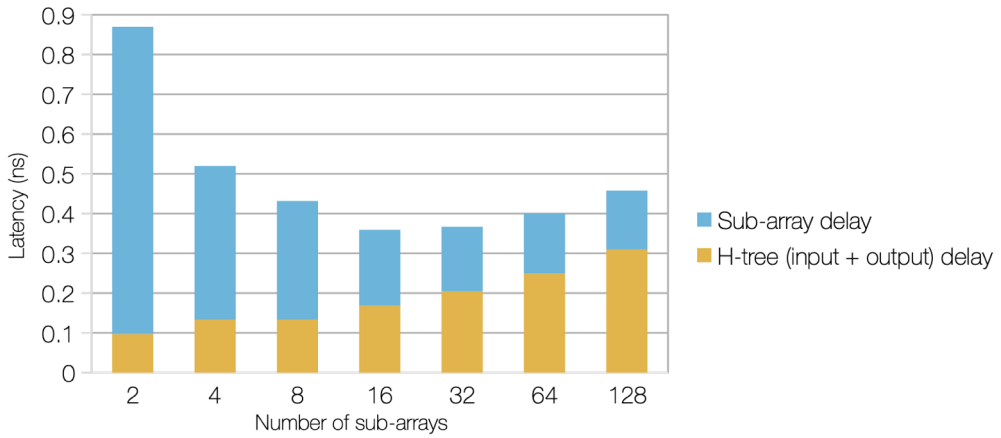


Fig. 4: Comparing the access latency of the same size cache built with different number of sub-arrays

An additional issue is the process details at any particular time. An SRAM consists of a dense array of transistors (that ultimately form the bits) overlaid with a dense array of wiring. In any particular process technology the transistors or the wiring may be the particular limiting factor, one being able to be denser than the other; and if wiring is a gating factor then you will skew your design to one that requires less metal overhead. Right now (7nm and 5nm) it seems like metal is the bigger constraint (especially as we will see that Apple add a number of non-standard additional lines to their SRAM arrays). One of the upcoming hot new technologies to be expected is Buried Power Rail; the idea behind this is to move all the boring grunt-work metal lines that handle power and ground to underneath the transistors rather than with all the metal layers above. This should relieve some wiring congestion and once again change the optimal cache layouts. (Multiple variants of BPR are possible, all

as usual with different tradeoffs; already the path announced by Intel is technically rather different from that expected by TSMC. Beyond BPR, the next step is to also move the clock distribution network below the transistors, providing a second reduction in wiring congestion.)

example - a direct-mapped 4kIB cache

So let's apply all this. Suppose I want to create a 4kiB L1 cache. 4kiB is $2^{12+3} = 2^{15}$ bits. Divide 15 by 2 to get 7.5. That means the obvious array options are 128x256, 256x128, or two 128x128. Suppose we choose a 256x128 bit array. What does this imply?
128 bits in a row means that the maximum size we can read or write in one operation is one row, ie 128 bits ie 16 bytes.
256 rows means into this array of SRAM we will be feeding 8 address lines and 128 data lines.

Alternatively the 128x256 bit option means we will be feeding in 7 address lines, and 256 data lines. With this configuration we could read or write (up to) 32 bytes in one operation.

We could alternatively split into two sub-arrays each of 128x128.
This would allow us to share the decoder between the two sub-arrays, and I'm guessing would probably be the preferred choice all things considered.
The sub-array is the basic unit of "energy activation" so that a request that activates one (half-sized) sub-array rather than the full-sized array will use close to half as much energy. (There are multiple subtleties around static vs dynamic energy and other details, but that's the basic insight, that there's a best-sized sub-array that runs fast enough for our needs; and the smaller we can make that sub-array, the less power we spend in each read or write.)

We have describe all of the above at the *physical* level. What about the logical level? Well, assume our cache *line* length is 64 bytes. None of the configuration above have a *row* length of 64 bytes. Note that a direct-mapped cache of 4kiB with 64 byte lines has 64 lines, and that $2^6 = 64$.
Assume we use the 128x256 option. The row length is 32 bytes. So we would treat two rows as a line. How we do this is not especially important, it's just a matter of bit addressing. So for example one obvious way is to say that the 0th and 1th row form a single line, then the 2th and 3th.

In other words the *logical addressing* is" we think of 6 bits set "which line" and 6 bits set "which bytes in the line", but the mapping logic just outside the SRAM needs to convert that into 7bits(=6 bits from "which line" appended to the high bit from "which bytes in the line") and 5 bits (which will turn into some collection of 32 possible bytes, then into some collection of 256 possible bits along a row that are read or written).

The point is – cache lines don't have to map to row lines, and there's flexibility in how you do the mapping depending on the goal.
Obviously even further outside this SRAM storage array, a tag lookup was performed on the full physi-

cal address to see if that matched the tag of the possible matching line.

example - a two-way set-associative 8kIB cache

Now let's say we want to upgrade from this 4kiB direct-mapped cache to an 8kiB 2-way set-associative. This means we now have
- 64 sets,
- a set is defined by the bits 6..11 of an address,
- each set holds two ways,
- total of 128 ways (ie 128 lines).

Physically the most likely choice is still something like duplicate the 128x128 (x2) that we described above, to give a 2x2 block of 128x128, sharing both the decoders horizontally and sense amps/IO vertically, as in the picture.
Logically we have the same sort of situation as before: we want 64B lines, but our basic unit is 16B rows. So we map 4 rows to a single line.

There are multiple ways we could do this.

1) The simple alternative is to map line 0 to rows 0, 1, 2, 3 of a single sub-array, and you can easily see that as an extension of what we already discussed.
We could then have the second way of the first set be rows 4, 5,6,7. And so on, for the first way of the second set, second way of second set.
All through the first 16 sets, all on one sub-array; then the next 16 sets on the next sub-array, and so on across the four sub-arrays.

2) But an alternative might be to spread line 0 as 16 bytes in the first row of all four sub-arrays.
The advantage of that is that (with a little extra trickery and care in the machinery that's converting logical addresses down to physical row and bit line signals) we might be able to handle reads that cross 16B boundaries by activating two sub-arrays. (Even though the sub-arrays share a decoder, for addresses within a line, they will share the same row in the two sub-arrays...)

3) Or we could put the first way of the first 32 sets in sub-array 0 (using the packing of line 0 to rows 0, 1, 2, 3 of a single sub-array) and the second way of the first 32 sets in sub-array 1, and so on.
*) Or you can think of other variants; it's all just a question of which bits in the address are pulled out and used to activate which sub-array.

EXCEPT how exactly do we plan to use this cache – at the *logical* level?
Do we plan to use it via parallel access (activate both possible ways while we perform tag lookup) ?
Or via way-prediction (activate only one way while we perform tag lookup)?

The third option puts the two ways in different subarrays. This means if we want to run the (2-way set

addressed) cache as a parallel cache, we can activate the relevant rows in both sub-arrays at the same time.
The first option does not allow us to do this, because the two ways occupy the same subarray, so they can't both be activated – we have to use a way predictor.
The second option sounded good, but
- we did not extrapolate it enough to ask where alternate ways of the same set are packed
- it looks like a single line covers all four sub-arrays. Which suggests that we will be forced to run the cache as a serial cache, with no ability to activate both ways of a set (since any given way straddles all four sub-arrays, and only a single row of a sub-array can be activated at one time).

But there's even a fourth option! What if we put the first 64 bits (eight bytes) of the first way into the first half of a row, and the first 64 bits of the second way into the second half of that row. (I told you! Be ready to draw a diagram; if you've got this far without diagrams you will be lost!)
So we are, in some sense *striping* ways into rows. This is just more complication in how we route the bit lines, but consider the consequences:
Suppose that I want to read a particular (aligned) 64bit word.
+ I know from the lowest address bits that it's the first word of the line.
+ I know from some higher address bits which set it is in.
+ What I don't know is whether it's in way 0 or way 1.
But by reading a single row of a single sub-array (and remember, I always have to read a full row anyway) I am actually reading both both possible ways! So in the next cycle, at which point I know (from tag lookup) whether I want way 0 or way 1, I can just decide which half of the output latch I want to read.

I don't need a way predictor, I don't suffer any slowdown, and I'm not paying the energy of two cache line activations! Sounds great, why not do that?
Well it's an option, but like everything it has its costs; for example it means that cache line replacements will become more complicated.
But ultimately it's one of these things to keep in mind as a possibly interesting option that's mostly been overlooked. The earlier *Addressing Energy Challenges in Filter Caches* has a diagram and a few details of what this looks like if you change the numbers slightly, and use this idea for an 8-way associative cache.

One can keep going with this. If we want the standard (for a long time) 32kiB 8-way set-associative cache, presumably we want something like double the 2x2 sub-arrays (forming an 8kIB block) that we have already described. And we have even more variants of what we have already described for how we might arrange a line as rows within a single sub-array, or spread across multiple sub-arrays, and how we might map the different ways of a set onto a single sub-array vs across multiple sub-arrays.

Keep all this in mind because we're not even done yet!

The main thing to appreciate so far is that
- the logical concepts (lines, ways, banks) can be mapped onto the physical concepts (sub-arrays, rows) in many different ways, depending on how you map bits
- different choices give you different payoffs in terms of figuring out which row(s) you need to access for a given way, in terms of how much parallel activity you can have (different sub-arrays can be doing different things, but a sub-array can do only one thing at a time).

practical multi-porting of the cache

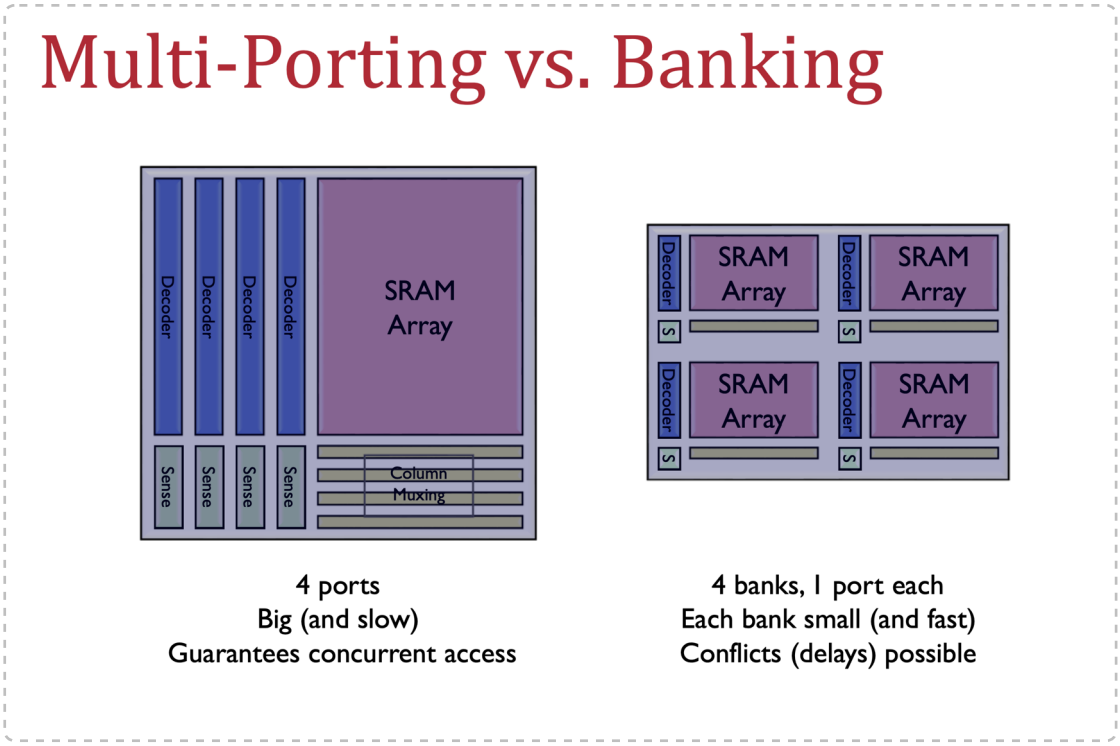
Let's run through some history.
Start in the 1990s. Even with the machines of that time, supporting only a single load-store unit, there are still two clients for the L1D, namely the LSU and the L2 cache. Every cycle that the L1D has to either accept a replacement line from the L2, or has to cast out a modified line from the L1 is a cycle that it cannot handle a load or store from the LSU.
And remember that for "normal" code ~20% of instructions are loads, ~10% stores (these numbers are slightly lower for ARMv8 than for x86 because of more registers, though not as much lower as you might expect), so even with only a 4-wide CPU with a single load/store unit, you'd like to be accessing the L1D$ pretty much every cycle. So to go beyond 4-wide (or even to support 4-wide well) it becomes essential to move beyond a single access per cycle.

But for the SRAMs we have described, in spite of all this sub-array'ing, the overall SRAM array as we have described it still only supports one read or write per cycle (not least because we have only provided one set of address bits and one set of data lines).
It is possible to add extra logic and lines to a single individual SRAM array to allow more than one read or write per cycle. It's ungodly expensive (in terms of area and increasing cycle time) and really the only place you do it (because you have no choice) is for the register file; certainly not for caches.
SO
The consequence of 1RW for an SRAM array is that if we want to load data from, say, three cache lines per cycle, then we need at least three SRAM arrays...
We'll start by exploring this statement but don't forget that I phrased this as "if we want to load data from, say, three lines per cycle"... Is that really what we want?...

Suppose that instead of a single SRAM bank, we provide two SRAM banks each half the size of the initial bank. Note that I'm now using the term bank. Bank occupies an ambiguous place in our terminology; it's halfway a physical concept, and halfway a logical concept. But, conceptually, assume:
- if the cache line length is 128B then address bits 0..6 describe the byte in a line.
- Store all lines with address bit 7=0 in one bank, the "even" bank, and the other lines in the second bank, the "odd" bank.
In other words we're now creating two separate arrays. These arrays are separate in the sense that they each have their own signal lines coming in. Each one has some number of address lines, some number of data lines, some number of control lines.

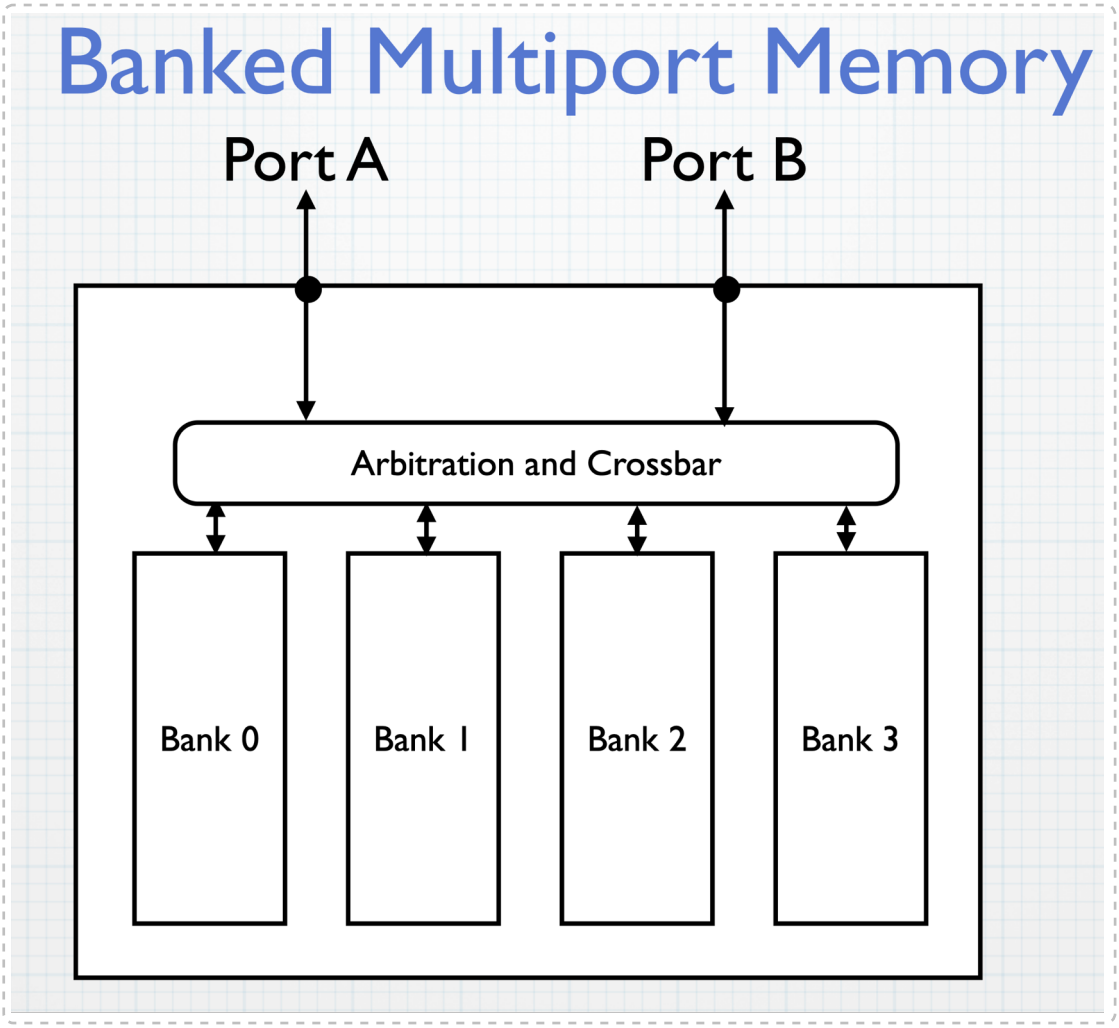
Each array/bank is then split into sub-arrays as we saw above.

(from https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp16/cse502/slides/04-caches.pdf , this is an example of a four way split)



The cost of this is some duplicated logic and duplicated signal lines, but the win is that we can now operate both banks simultaneously, to achieve two accesses per cycle.

If we assume the stream of lineIDs is random, then
- half the cycles will have non-matching in address bit 7 and we get two accesses/cycle;
- half the cycles will have matching in address bit 7 and we have to choose one of them gets access while the other has to wait till next cycle.
So we also need some extra logic like arbitration, to choose which request to service when both requests route to the same bank.
And we need switching logic to route addresses and data between the requesting agents and the banks.



Why is this bank concept not a purely physical concept?
Because the possibility of bank collisions (ie in a particular cycle we can service only one read, not two, if both reads wanted to access the same bank) is programmer visible if you care, either in probing the machine design, or because it's causing a performance problem.

One way to think of it is that an array/bank are both objects that have address lines and data lines; a sub-array is something without independent address/data lines.
So if we split a single array into two sub-arrays, we can still only perform one access/cycle.
But if we upgrade these sub-arrays to full arrays (ie give them independent address, data, and control signals) then we have created two banks, each independently operable (though probably not useful until we also add arbitration and a routing crossbar).

Historically, with transistors and metal levels always a limited resources, one might have prioritized sub-arrays over banks (they are simpler). But in the nanometer world, might it make sense to upgrade all, or almost all, sub-arrays to banks, thereby reaping the advantages of simultaneous access to

separate banks?...

Obviously (really? hmmm? but let's go long with the claim for now) if you have more banks, you'll have a lower chance of any two accesses colliding; and so we arrive at a cache model like, say, the above with
- two possible requests (loads or stores) per cycle
- routed to four different banks rather than just two banks.

What happens next depends on whether you prioritize GHz or performance.
If you prioritize GHz then you want the entire flow to be as lean as possible, no additional delays.
But if you prioritize performance, then even with this simple model you can see a few ways to substantially improve things (at the cost of some extra logic, and a slight reduction in cycle time).
For example instead of simply accepting load and store requests from the LSU in the order they arrive, sort those requests into an even queue and an odd queue, matching the even bank and the odd bank.
Now, instead of just trying to service the first two requests in the unified queue, you can pull the one request from the even queue, one from the odd queue, and substantially increase the odds that you can support two accesses in a cycle, at least under conditions when
- there are so many loads in the code that both queues are frequently occupied by at least one entry, and
- the data access is not some strange weird pattern that only ever requests from the even data bank, never the odd.

how many banks do we want?

What if you simply provide many more banks (say you use address bits 7 8 9 to route lines between 8 different banks), and hope that you will usually not get a bank collision (ie two lineIDs match in the same cycle)?

This sounds good but in fact simply using many banks without additional smarts (starting with the even/odd type queues I described, but just these are not enough) doesn't buy you much in terms of performance.
The reason for this was first articulated in (1997) https://course.ece.cmu.edu/~ece447/s12/lib/exe/fetch.php?media=wiki:juan-ics1997.pdf *Data Caches for Superscalar Processors*. (The performance simulations in this paper are essentially worthless, but the explanations in sections 5 and 6 of common code behavior are important.)
The important fact is that **data access is extremely localized**, mostly consisting of either streaming behavior, or of mixed accesses (some loads, some stores) into a struct. Which means that in any given cycle it's much more likely than not that all two (or three or even four) of the cache accesses that you hoped to service in that cycle all route to the same bank... Oh dear!

Before we consider how to deal with this unexpected glitch, let's note some numbers, just to have a feel for what's reasonable.
It's hard to get accurate data for any modern machine, but Nehalem in ~2010 had

- the TLB implemented as 4 banks

- the tag lookup implemented as 8 banks

- the cache storage is 32kB of 64B lines, which is 512 lines. Given what we have described, you might think that would be split into something like 8 banks, each bank holding 64 lines, and the bank chosen by the three lowest order bits of the lineID.
In fact that's not what is done; rather than banking by line, the banking is by 8B word.
Imagine the memory as 512 lines stacked vertically. If we bank by line, then we split this stack into some substacks by means of horizontal tiles.
But we can also draw vertical lines down the stack, and call each of these columns a bank. So, for every 64B line, bytes 0..7 are physically part of bank 0, bytes 8..15 are physically part of bank 1, and so on on.
This was Intel's (better than nothing) way of dealing with the access locality problem. If you have two loads that reference the same cache line then, when an entire cache line lives in one bank, only one of the loads can be serviced. But if the cache line is spread over eight banks then much of the time the loads will be to different words of the line, and so will hit in different banks and so can be serviced simultaneously.
This is better than nothing, but one can do much better!
(BTW from Haswell forward Intel claim that L1D bank conflicts are no longer present. I strongly doubt this in its most extreme form, ie a claim that the L1D no longer uses banks. I suspect the issue is more that they use a few of the smarts we will discuss below, so as to remove the most obvious cases of extreme slowdown due to multiple requests all wanting to access the same cache bank in the same cycle.)

Having read all this about both reference locality and the desire to service multiple requests per cycle, you might want to think about how all this plays against what we have said about physical SRAM design.
If we want a 32kIB cache, with 8 banks, then we are back to the world of 4kIB arrays.
But now, instead of wanting to use multiple array rows (say 128bits wide) to hold a 64B cache line, instead we want our 4kIB SRAM array to appear to be 64bits wide; so we need to fold the array to make it 128b wide. (Or we could make the lower 64 bits and the upper 64 bits map to two ways of the same set, so that a particular, way-predicted, lookup actually looks up for two ways rather than one, and has a slightly higher chance of success... But as I said, then line replacement is more complex logic.)

As you can see there are many non-obvious choices in design, and the choices made by Intel (and most other companies) have mostly reflected minor tweaks on the historical journey from the earliest caches till today.
You start with a small simple cache (Motorola 68030 had 256B data cache! 80386 came in various versions from no cache [but support for an off-chip cache of a few kB] to an IBM-specific version with 8kiB on-chip cache).
Over the years you increase the size, then the associativity, then you're forced to add banking.
At each stage there's a fairly obvious "good enough" modification to what you already have that is also the simplest choice.

But this also means there's no point at which you look back at all this and ask: Is this actually the optimal solution? What if I started completely from scratch?
Let’s consider ways to modify the SRAM cache ignoring backward compatibility.

some less orthodox techniques used by Apple

smaller subarrays

We've seen that smaller arrays are lower power and (internally) faster, but may result in an (overall) lower clock speed because of H-tree issues. But what if we don't care about absolutely minimizing the cache cycle time because we're designing based on high IPC, not maximum GHz?
This allows us to have *many* small sub-arrays.

energy dissipation (via word and bit lines)

Cache energy is dissipated both in *activating* word lines, and in the *pre-charge* of bit lines. Both operations boil down to raising a wire to a particular voltage level, and dissipating energy by pushing some number of electrons into that wire.
We have control over word line activation in the sense that it
- is part of the addressing of the subarray,
- only occurs one word line per cycle, and
- has to occur at the end of address decoding.
Meanwhile traditionally
- bit lines maintain a permanently pre-charged state,
- recover to their proper voltage levels in a (largish) fraction of a cycle after a read, but in time for a read in the next cycle.
Ideally they would not leak power once pre-charged, but in reality transistors are not ideal and this does happen. (This is one part of the *static power* or *leakage power* you may have heard of, as opposed to *dynamic power* which only happens when bits change their value.)

Suppose that we could pre-charge bit lines fast enough to get this done during some part of the cache usage cycle that's otherwise free. How could we use this?
For example suppose we could decode enough of the address to know which subarray will be read this cycle. We could then pre charge bit lines for only that subarray, leaving the other subarrays un-pre-charged. This would allow us to fire up subarrays on demand, while otherwise leaving them running in a lower power state.
Such a design might encourage us to concentrate as much data as we expect to read into each subarray, so that we can fire up the minimal number of subarrays each cycle and leave the rest untouched.

We can then extend this idea along multiple dimensions.

no pre-charge while sleeping arrays

The easy variant is to switch off pre-charging, at least when the CPU is in the most lightest sleep states. In those lightest sleep states, we maintain power to the SRAM cells to preserve their data; but the clock is stopped, and we know that no reads or writes will occur, so there is no need for the bit-lines to be at their access voltage levels. We can always do this, regardless of the above sub-array trickery.

pre-charge sub-arrays on demand

A more sophisticated version is to observe that sub-array references tend to be clustered, so that a sub-array is used for a few cycles, then not touched. (Obviously this depends on how you map logical data into sub-arrays...)
What one can then do is accept that the first hit of an un-pre-charged sub-array may have a one cycle delay; but that starts a counter which is reset every time the sub-array is hit, and otherwise decremented. When the counter reaches zero, we stop pre-charge.
This mechanism is described in (2003) https://www.microarch.org/micro36/html/pdf/yang-NearOptimalPrecharging.pdf *Near-Optimal Precharging in High-Performance Nanoscale CMOS Caches*. Unfortunately this, and almost all I can find on cache design dates from ~15 years ago, before finFETs changed the details of leakage power and changed the sensible design points.

Even so, Apple does appear to be powering off sub-arrays for at least some purposes.
(2009) https://patents.google.com/patent/US20100329062A1 *Leakage and NBTI Reduction Technique for Memory* discusses a control signal for an SRAM subarray that allows the bit-lines to float rather than staying pre-charged.

(2014) https://patents.google.com/patent/US20150227456A1 *Global write driver for memory array structure*, takes this further, implying the sort of thing described above, whereby subarrays are pre-charged on demand (look at eg Fig 5 and the text leading up to it).

avoid repeated row activation and pre-charging for unvarying row reads

A slightly later companion to the above (2014) https://patents.google.com/patent/US9236100B1 *Dynamic global memory bit line usage as storage node* takes this in a different direction as described below:
The traditional timing is that
- in cycle *n* – 1 we begin bit-line recharge, so that by the middle of cycle *n* we can read the data.
But suppose we can perform the bit-line recharge fast enough to have this start at the beginning of cycle *n*, as suggested.
This opens up a very interesting possibility. When a row (ie a particular word line) is activated, *in the absence of bit-line recharge* the data extracted from the row remains present for a few cycle.
This means that if the next cycle involves reading from the same word line, this read can be performed without paying the energy cost of the bit line recharge or the word line activation. (If you know any-

thing about DRAM, this should sound something like page mode.)

Does Apple still use this idea? It's unclear. The current design seems to use extremely short cache rows, enough so that it's very unlikely that a request in one cycle will be followed by a request to that same row the next cycle. Like the rest of the CPU, the cache designs have evolved.

partial tag comparison informing sense amplifier activation (way filtering, for the L2?)

There is yet a third variant on the above, in the context of way selection!
Remember the way selection issue is that the traditional model simultaneously activates all the possible ways where the line could live while it performs tag read, then, based on the matching tag, only reads out one of those lines. But as we have seen, all the processes involved have multiple steps.

Consider now the following refinement of tag lookup and data lookup
- on the tag side, each tag is some number of bits. Assume for M1 the maximum address space is 40 bits (1TiB), with a 14 bit (16kiB) page size. That means tags are 26 bits in length. Split tag storage into say 22 high bits and 4 low bits. Perform separate tag comparisons on each of these two. Because 4 bits is shorter, we can run that *partial tag comparison* faster.
- on the data side we perform all the data lookup steps till the sense amplifier step. We only activate the sense amplifier if partial tag comparison for this line indicates a match. This works out if the timing is such that the 4-bit comparisons can be completed just before the sense amplifier needs to be activated.

If our design is 8 way associative, then we expect that most lines will not match random 4 address bits, so usually only the correct line gets activated, or sometimes two lines by bad luck.
But most of the time, we can avoid the energy costs of the sense amplifier step which are substantial – at the expense of yet more complexity!

This scheme achieves much of the goal of way prediction, but in a very different way which doesn't require the lookup tables or MRU bits of traditional way prediction. The idea is sometimes called *way filtering* as opposed to way prediction but, like the bank vs sub-array distinction, people are frequently imprecise in their terminology.
I am told that AMD uses this idea for their L1 cache. I can find no evidence as to whether Apple is using this simple partial tag comparison as L1 way prediction, but it might be yet another way that Apple achieves low energy L1 caches without ever, apparently, paying the randomness and occasional mismatch costs of a traditional way predictor.
Certainly (you might not believe it; the patent is not an easy read unless you already understand the idea!) a variant of this idea appears in (2015) https://patents.google.com/patent/US10157137B1 *Cache way prediction*, in the context of an L2 cache.

optimized pre-charge curve

Even all this doesn't not exhaust the complexities of pre-charge!
We have so far talked about optimizing voltage levels for various parts of the circuit and for various tasks, but pre-charging is basically a question of moving charges from one place to another, and if you've studied any physics you'll know that there'll be an optimal $V(t)$ voltage curve that moves a given amount of charge using the minimum amount of energy (and that it will look like an exponential, because these things always do; and it will do the job at minimal energy – but taking an infinite amount of time, because these adiabatic processes always do!)
The takeaway from this digression is that pre-charging at a single voltage, while fastest, is not energy optimal; you can get closer to energy-optimality (and still meet cycle time) if you use two different, appropriately chosen, voltages, and switch from the one to the other at the appropriate time.
And that's the content of (2018) https://patents.google.com/patent/US10720193B2 *Technique to lower switching power of bit-lines by adiabatic charging of SRAM memories*.
Compare this with (2013) https://patents.google.com/patent/US8947963B2 *Variable pre-charge levels for improved cell stability* which also varied the precharge voltage with time, but using a slightly simpler model of a first-level charge, followed (if the access was a read) with a second level charge.

Another way to think of this (or if you prefer, another way to use this machinery once you have it) is that a certain voltage level (and thus certain energy cost) is required to pre-charge within a certain duration, but if the time constraint is reduced (ie we are operating at lower frequency) then a lower voltage level will do the same job at lower energy. Thus we can vary the cross-over time between the two voltage levels depending on the current cycle time.
That's the effective content of (2018) https://patents.google.com/patent/US20190272859A1 *Pulsed sub-vdd precharging* of a bit line.
(The second patents was filed a few months before the first, but the above is my analysis of the optimal way to use the two ideas together.)

how wide should the sub-array row be?

Suppose that the latch that grabs the data from a subarray row and holds onto it to be read the next cycle is low power to maintain.
We've already discussed a version of this idea in the 2014 patent that maintained the word line active while not pre-charging, as long as we kept reading from the same word line.
This might encourage us to store as much of a cache line as possible, in sub-array rows that are as wide as possible, in the hope that the next cycle we can read what we want by reading a different set of bytes from this "pre-latched" line, rather than firing up the sub-array again.

But an alternative way to look at this is that working with a sub-bank wider than byte means that much of what is read from a subarray is wasted. Even if we can limit the waste by reusing part of the row, the least waste comes from the most minimal sized row.
With the corollary that optimally sub-banks should be a byte wide?
Obviously this will cost a little more in surrounding logic and suchlike, but it will save some energy.

And this appears to be what Apple has done, that the fundamental cache access at the lowest physical level is the byte (or perhaps the two-byte word) rather than the much wider 8 bytes or so that appear to be the standard for the last Intel caches I have seen described, as of around Sandy Bridge. Over time we will see various pieces of evidence for this.

different voltages for different tasks

If it interests you, you now have enough knowledge to understand some of the interesting things that can be done at the physical cache design level.
For example, an SRAM array uses voltage for three purposes:
- to perform logic (eg address decoder)
- to maintain the 1 or 0 bit value in a cell
- to read or write (via the wordline and bitlines) the values in a cell.

Do those three voltage values have to be the same? Obviously setting the same is by far the easiest as a design choice.
But suppose we used the principle of task disaggregation to break the job down to three different tasks. We could save power if each were set to the minimum the job required.
This is not a trivial task! Think about it. Not only do you now need multiple voltage planes (and to connect each power tap to the correct plane) but at every point where signals move between the logic part of the CPU and a memory array within the CPU, there has to be a level shifter to match logic level 1 (which runs at a lower voltage) to memory level 1 (at a higher voltage).
And there are connections between logic and SRAMs everywhere (not just cache, but register file, performance counters, branch predictors, ...)!

Even so Apple has done the work:
(2005) https://patents.google.com/patent/US7355905B2 *Integrated circuit with separate supply voltage for memory that is different from logic circuit supply voltage*, which separates out the logic voltage levels from the SRAM cell voltage levels;
Followed by (2009) https://patents.google.com/patent/US20100182850A1 *Dynamic leakage control for memory arrays* where we power the memory cell with a virtual voltage line that drops to a lower value whenever possible, to be pulled higher when necessary to ensure no loss of data,
then by (2012) https://patents.google.com/patent/US8885393B2 *Memory array voltage source controller for retention and write assist*, which discusses providing three separate voltage levels for data retention, when writing, and under “normal” (read, I guess?) conditions.

different transistors for different tasks

The next step down this path is to also use optimal transistors with different voltage and leak characteristics for each of these tasks (data storage, word/bitline control, logic calculations like decoding, ...)
This is discussed in (2009) https://patents.google.com/patent/US20100254206A1 *Cache Optimizations Using Multiple Threshold Voltage Transistors*.

dealing with manufacturing defects

Of course another real-world issue is variability in manufacturing.
One version is this is "weak cells" which generate a smaller bit line signal than usual. This can be solved by using a stronger sense amp when reading those cells: (2012) https://patents.google.com/patent/US8559249B1 *Memory with redundant sense amplifier.*

The more extreme version is that some of the SRAM cells may just not work! The obvious way to deal with this is to add some redundancy, so that faulty rows or columns can be ignored. That sounds good, but how exactly do you implement it?

An earlier solution (which I don't understand, but I assume is a tweak to a traditional solution, thus not explained) is 2010 https://patents.google.com/patent/US8130572B2 *Low power memory array column redundancy mechanism.*

The more modern solution is (2018) https://patents.google.com/patent/US10592367B2 *Redundancy implementation using bytewise shifting*. The more modern solution conceptually is something like
- provide a 65th redundant column for every 64 columns
- detect there's a fault with, say, column 30
- for all control signals that route to bits <30, pass the control signal as is
- for all control signals that route to bits>=30, shift the control signal one bit over.
I *think* this is not quite as extreme as you might first expect, because the data being read or written mostly has to be shifted by some number of bytes anyway, to line up with the SRAM, and so this shifting is made somewhat more complex by having some parts of it possibly shift by nine rather than eight bits.

A significant thing, to my eyes, in this solution is the amount of logic it adds. Traditionally SRAMs have been all about minimizing the number of transistors, but that makes ever less sense going forward. Not only are transistors cheap, but the real constraint on density is becoming wiring.
So if problem can be solved in a way that uses up more transistors but does not impact wiring (at least at the most critical metal levels) why not use it?

It's worth remembering that much of this redundancy technology can also be thought of as power-saving technology. A cell (or column) may not be, exactly, dead; rather it may just not work at a particularly low voltage. But if all the other cells and columns do still work, then redundancy allows us to work around the few finicky cells or columns, while still operating at a lower voltage.

further reading

What you should be seeing is just how large is the gap is between the EE101 basic SRAM and a real SRAM!

Real SRAMs
- incorporate many additional lines (eg to support multiple logic levels, to provide write assist),
- tweak the timing and voltage curves of basic operations like pre-charge,
- tune the difference transistors in different parts of the design (rather than just assuming one transistor type or one N and one P type),
- play tricks with timing,
- include various test circuits (to discover bad cells, and to track how long voltage can be dropped while retaining functionality),
- and have to worry about redundancy!

If you want a summary overview of what we've said about SRAM design, you should now be able to pick out the interesting and important points in this short white paper: (2019) http://www.sure-core.-com/new-wp/wp-content/uploads/2020/03/SureCoreTechnologyPrimerOct19.pdf *SureCore Low-power SRAM Technology Introduction*.
Likewise (2021) https://semiwiki.com/semiconductor-manufacturers/tsmc/296253-register-file-design-at-the-5nm-node/, although it describes register files rather than SRAMs, includes many technical asides about SRAM, the point of which you should now understand!

Cache design

I’ve now repeated a few times the nexus of
- performing multiple TLB or cache requests per cycle
- multi-porting (implemented as a practical matter via multi-banking)
- but naive multi-banking does not work nearly as well as you’d hope because of address locality.

The trick in computer design is, whenever you see a pattern, rather than cursing that the pattern breaks your assumptions, ask how to change your assumptions so as to exploit the pattern.
Recall our goal: we want to perform 4 load/store operations per cycle. But we don't care if that translates into one or two or four cache line accesses per cycle.
We have found that address locality breaks the simplest, most obvious, traditional models for how to design and implement a cache. So we design a non-traditional cache.

The goal is four accesses per cycle, not four line accesses per cycle! Once you understand the difference in these two statements, a whole new design space opens up!

Exploiting this knowledge takes two main forms:

The first is after you have performed a lookup for one address, see if what you have looked up can be used by subsequent addresses.
Lookups are performed by the TLB, and by tag comparison/the way predictor. In both cases, it's highly likely that the pageID or lineID you have just figured out is appropriate to subsequent accesses.

In the simplest form you just look at the next (or next two, or next three) accesses in the queue and see if your result is relevant to them.
Slightly more sophisticated is to perform some light sorting of the accesses in these queues to bunch together accesses to the same page and to the same line. This reuse of lookup data is called a piggy-back port: 1996 https://web.eecs.umich.edu/~taustin/papers/ISCA96-hbat.pdf, *High-Bandwidth Address Translation for Multiple-Issue Processors*. We've already mentioned/seen these ideas in the context of the TLB.

The above is as far as we need to go for TLB lookup, but it doesn't do us much good to have replicated the lineID for two loads that both want to access that same lineID, if you can still only service one load per cycle.
So the second step that needs to be performed is to *coalesce* accesses to a particular line.
For stores this might, for example, take the form of merging together retired stores (ie stores that have been validated as incapable of generating a fault or being mispredicted), as many as possible, into a single buffer up to the width of a line (or perhaps a half or quarter line); then in a single cycle transferring that unit to the cache.
For loads, this might take the form of creating a union of the byte-enables of two or more loads so that the cache returns data relevant to both loads in a single wide payload unit, which is then disambiguated by the LSU into the individual bytes requested by each load.

The sort of model we have in mind is as below. This is from an Apple patent (2009) https://patents.google.com/patent/US8036061B2 that's primarily for an L2 cache, but shows the essential idea – four requestors are coalesced down to two actual requests going over two ports into the data storage.

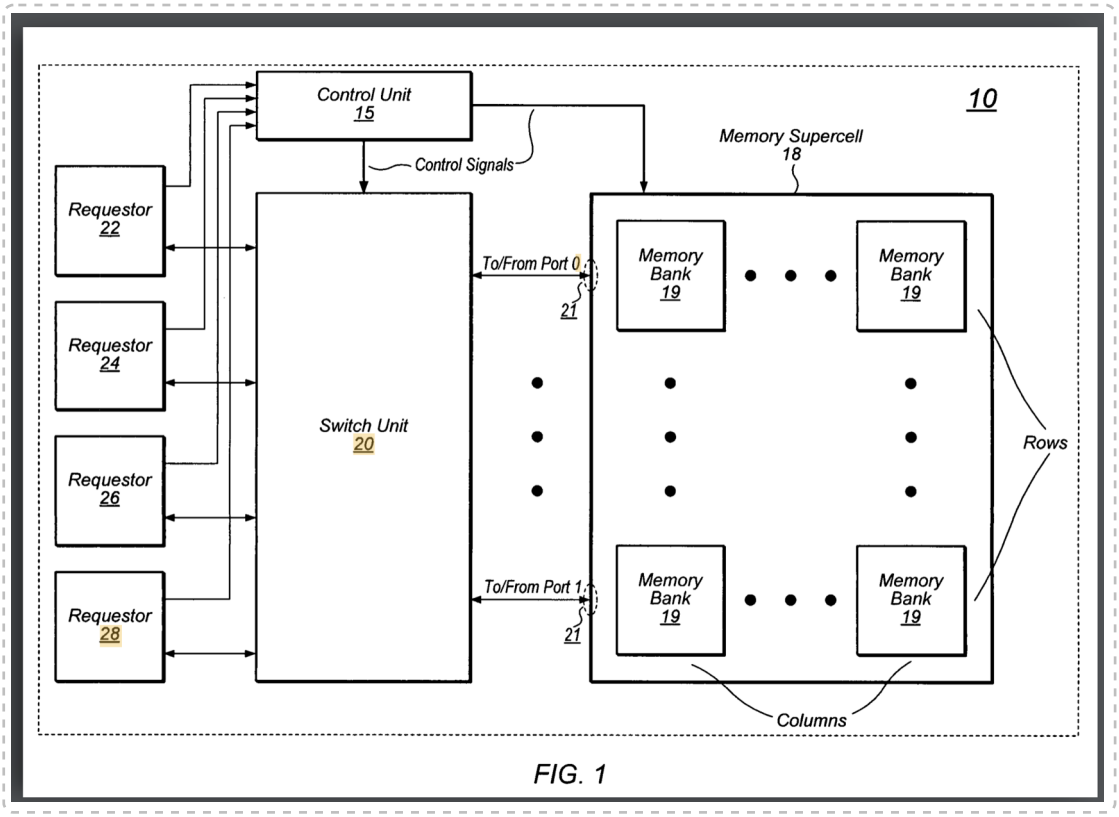


FIG. 1

Let's work though our set of steps that are required to service a load, now listing various ideas that have been suggested to improve each step.
Mostly we will not mention Apple's specific solutions yet, this is more a survey of ideas.

1) construct the address (generally add a base pointer to a [possibly shifted by a small amount] index pointer)
2) compare that address with all the relevant (earlier) addresses in the store queue
3) look up that address in the TLB (based on address bits 14 and higher) to learn the physical pageID
4) use address bits 7..13 to form the setID, look that up in the tag store (which will be holding 8 physical pageIDs per setID)
5) pre-charge the eight lines of the setID
6) if one of 8 tags from step 4 matches the physicalPageID from step 3 that tells us the way (ie the lineID), and select the data from that line of the 8 lines of (5)
otherwise we have a cache miss

address generation (hypotheses)

How can we speed up or improve step 1?
Intel's trick for this stage is to note that the common case when adding an offset to a base pointer is for the offset to be small. So in the same cycle that the address is being generated, the base pointer is looked up in the TLB. If the address (after the addition) is on the same page as the base pointer, which

is frequently is, then you've managed to perform steps (1) and (3) in parallel https://stackoverflow.-com/questions/52351397/is-there-a-penalty-when-baseoffset-is-in-a-different-page-than-the-base .

Another way to exploit this idea (generically called *pretranslation*) is to attach a pageID to a base pointer, and then check if there's a page overflow when performing the address addition; if not you can use the pageID attached to the base pointer. This obviously works best with architectures that have more or less well defined address pointers (think of the old M68K series) but one place where ARMv8 has a well-defined base pointer subjected to small increments is the program counter.
Apple have a patent on this for the PC (2010) https://patents.google.com/patent/US8914580B2 *Reducing cache power consumption for sequential accesses*. (The patent is actually on the idea of reusing the lineID [ie the way] of instruction cache lookup, and so not retesting the cache tags, until the PC crosses a cache line boundary; but obviously if you are doing this you will do the same thing for the TLB!)

Alternatively you can perform the sum of the low bits of address separately from the high bits, ignoring the carry. This is called a pseudo-sum. Later we will explain how it can be exploited.

tlb lookup (hypotheses)

It seems that there is enough locality in the access stream that one TLB lookup per cycle is sufficient, if that TLB lookup is shared with all appropriate entries in the queue of outstanding TLB accesses, not just the next three such accesses.
I've found one paper, https://eprints.soton.ac.uk/347147/1/__userfiles.soton.ac.uk_Users_spd _my-desktop _MALEC.pdf *MALEC: A Multiple Access Low Energy Cache*, which talks about a full design using these sorts of ideas, and it's confident that a single-lookup TLB can service up to a 6-wide LSU without losing much performance. We'll see some evidence that Apple is like this.

tag comparison (hypotheses)

One part of cache lookup is knowing the physicalPageID. But what you really want is the lineID, which tells you where to look in the actual physical storage matrix.
Recall the essential ideas here are:
- the cache is 8-way set associative
- meaning that storage is divided into 128 sets and a block can only be allocated into one of those 128 sets. The set is defined by bits 7..13 of the address.
- so given bits 7..13 I know which of the 128 sets to activate to start reading data (which has implications for power -- done correctly I don't have to touch 127/128ths of the cache) BUT
- but a given set can hold 8 lines. How do I know which line (in other words which way of the 8 possible ways) holds my data?
- the old answer was as I describe in 4, 5 and 6.
- an intermediate answer (used on older low-power cores) was the so-called phased cache, which accessed the tags (and determined the way) then accessed the cache storage in the next cycle. Lower power – but one extra cycle to load the data. However this phasing may still be used in L2 or L3 caches,

where one extra cycle is not that big a deal, while the associativity of the cache can be a lot larger.
- the newer answer is to use a way predictor and start by activating just the predicted line, while simultaneously checking that the way predictor is correct.

Let's consider this last step in more detail.
Most textbooks still write as though a cache is organized as a set of lines with the line flags (modified, valid, ...) and the physicalPageID of the line all arranged along a literal physical line of bits. In this model we fire up all eight lines, simultaneously load the tag (ie the physicalPageID) from each line, compare the tag (using eight comparators) against the desired physicalPageID, and select the bytes from the line that matches. But this is way out of date.
Nowadays tags are physically separate storage that is optimized for their particular job, while the line data is physically different storage optimized for a different job.
The tag storage job is to be given a physicalPageID and to say which way corresponds to that physicalPageID.
This is still done essentially as described above – if there are 128 sets, there will be 128 "tag storage entities", each holding eight physicalPageIDs, one of these will be activated based on bits 7..13, and eight comparisons will be made.
But everything else is different.
First: We're using piggyback ports, so each lineID lookup can be used by any of the successor requests that require access to the same storage.
Second: Everybody these days uses a way predictor. The way predictor will provide a guess as to the appropriate lineID before the tag comparisons are completed. This means that only one of the 8 possible ways in the set has to be fired up. If the way predictor guessed incorrectly, next cycle we will have to fire up the correct line and we have lost one cycle – but usually the way predictor gets it right, and we save 7/8th of the energy.

way prediction (hypotheses)

Various alternatives to way-prediction have been suggested. For example MALEC (a paper I referred to a few paragraphs before) attaches the "way predictor" to the TLB, so that each "entry" in the TLB is not just the mapping from logical to physical address, it also records (for each line of the page) the way in the L1 cache that holds the line. (This is not that outrageous. A 16kB page holds 128 128B lines, so if we associate 3 bits of storage that's 384 bits per TLB entry. For 256 TLB entries [size of M1 L1D TLB] that's ~12kB which is not that big a deal.)
Of course this is no longer a predictor; it is telling us exactly where the line is, rather than us knowing the line can be in one of eight places and looking at tags to figure out which of the eight is the one we want.
But this scheme is problematic for sharing a single TLB port; it becomes messy (though nothing's impossible) to try to extract the way numbers for four different lines (corresponding to four different requests).

You can make this even more interesting by widening the storage to 4 bits and using the extra bits to indicating whether the data is in a particular L1 way, or in L2, L3, or out in DRAM, and so jump directly to the appropriate storage, as described in (2014) https://www.it.uu.se/katalog/andse541/isca14-final.pdf *The Direct-to-Data (D2D) Cache: Navigating the Cache Hierarchy with a Single Lookup*.

These ideas use a fair amount of storage (though not much when you consider what can be done with 5nm technology!) but if that's a concern, an interesting idea between the way predictor and the above ideas is the Way Determination Unit, (2003) https://www.ics.uci.edu/~alexv/Papers/date03.pdf *Reducing Power Consumption for High-Associativity Data Caches in Embedded Processors*, which achieves essentially the same goal as Way Prediction (*usually* establish the correct lineID rapidly, and without paying much energy) using a different idea that's usually lower energy than Way Prediction.

Does Apple use a Way Predictor? I'm not sure they do.
I didn't push this very hard but I did try a test like the following:
- generate a 64bit random number (ie a string of 64 uniformly random bits) and store it in x3
- initialize x6 to 0
- use the probe

```
LDRB w10, [x0, x5]; LDRB w11, [x1, x5]; LDRB w12, [x2, x5];
LDRB w10, [x0];     LDRB w11, [x1];     LDRB w12, [x2];

BFI x6, x3, #9, #1
ROR x3, x3, #1
ADD x5, x5, x6
```

The index increment now takes two cycles (because of the serial dependency of the BFI then the ADD), so we have to double the number of loads in the inner loop.
What we're now doing, even apart from bouncing between loads based off x0+x5 and just x0, is that we extract the lowest bit of x3 , shifted to bit 9 and stored in x6, then rotate x3 by one bit. This means that x6 alternates between 0 and 1024 in a random fashion, and we add this to x5.
This pattern should foil any way predictor, meaning that we should see a noticeable performance difference between the case where x5 is always incremented by 1024 and the case where it is randomly incremented by 1024 (and so presumably frequently has to discard the first optimistic load, based on the way predictor, and try again). But I see no such performance difference...
If you look at Apple patents you will see a few more recent references to ways, and even a 2015 patent for Way Prediction (based on partial tag matching). But looking at the details, these are all about the L2 cache, not the L1D.

bank structure (odd+even hypothesis)

We have some validation that Apple are using odd+even banks at the micro-architecturally visible level in this patent (2011) https://patents.google.com/patent/US9131899B2 *Efficient handling of misaligned loads and stores*.

The patent is old and much of it is, I suspect, completely different in the M1 (for example the tag lookup, how way is determined, even the basics of how parts of lines are handled).
Even so the patent validates that there was a time when Apple based their L1D on even and odd banks, and as far as I can tell that aspect of the design remains (because it remains a good tradeoff of high performance for low power for almost all code).
Misaligned loads and stores has been a constant bugaboo for Apple (and I assume all CPU designers) and we'll see that they keep changing the details of how they handle them.

Regardless of the details, what we have at some point (after TLB and way predictor, or tag, lookups) is either up to four requests, now each with an associated linedID and an offset. These can be associated with the even bank and the odd bank. Hopefully, we have multiple requests (loads or stores) with matching lineID's so they can be coalesced (so as to to perform at least two loads with one cache lookup).
We feed the lineID plus the byte enables to the data array and our bytes flow out. Are we done?

sub-array layout (hypotheses)

Not quite yet. How are our banks physically arranged?
We've already said that we have an odd storage bank and an even storage bank. The storage in each bank could be divided into smaller sub-arrays that have physical relevance (eg lower power, shorter access time) but not programmer-visible relevance. And these sub-arrays could be defined by line or by word or in some other way. We have described how Intel used to bank their cache by words rather than line (but that was a sub-optimal solution to a problem that is better solved by coalescing multiple stores into a single byte-enable mask, so it's not relevant to Apple).

However Apple apparently sub-array their cache by doublebytes. Why?
Suppose I sub-array by lines. Then in a single cycle I can access one (and only one) line, which is fine (locality!); and from that line I may read up to 48 bytes (3*16 bytes) by means of the appropriate byte enables.
Suppose I sub-array by bytes. Then, in a single cycle, I send the lineID of interest to all the byte sub-arrays that are of interest, each byte sub-arrays activates the byte corresponding to that lineID and sends it out.
What's the difference?
The byte-sub-array alternative probably has more complicated data routing and requires the lineID to be replicated to more sub-arrays (or the word line from a single decoder routed to multiple sub-arrays), but each sub-array only has to deliver 1 byte (rather than anywhere between 1 and 48 bytes), and this *reduced current variability* probably makes the physical circuit design easier.
But for both schemes I feed in a single address and out come a number of bytes.

The difference comes when you include stores.
- Stores tend to be as clustered as loads, frequently interleaved with them (think reading a few fields of

a structure while writing a few other fields).
- Stores are not latency sensitive, but there is a bandwidth issue. If we don't service stores reasonably fast, then eventually the store queue will fill up and our machine will stall until some stalls are moved out of the store queue into the cache.
- In the design described (where we are trying to fake the performance of a 3+1 or 2+2 4-wide load/store system while only paying the costs of a 1 or 2-wide system, we would prefer not to spend much cache bandwidth on stores if that can be avoided
- Frequently our stores match to the same lineIDs as loads (as we said the stores cluster with the loads) but stores cannot occur in the same sub-array as a load in the same cycle – you fire up the word line, you fire up the bit lines, and either you send data in or pull data out, but you can't do both in the same cycle

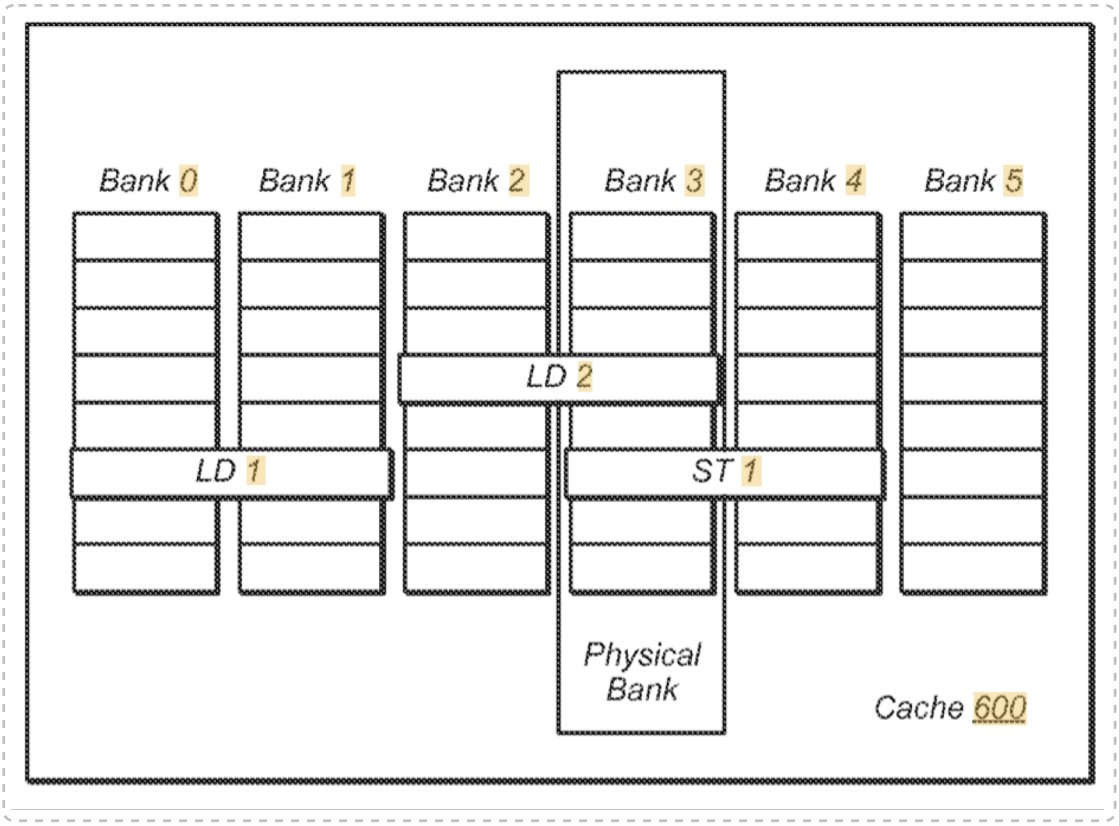
When you are banking by cache line, one way to reduce the cost of stores is, as some of the papers have suggested, to use store merging in the store queue as aggressively as possible, so that when a free cycle opens up on one of the cache banks, a merged store buffer of maybe 40 or so bytes can be dumped in one transaction.
But this still requires a cycle for the transaction, taken away from loads.

Apple utilize a different, slightly more ambitious, alternative:
suppose you are running a byte-banked cache. Then you can run your store cycle somewhat in parallel with your load cycle! You can write (using the same, or a different lineID) to any byte sub-arrays that are not being used for loads in this particular cycle.
Compare the bytes that will be read in this cycle with the bytes that are being written, and usually they will differ (eg I'm reading bytes [0123] of line A and writing bytes [4567] to either line A or a different line B). As long as the bytes differ, I can perform the loads and stores in the same cycle because they live in different sub-arrays.
This gives me the ability to perform most of my stores "for free" in the same cycle that I perform a load, just as long as the store and load do not overlap, but share lineIDs.

Apple take this idea one step further by allowing partial stores. The idea is that even if a load and subsequent store overlap in a few bytes, the non-overlapping bytes will be written out, a bit mask of those bits recorded, and the remaining bytes written out later as opportunities open up. Personally I'm surprised this case occurs enough that it's worth optimizing for, but apparently so. This, of course, allows an even larger fraction of store bandwidth to be free.
The patent is here: (2014) https://patents.google.com/patent/US9448936B2 *Concurrent store and load operations*. The patent also suggests that, opportunistically, lineIDs that are learned by the tag comparison mechanism are associated with the pending stores in the store queue. This means that whenever those stores find an opportunity to access the cache they can do so immediately because everything required for the task is has already been looked up.

My timings suggest that we do have these very narrow cache sub-arrays but that, as of the M1, they are

doublebyte wide rather than one byte wide (so that, eg, loads to bytes [0] and [1] of a line will interfere (cannot execute in the same cycle), but [0] and any byte other than [1] will not interfere.
(This also suggests that while there is all sorts of clever routing of data from sub-arrays sharing a lineID to requestors, this routing happens at the pet-sub-array level; requests within a sub-array are not cracked to allow one byte to flow to one requestor, the second byte to an alternate requestor.
One could imagine that the original design was byte based; at some point simulations and experience suggested double-bye based was a better width, but the next step of “what to do if two request desire the two halves of a doublebyte” was left for a later optimization?)



So the flow for stores appears to be something like
- calculate the address
- opportunistically grab the TLB translation from an appropriate load that's being translated
- opportunistically grab the lineID from the way lookup of an appropriate load
- wait until the store is no longer speculative (although it may be far from the head of the ROB)
- move the store data and address data into a write buffer sitting between the store queue and the cache
(this allows the STQ entry to be freed, even though the data is not yet in cache)
patents suggest there are at least four of these buffers, and that there is probably some sort of store coalescing happening (eg if the buffers are 16B wide, and multiple narrower stores fit in the same buffer, go ahead with that)

- rapidly as practical (without slowing down loads) take advantage of any lineID matches that occur, or a half of the cache that is not being used in a particular cycle, to write out as much of a buffer as possible in any cycle

One final important point. We have written our code as a sequence of loads, and, looking at it on the page it seems obvious that the data should be accessed in certain ways -- if we have three back to back loads from the same line, surely they will execute together! But in the real CPU instructions get reordered in various places, and while the CPU heuristically attempts to execute oldest loads first, that's not a promise. This becomes more of a problem the more the CPU rearranges instructions (in particular loads), it becomes less of a problem the more the LSU and cache rearrange loads to maximize reuse of a particular lineID across multiple loads.

so where are we?

As I have said, I've written and rewritten, experimented and re-experimented, with this section repeatedly. I would come up with a general theory of how it all fitted together, then realize all my probes did not test an alternate explanation, run a new set of tests, and again be surprised.
If someone else has a better theory of how it all fits together, or devises a different set of experiments showing a flaw in my analysis, or even realizes I made a mistake in some of the experiments (it happens! especially when you have no idea what to expect, what counts as an unexpected result), please help advance our common understanding!

Questions:

Obviously there is a lot about Apple's L1D that's unusual. Let's recalibrate.
The **standard** SRAM model includes these aspects:
- a "block" of SRAM (call it a bank) has a single set of address and data lines, and R/W line. It can support one operation (one read or write) per cycle
- it's as close to square as possible (I don't understand why)
- a read/write operation requires an initial step, called activation, which "excites" a row of bits, once this is done signals can be sent down column lines and, based on the bit stored at the intersection of a bit line and column line, a small voltage will be established which can be amplified by a sense amplifier and the bits that were probed can be read.

So imagine a block of SRAM 128x128 (bits) wide.
We will interact with it via
- a 7 bit address bus (which will pass through a decoder to be turned into one of 128 vertical word activation lines)
- a data bus of the maximum width we wish to read or write in one operation (could be as high as 128 bits, but let's say it's 64 bits)
- some byte enables or whatever that tell us which specific bytes (or bits) of the 128 bits in a line are of interest

- some command signal lines
Operation will consist of sending the 7-bit address along with a precharge command, then, some time (maybe a cycle) later the data bits, byte enables, and a read command.

Now the amount of energy used by the precharge is essentially proportional to the length of a row, as is the time taken for the precharge.
So an obvious next step might be to split this 128x128 block into four 64x64 blocks. Since each row length is halved, it should take less energy and less time to activate just the bits of interest. The down-side to this is slightly more complicated design and routing (to feed all the lines appropriately between the four different blocks), a duplication of some of the machinery perhaps the decoder, probably the sense amplifiers). But those are small costs.
So why not keep doing this to get even smaller (and faster, and lower power).
The standard answer is that keeping all these subbanks in sync requires a H clock tree, and that comes with its own costs in terms of power, design, and difficulty in keeping it behaving correctly as you split it finer and finer. And so there's been a (fairly coarse) limit to how small you make your banks.

The above explains the standard model of L1 access.
Supposed you have a standard 8 way set associative VIPT cache. That means that any piece of data
- lives in only one specific set, based on the lower (within-page) bits of the address
- because those lower bits are within-page, they are known independent of TLB lookup
- but that set holds 8 lines (each line in an SRAM row), and the data could be in any of those lines
The original version of the model goes, essential
0: TLB lookup | precharge all 8 rows
1: tag lookup (compare the physical address from TLB with the line 8 tags)
2: depending on which tag matched, read the data from that row
This means 8 energy precharge costs.

So the next model was
0: TLB lookup | read guessed way from the way predictor
1: tag lookup (compare the physical address from TLB with the line 8 tags) | precharge the predicted way (1 row)
2: hopefully, read the data from that row, but if (step 1) indicates a mismatch precharge the *correct* way (1 row)
3*: (maybe) read the data from the correct row

But many aspects of **Apple**'s design suggest that they are operating with a different mindset. I don't know which of the above set of standard assumptions they have modified. It's possible that they are willing to spend 10 or 20% more area or routing wire to design something with very different assumptions.
I cannot really understand their (quite a few) SRAM patents, but they seem to be operating on a mindset that

- isn't quite as concerned with squareness of the SRAM arrays
- uses much smaller SRAM sub-arrays than seems common for other vendors
- these smaller sub-arrays seem to allow them to perform precharge in half a cycle rather than a full cycle, meaning the second half of a cycle can perform read or write, meaning in turn a different sort of timing that allows for sequential tag access then precharge. Maybe this is how they avoid a way predictor (recall that I could not find any timing evidence of a way predictor no matter how randomly I tried to jump around addresses)?

We have as very early patents (2005) https://patents.google.com/patent/US7355905B2 *Integrated circuit with separate supply voltage for memory that is different from logic circuit supply voltage* and (2005) https://patents.google.com/patent/US20070002650A1 *Recovering bit lines in a memory array after stopped clock operation*. Both are low-level circuit patents that don't mean much to me, but seem to indicate a willingness to add additional wires and logic to the standard SRAM array for the sake of better behavior.

Back to the cache data

So with all this in mind, let's return to our cache data.

Out[=]//TableForm=

	[0 0 0]	[0 1 1]	[0 1 2]	[0 3 7]	[0 7 15]	[0 31 63]	[0
128*0=0	3	3	3	3	3	3	3
128*1=128	1.89	1.89	2.29	3.	3.	3.	3.
128*2=256	1.13	1.13	1.53	2.16	2.16	2.16	2.1
128*3=384	1.89	1.89	2.29	3.	3.	3.	3.
128*4=512	1.13	1.13	1.53	2.16	2.16	2.16	2.1
128*5=640	1.89	1.89	2.29	3.	3.	3.	3.
128*6=768	1.13	1.13	1.53	2.16	2.16	2.16	2.1
128*7=896	1.89	1.89	2.29	3.	3.	3.	3.
128*8=1024	1.13	1.13	1.53	2.16	2.16	2.16	2.1

Out[=]//TableForm=

	[0 64 128]	[0 65 130]	[0 129 258]
128*0=0	3	3	3
128*1=128	2.29	3.	
128*2=256	2.16	2.16	
128*3=384	2.23	3.	2.22
128*4=512	2.16	2.16	2.16
128*5=640	2.23	3.	2.16
128*6=768	2.16	2.16	2.16
128*7=896	2.23	3.	2.13
128*8=1024	2.16	2.16	2.16

Let me give my explanation one time, then we'll investigate various items and try to explain them in light of this, best explanation I have come up with.
To simplify, cache lookup requires, for each load/store a TLB lookup, a cache lineID lookup (ie compare

the physical page number with the 8 tags associated with a particular set), and then a byte lookup (to extract the bytes from the data SRAM).
These various stages can interfere with each other; a probe that is designed to test the TLB will inevitably also test the tag lookup and byte lookup; my explanation is my best guess after multiple iterations of probing one aspect, discovering something unexpected, and redesigning some probes to re-investigate some other aspect.

The TLB step we have discussed in detail. Our belief is that
- it is single -ported, but with up to four piggyback ports, so that up to four requests to the same virtual page can be serviced
- with a queue before it that can hold up to about 16 entries (so with fewer than 16 virtual address pages simultaneously active in a loop, we will score multiple lookups each cycle; with more than 16 such pages in a loop, we will never get more than one load/store serviced per cycle)
- with the queue structure actually being four queues, chosen by the two lowest bits of the virtual address, holding up to four elements (by investigating patterns of varying virtual page numbers and seeing how many requests could be serviced per cycle)
- with the non-empty queues probably being chosen by round-robin

The tag comparison/cache lineID lookup surprised me. I expected something like a cross between
- the cache lookup (which seems to have such a striking pattern of an even half and an odd half (by lineID), independently accessed, so ultimately requests that can be serviced boil down to something like
+ multiple requests (up to four) that “appropriately fit into” (to be explained) a single lineID or
+ multiple requests (up to four) that “appropriately fit into” an even lineID and an odd lineID.

and the TLB lookup with its single lookup and piggyback ports.
But that’s not what I found. Consider the address pattern:
(0, 1, 2, 3)*(0*1024+ 128+ 16)
This can be sustained indefinitely (with these four unvarying addresses) at four operations per cycle. This seems to suggest either 4 lineIDs (and cache lookups, but for now we care about lineIDs) per cycle, or some sort of queue that allows batching of loads using the same lineID via piggyback ports.

So let’s do the usual thing. If we assume that tag lookup is split into an even half and an odd half by setID, then a stream of requests that are directed to just one of these two banks should run slower. So
- we switch to just loads (again less symmetric, but writes can confuse the issue in other ways)
- we use the addresses (0, 1, 2)*(2*128+16), and now after each of these three loads, we increment the address by 128.
This still runs at full speed.
The pattern of loads is
024
 135
  246

357

And as you can see from the vertical columns there is both plenty of reuse of the same setID and we switch between evens and odds. So the fact that this runs at full speed (ie 3 loads per cycle) does not convince us of much.

Boost the increment per iteration to 256. Now the pattern of loads looks like

024

246

468

6810

Note that every setID still appears three times in the unrolled pattern of requests. So it could still be the case that a queue before the tag lookup is gathering three (or more) requests matching the same lineID for one piggyback lookup.

This pattern runs at a reduced speed of 400 iterations in 556 cycles, ie (400*3)/556=2.16 loads per cycle. So we have reduced performance. This could be argued as meaning that we are now restricted to one lineID lookup per cycle (only one of the two halves that we suspect tag lookup is split into) but we get reuse because of pre-queues (same argument as TLB lookup).

Let's now increment the address each iteration by 512. Now the repeat pattern will be

024

468

81012

and we see that there is a reuse of each setID only twice, and that only for some of the addresses. 4 and 8 repeat twice, 2 and 6 only once. BUT we get the same performance, about 2.16 loads per cycle. This is not compatible with our theory!

If we now increment the address each iteration by 768, there is zero re-use of a lineID between successive blocks of LDRs. Even so, we still sustain 2.16 loads per cycle. This suggests our hypothesis is wrong. Tag lookup appears to be able to generate

- at least 3 (presumably 4?) different lineID's, something higher than 2 anyway, per cycle.

Maybe none of this surprises you? After some thought, I realized I was trying to be too clever by half. What we see does pretty much match the simplest possible model:

- take the lineID bits (address bits 7.13) defining a setID between 0 and 127
- have 127 separate small CAMs each holding 8 tags
- activate up to four of these 128 CAMs depending on the four setIDs in the four requests
- match the physical page against the 8 CAM entries (the tags)

Apple probably have a small queue (up to four entries?) in front of each of the 128 CAMs so that up to four requests that all have the same setID (and matching physical page) can be serviced by that one (single-ported) CAM in once cycle.

These would be a large number of small CAMs, hence a lot of non-shared overhead (decoders, compara-

tors and suchlike) but the pattern seems to match Apple's general style.

I was thinking in terms of placing the reuse queue at a higher level (eg a single reuse queue/piggyback port construction) either in front of all the CAM, or with the CAMs split two ways, and two queues for each half. But presumably queues and piggyback port logic are cheap enough [relative to the degree of randomness rather than lineID locality] that every CAM gets one?

So at this point we have concluded that
- we can break TLB lookup, but not seriously, and not for normal usage patterns
- tag comparison/lineID lookup is not a constraint, we can do as many of these as we want to the same or different sets. (Presumably we would hit a problem if two requests from two different pages matched the same setID; in that case only one could be serviced. But that case is going to fail in data lookup anyway, it's not specific to the tag comparison stage or to Apple's design; and it's rare; and it's hard enough to probe [careful setup of the L1D required] that I don't care to investigate it experimentally.

- on to data lookup. We have already seen one degree of weirdness there, but let me throw another at you.
Consider eg the lineID lookups we have been testing.
So we create a pattern of lookups that looked something like
(0,1,2)*(2*128+16), call these requests 0, 2 and 4; and we incremented all three addresses each iteration by some number of lineIDs, so that in the extreme no-reuse case the pattern of requests was
024
    6810
        121416
and we saw that, even this this extreme no line-reuse case we could sustain a little over two loads per cycle.
But why do we have the three requests offset within a cache line of 128B, so the first request is always at offset 0, the next at 16, the next at 32 within those 128B?
Let's drop that offset of 16 and use the pattern (0,1,2)*(256) as our offsets.
Yikes! that drops our throughput to a perfect 1 load per cycle!

Do you see how weird this is? We have the same pattern of TLB lookups, the same pattern of cache lineIDs. the only difference is location within the cache lineID. You might assume that (modulo some other issues) each cache line is an independent unit of access; but it looks like the independent unit of access is more fragments of a cache line; something like "I can access separate quarters of lines in parallel, but I can't access the same quarter of two different lines in parallel".

In fact this granularity is, I believe, at the twobyte level. Think of a stack of cache lines. rather than treating a single 128B cache line as the unit of activation, split that stack of cachelines into 64 stacks each twobytes wide. Each of these sub-lines can be independently activated.
This has a few interesting consequences.
For many use cases the system will behave just like a “normal” cache with two banks. Two lineIDs will be activated, one in the even half of the lineIDs, one in the odd half. If the requests are wide (load pair, or a vector load) some large number of the twobyte banks will be activated, but all to the same lineID.

If we have multiple requests to different parts of the same line, we can also service those requests by activating different collections of the twobyte banks corresponding to the different requests (as long as they don’t overlap). This means we can service multiple requests from the same line, but without the need for piggyback ports. We can even perform writes in the same line, in the same cycle, as reads.

Now suppose we once again put a small queue (perhaps just one entry) at the top of each of these stacks? Then, we could have some degree of hysteresis where, if two requests come in during the same cycle with different lineIDs, or the same lineID but they overlap in some way; we could enqueue that load and execute it in the next cycle.
This means if we repeatedly hit the stack of twobytes corresponding to the 0 and 1 address of a given cache line, we will only see a throughput of one load, because one subarray can deliver one lookup per cycle, no more.
But if we spread our lookups across three subarrays (at offsets 0, 16, and 32) then as far as subarrays are concerned, we can perform three lookups per cycle.
Now we don’t always get three lookups per cycle! But we can get a little *over* two lookups per cycle, even when restricted to the even half of the cache, and with a constantly changing cache lineID. It’s easy to imagine ways in which we could be restricted to just two lookups per cycle (for example perhaps there are two decoders, shared by all the 64 subarrays, so in any given cycle, only two lineIDs can ever be activated across all the 64 subarrays). But I have not been able to come up with a convincing theory of how we can get close to two, but slightly higher, throughput in these multi-lineID situations.

My best guess is something like an energy saving story.

- Imagine the half-cache as a stack of lines, divided into maybe eight sections. Think of these as all first ways of each set, then the second ways then the third ways and so on. So a streaming workload would activate each set in turn then loop around to the first set after touching the last. And a localized workload might have maybe three lines active, so its touching three eighths of the sections. Each section is either active or (after a cycle or two of non use) powers down, with a one cycle activation time.
/*
- Imagine that each section has an independent queue of requests, like the TLB queues, not too long. Just like the TLB case, we want to aggregate requests to the same line, and we can do so because of how the lines are split into multiple independent twobyte segments.

- requests are sorted into these queues as they come in. Normally there's a fair degree of line locality, and the queues work well, eg holding overlapping requests in the same line, or two requests to the same subarray but with different lineIDs from once cycle to the next, and usually the next cycle or two brings in more requests to that same lineID, but dealing with different bytes, so the requests in the queue can be aggregated to a single line activation
*/

- two address decoders are available to be shared per section.
- when we use probe code, we are rapidly running through the entire set of cache lines of the even half of the cache. Each cycle, three requests are dumped into the queues (remember 8 queues, one per section) and two are serviced (two decoders for the lineIDs). Usually we only have one section active. But occasionally activity crosses from one section to the next, both sections are active, and so three (actually four) lineID decoders are temporarily available.

This model explains what we see, but I find it unsatisfactory. The low frequency at which we see three-lines-in-one-cycle activity suggests that the packing of "currently active" lines to these eight sections must be very accurate, so that most of the time all activity is happening in only one section, and with static line allocation determing placement in these sections, that seems unlikely. However the test code I am using does have a somewhat streaming structure, so... XXX try code that uses quadratic or boolean logic to generate the addresses...
To get the numbers to really work, we need about 7 requests to be two lines, for each 1 request of three lines. That means a section size of ~16 lines. That means we're working with sub-arrays of 16 lines x 16 bits. Small, yes, lots of overhead. (Though some things like decoders are shared, two for the subarray of blocks associated with 16 lines? Is that possible? Starting to look weird! But logically [as opposed to practically?] it is a great idea!)
Great for saving power. Maybe that is a reasonable assumption, matching our similar hypothesis for lots of small CAMs for tag lookup?

The model contains the essence of what we want
- a constraint that usually only two lineIDs can be active BUT
- occasionally three
and explains this random rareness through energy savings. I'd love to hear better ideas.

I've tried to build models based more directly on the TLB queues (which IMHO explain a similar sort of phenomenon very well). But I can't get them to work in this context given two constraints that appear to be very definite aspects of the cache
- no piggyback ports
- only lineID access to a subarray per cycle.
A constant stream of requests to offset 16 of a variety of different lineIDs, regardless of how it is queued, cannot generate more than one request into the "offset 16" subarray per cycle... This is unlike

the piggyback ports situation, for the TLB, and for tag comparison.

My best guess is a version of the TLB story, again based on queues. So forget queues per subarray: imagine something like this
- We have 1024 cache lines (128kB/128 line length), 512 in each half.
- We (just like the TLB) want to aggregate requests to the same line, and we can do so because of how the lines are split into multiple independent twobyte segments
- so we have some number of queues that hold incoming request, maybe one per line (so 512 queues on one half, each holding maybe one additional request)
- requests are sorted into these queues as they come in. Normally there's a fair degree of line locality, and the queues work well, eg holding overlapping requests in the same line, or two requests to the same subarray but with different lineIDs from once cycle to the next, and usually the next cycle or two brings in more requests to that same lineID, but dealing with different bytes, so the requests in the queue can be aggregated to a single line activation
- two address decoders are available to be shared across the half cache.
- when we use probe code, we are rapidly running through the entire set of cache lines of the even half of the cache. Each cycle, three requests are dumped into the queues (remember 512 queues, one per line) and two are serviced (two decoders for the lineIDs).
- so requests build up across all the queues
- of course this can't go on for ever and at some point resources run low, and we reach an equilibrium of two requests coming in, two going out
- but because of the queueing (and because servicing is, who knows, again maybe random among non-empty queues, or round-robin?) we will occasionally hit situations where our probe addresses have wrapped around, and one of them matches an earlier probe address sitting in a queue and not yet serviced, under which

- The odd/even number of lines pattern is easily explained. When we increment x5 by one line, then we'll generate a nice stream of loads that alternate between the even and odd banks, giving us the ability to load from two line ID's per cycle. When we increment x5 by two lines, we can only generate one lineID per cycle.

- consider something like [0 7 15] with an increment of 128. This alternates between lines (so in any cycle it should be able to generate two line IDs) AND we only need one line IF we're allowed to coalesce three loads into a single line access. We see good (but not perfect) performance -- recall the noisy graph we displayed.
To me this suggests

(a) Only two loads can be coalesced into a single cache access, not three. So we do actually need two lineIDs per cycle.

(b) What's causing the occasional glitches that we see in the noisiness of the curve I displayed? My first thought was Way Prediction failures, but that no longer seems plausible to me. Tests probing Way Prediction show that it never seems to fail, not even for a sequence of random loads, so I don't think there is Way Prediction. Also the noisiness we see is not "really" random in the sense that it's easy enough to reproduce (or not reproduce) depending on exact parameters chosen. There's something I'm missing that acts as a source of "randomness" depending on the exact length of an inner loop, as you see in the graph, but I've no idea what it might be.

OK that sounds plausible, but now compare that to [0 7 15] with an increment of 256. We expect this to be worse, but we specifically expect it to be fewer than 2 loads per cycle. If we can only generate one lineID per cycle, and if we can only coalesce two loads onto a line, then how can we do better than two loads per cycle? This is a complete mystery to me!
We can clarify this point by considering the column [0 129 258]. Now, clearly, each cycle we are loading from three different lines, and (once we start incrementing by at least three lines each iteration) there is zero possibility of within-line reuse.
Yet in all these situations, across the board we seem to be able to achieve about 8% better than the raw 2 loads/cycles expected, which presumably must mean we can do (slightly) better than two lineIDs generated per cycle.

There is a second mystery, which is the columns [000] and [011]. Why should these be substantially lower?
In this case I think I have an answer. We have suggested that Apple is making aggressive use of piggyback ports to reuse a single TLB lookup across multiple accesses, and to reuse a lineID calculation across multiple accesses. Are they doing the same for data lookups?
In other words, suppose I want to load bytes 0 and 7 from a single line. We agree that a single TLB lookup and lineID generation occur, and that the two loads are treated as a single set of byte enables (for bytes 0 and 7) generating two byte values which are then routed to the two LDRB's. But suppose I have two load requests that are *both* byte 0 from the same line. This generates a single byte enable, and a single byte is returned from the cache. In principle, sure, one could imagine the same idea of piggybacking to route that same byte to both loads. But piggybacking is not completely free; adding the logic to route the same value to more than one place takes a little area and adds a little cycle time. In the case of TLB and lineID lookup, it makes sense because a common TLB and lineID across successive accesses is a very common pattern. But in the case of data bytes it makes no sense. No real code repeatedly loads the same bytes from the same place in the same cycle! The best you might get is something like one load of bytes [0123] and a second overlapping load of bytes [2345]; and even that (rare!) pattern is probably better handled by loading non-overlapping words and using the EXTR instruction on the two registers.
So I think it's reasonable to assume that in the cases of [000] and [011] the duplicated byte loads have to occur in distinct cycles. In the case of [000] all routing to the same bank, the best we can

get is one load per cycle (one bank access, no overlap); in the case of [000] routing to both banks we can get two loads per cycle on half the cycles, so we'd expect an average throughput of 1.5. This (mostly) explains the pattern of the [000] column.
What about the [011] column? If 0 and 1 can be loaded from separate banks then we should have some of the loads able to share the 0 and the 1 access, and should have better performance, but in fact we see identical performance. My best guess is that, in spite of what's said by the the Concurrent Store and Load Operations patent, the banks (or perhaps the data routing network above the banks that splits bytes across separate loads) operates at halfword (16bit) rather than byte granularity. Obviously you pay some overhead for smaller granularity, and perhaps Apple's investigation of real code showed that being able to coalesce loads down to byte granularity (and perform partial stores at that same granularity) just wasn't common enough to be worth it. (Of course successive byte access is probably common. But consider a stream of successive byte accesses to bytes 01234567; if you coalesce [024] and [135] then you still get three loads/cycle throughput, at the cost of slightly modifying the timing (you get bytes 024 in the first cycle rather than bytes 012), which is probably fine in the overall OoO scheme of things.)

So that's my explanation for (most of) the first two columns; they show that piggybacking is not used at the data level only the TLB and lineID level, and that data coalescing is at the halfword rather than the byte level.
If we accept these (tentative!) epicycles to the underlying model then essentially what we're saying is that across the entire table
- where we see ~2.16 we should expect 2
- where we see ~1.13 we should expect 1
- where we see ~1.53 or even ~1.89 we should expect 1.5 (alternating two loads in one cycle, one in the next)

This still leaves a lot unexplained!
Most of the numbers are a few percent higher than expected. It's easy to ignore numbers that are lower than expected -- random way predictor failure or whatever. It's harder to explain why things work better than expected! If we assume that the system is more generously provisioned than in my model (eg three tag banks allowing for the generation of three lineIDs per cycle) then it's hard to see why the improvement is just a few percent, or why the pattern is so strikingly odd vs even.
It's like there's a parallel secondary way for loads to acquire data that bypasses the cache completely and so is not subject to the limitations of only two lineIDs.
In fact there are such additional parallel channels. For example
- loads check stores in the store queue before trying the cache. In principle some stores that just happened to point into the buffer from which we're loading could still be sitting in the store queue. I added a quick loop that writes hundreds of thousands of bytes to a different region of

memory to try to flush out all such stores; no change.
- Apple has a few patents for implementing loads from the registers that fed earlier stores. For example https://patents.google.com/patent/US10838729B1 detects if the pattern of register usage (base register is the stack pointer, same offset) matches an earlier store to the same location. If so, and if the physical register stored at that location has not been overwritten, then the load can be transformed into a rename to that physical register. This is a neat idea (somewhat like Intel's Stack Engine) but doesn't seem relevant to our benchmark.
- Apple also has a patent on a load-related value predictor, https://patents.google.com/patent/US20210049015A1 . It's *possible* that some subset of our loads match their patterns enough that the value being loaded is stored in the value prediction table, and is thus accessed via a parallel channel. This hypothesis nicely matches the fairly small improvement over baseline that we see and its somewhat random nature. The problem I have is that even if you use a load predictor in this way, at some point, before the load retires, you have to validate that the prediction was correct, actually going to the cache and looking at the ground truth data -- which runs into the same problems as before, that this lookup requires a lineID.

Finally there is the case of zero increment and [000] or [011]. If everything I have said is correct, about no data piggybacking then this should perform like 128*2, requiring a separate lineID calculation (in a separate cycle) for every one of the three loads. Clearly we don't see that.

So this is where I'm at right now :-(We have a model that explains much of how Apple achieves their wide load store performance at low power, backed up by some literature and some patents, and matching much of the data. But it can't be denied that the exact match numbers are often off by a few percent (in a direction that's hard to explain) and that a few particular cases like [000] and [011] at x5=128*0 simply don't match our model at all.
Ideas?

Wider loads (and non-aligned loads)

What more can we investigate?
One option is wider loads. Let's move to the next simplest level, naming loading halfwords rather than bytes. What we would expect is that
- aligned halfwords behave essentially like bytes (and this is what we see)
- non-aligned halfwords pay essentially no penalty, given the mechanisms we have described (and again what we see)
- but what about halfwords that straddle a cache line? These cannot be handled for free via the byte enable mechanism, so we expect some cost, but what of the details?
The easy way to handle misaligned loads of this sort is to split them into two and issue two

separate partial loads. While this is not optimal, it will "automatically" handle all the complicated possible cases (consider eg half the load can be found in one element of the store queue, while the other half takes a TLB miss, then a cache miss...)
But of course Apple go in for optimal rather than for easy, so that, as described by this patent https://patents.google.com/patent/US20130013862A1 , a misaligned load is still handled as a single unit with two line requests sent to even and odd banks.
We can predict a few consequences of this. Consider eg the case [0 63 127] where the 127 halfword spills over to the next line.
First we'd expect that in the cases where we are stepping by 256 (so all loads are nominally limited to one bank) the misalignment will cost us nothing (because it is a transaction against the other bank, which we are otherwise not touching), and this is what we see.
Second we'd expect that when stepping by 128 we'd see some interference, a small slowdown, and again that's what we see. (Rather than a noisy rate between about 2.5 and 3 loads/cycle, we see a non-noisy rate of 2.3 loads/cycle.)
Finally consider a more extreme non-alignment case where both the first load is non-aligned at -1, and the third load at 127, so we have something like [-1, 2, 127] . This runs at 1.9 loads/cycle.
In principle this would require loads from lines
-1 [byte 127]
0 [bytes 0, 23 and 127]
+1 [byte 0]
So a sub-optimal implementation might max out at 1.5 loads per cycle (requiring 2 cycles for each round of 3 loads). Since we're seeing something closer to 2 loads/cycle the machinery must be smart enough to merge successive partials load (as byte enables) across successive load cycles. (But this would appear to piggybacking byte loads... I still do not understand why the machinery appears to piggyback all data loads except the most obvious case of identical byte offsets.)

There are a few other interesting aspects of non-aligned loads or stores.
Suppose a load is aligned so that part (but not all) of it is covered by part of a previous store. This can become truly horrifying! Imagine a 16 byte load, for which all the even bytes are distinct byte stores that are sitting in the store queue! The easiest way to handle this is when the load tries to execute and realizes there is a problem, the load is scheduled for Replay. In other words, essentially the load will retry (probably multiple times, wasting load bandwidth each time) until all the stores are finally propagated to the cache and so the load hits in only one place.

There are a few different aspects to this:
(a) how do we detect this situation?
(b) how to we extract the load data under these conditions?
(c) how do we handle a non-aligned load that hits in the cache but crosses two cache lines?

For detection we have (2007) https://patents.google.com/patent/US7996646B2 *Efficient encoding for detecting load dependency on store with misalignment*. The easiest solution to the problem is to allocate entries in the LSQ by their cache line, along with a bit map indicating which bytes in the cache line are of interest (read or written). We can then find matching cache lines, and for those matching lines simply AND their bitmaps.
But that requires a lot of extra storage, 64 bits for a cache line of 64 bytes, per LSQ entry.
(What to do for a load or store that crosses a cache line? Easiest is to allocate two entries in the LSQ, one for each part of the load or store.)
The patent describes an alternative way of specifying which bits in the cache line are read or written, using only 20 bits, which can still be reasonably rapidly compared against each other. It's not perfect (like a Bloom filter, it will cache every problematic case, but will also generate a few false positives, which can be more accurately tested before panicking and Flush'ing) but it's good enough; and while the details are for an older class machine (64B cache lines, and essentially ignoring 128B length items as not relevant to the problem) it can be updated.

For performing a load that partially overlaps with one (or more) stores in the store queue, Apple's original solution was to have a predictor for this case, (2005) https://patents.google.com/patent/US7376817B2 *Partial load/store forward prediction*. If the case is predicted, then the load is split into smaller sub-loads which (hopefully) hit in only one location and are then pieced back together. It's an interesting patent to read in terms of how the solution was achieved – both the idea of how to reuse parts of the existing design, and the way the predictor elements are created (by successively splitting troublesome loads in two and seeing if those pass through without difficulty).
Even today, maybe the same solution is used? It's a nasty problem with no obvious (to me!) better answer. Sure, it's not especially performant, but if you are writing code like this, what do you expect?

How about handling the cache side of non-aligned loads or stores?
The original solution was (2005) https://patents.google.com/patent/US8117404B2 *Misalignment predictor*, which uses a predictor to detect loads or stores that cross cache boundaries, and crack them into sub operations. In other words we reuse machinery that already exists in the CPU to treat such a load or store as two loads or stores, one for each cache line that is covered. (Presumably the first time we try this, before the predictor knows about this load/store, it gets flagged by the LSU, the predictor is trained, we Flush, and restart at this problematic instruction.)

This mechanism is followed up (2011) https://patents.google.com/patent/US9131899B2 *Efficient handling of misaligned loads and stores*, which we have already discussed regarding the cache having even vs odd sides. This patent describes how the LSU includes "sidecar" storage which is

some temporary storage in the LSU that can be used to glue together these fragmented loads, rather than using standard physical registers and the ALU as in the 2005 patent.
We can see here that we're now dedicating some storage and logic in the LSU to solving this problem, rather than reusing integer ALU storage and machinery.
(Presumably the predictor for the misaligned loads/stores is now no longer necessary.)
It's worth being reminded that all these patents are pre-A6 (2012), and A6 had a single load-store unit. But there appear to be no subsequent patents in this space, and it seems likely that the ideas of the 2011 patent would generalize well to the L1 cache model described here.

Another set of uncommon cases that, nevertheless, have to be handled correctly, is uncacheable loads (and stores), usually for the purpose of controlling some IO. There's a patent (2006) https://-patents.google.com/patent/US20080086594A1 which is, interestingly, abandoned as of 2014. Was it considered too obvious to be patented? Or is there some technical reason why it was viable for PPC (PA Semi's primary target before the acquisition) but not ARM ?
The idea is that uncacheable loads are placed in one (of multiple) MRBs (memory request buffers) attached to the L1D and, if subsequent "nearby" uncacheable loads occur (think load in the same "cache line") they are merged into the MRB. In a sense this is an obvious optimization (though one has to be careful if some of these uncacheable loads are involved in device initialization or whatever and need exact sequencing...), but on the other hand how valuable is it? It's an uncommon case (uncacheable load) with an uncommon subcase (two of these addressing nearby storage, nearby in time) that can be handled in software (just use a wider load and split the pieces apart in the code!)
My guess is that this (and various similar cases that also involve the MRB, like having a store that merges in an MRB that's about to be written out) are all individually uncommon, but they can all be handled with much the same machinery, and if you add up all these three or four different cases, they make the effort worthwhile?
Interesting (simply in the case of wondering what's going on!) is the followup patent (2012) https://patents.google.com/patent/US9043554B2 . This covers the same store territory, with the same solution of coalescing uncacheable stores as much as possible, and looks no different to this outsider apart from the terminology being used. (Perhaps the different terminology hints at important differences at the implementation level?) This patent also includes a strange description of the handling of "uncacheable" loads in the L2 which makes no sense to me; presumably it's using some technicality whereby those loads can be cached in L2 given the precise details of exactly how Apple's SoC handles coherency and non-coherent IO blocks. I suspect this load material has become irrelevant as Apple has had time to improve their coherency infrastructure and to make every external agent and block of memory coherent.

Superficially in the same space is (2006) https://patents.google.com/patent/US7624235B2. This

appears to have the SLC as its primary concern, but the ideas are interesting and could in principle be used in the L1 or L2.
The main idea is that rather than providing the cache with some number of special purpose buffers (in particular buffers handling IO transactions with unusual characteristics), the general lines of the buffer can be used for this purpose. This requires a way to indicate that those lines are being used in a special way, which is easy enough. But the idea raises a concern -- given the set-associative nature of the cache, what if a particular set is heavily used by IO, crowding out all the ways that should ideally be used as cache?
This is explained by a second point which is (IMHO) far more interesting than the patented point! What is actually set-associative in this cache is only the tag storage; tag storage provides a lineID describing the placement of the line, but that connection between lineID and placement can be arbitrary (which in turn requires a list of free lineID's, again no big deal). The patent describes how using a line for staging buffer (ie IO) purposes does not require a tag, and so using many lines in this way will reduce the cache capacity a little, but will not hurt associativity. But of course, once you have severed the link between line placement and associativity all manner of possibilities open up! These include easily allowing parts of the cache to become drowsy, or totally powered down; or allocated for different purposes (eg per CPU QoS) while still retaining placement flexibility.
(This cache used as staging buffer aspect of this patent is easier understood by comparing it with (2005) https://patents.google.com/patent/US7412555B2, which describes an earlier version of the same essential design, and mostly using the exact same words+diagrams, where a distinction is drawn between an IOC [IO cache] and IOM [IO memory, used as staging buffer]. A year later the design has evolved to consolidate the IOM and IOC as a single storage pool.
The 2005 patent itself is interesting in that I have never seen anything like it before. The problem to be solved is that buses, like PCIe, have various types of transactions and various rules for when these transactions may or may not be handled out of order. One wants to follow all the rules, necessary for correctness, while also boosting performance by taking maximum advantage of whatever ordering flexibility the rules allow. The patent describes a way of doing that, at least to some extent, with minimal additional logic.)
Certainly I think the assumption that everyone has made that the M1's L1D must behave like an 8-way set associative cache (because 128kB/16kB page size, QED) is something that needs to be tested, not just assumed.

Even as recently as 2019 uncacheable memory operations are still on Apple's mind, though I have to admit I have no idea what parts of the SoC are still operating as non-coherent memory, and the patent gives no suggestions. The actual ideas in the patent, 2019 https://patents.google.com/patent/US20210056024A1 *Managing serial miss requests for load operations in a non-coherent memory system*, are fairly simple, essentially a load equivalent of store merging:

• there is a pool of buffers in each cache controller for the use of uncachable loads
• when a load is placed in one of these buffers a timer starts
• if a subsequent load comes in that can be aggregated with this load (ie the two together hit in the same cache line and so for a single wider load) that is done
• until either the timer expires, or a maximum width load is constructed
I assume the idea is that many use cases for these uncacheable loads consist of a sequence of back-to-back sequential loads of some limited width (maybe 32b or 64b depending on the block of the SoC) and this is a fairly easy way to consolidate them to full cache line width transactions.
You'll also note the long gap between the early round of these patents, from the PA Semi days, and this patent. Perhaps, in a sense, it's a consolidation of the ideas from these early days, optimized to take full advantage of the precise memory-ordering rules of the IP Apple currently care about (latest versions of ARM, AMBA, PCIe, etc) without having to worry about some of the issues that seem to have limited all the earlier patents to only store aggregation, never load aggregation?

Bandwidth

Now what about bandwidth? Just because we can run 3 loads per cycle, the cache might not be able to sustain that bandwidth if the loads are all maximally wide!
Let's update our standard load probe to load quad words (ie NEON 128 bit registers), so (LD q0, [x0, x5]; LD q1, [x1, x5]; LD q2, [x2, x5]; ADD x5, x5, #)

And we'll use the line layout [0 31 63] (yes, these are not aligned, but as we've established our cache, based on line byte enables, handles misalignment with aplomb as long as we don't cross lines).

The easy case, where x0 is never incremented, runs at three loads (48B) per cycle. Not too surprising. (Except, of course, it IS surprising -- matching what we have already seen for repeated loads from the same cache line, but no easier to understand!)
More demanding is where x5 is incremented by one each cycle, which also runs at 48B per cycle, though with some noise.

Incrementing x5 by 256 runs at the now familiar 2.16 loads/cycle. To me this validates our earlier hypothesis that only two (non-overlapping) loads can be coalesced to make use of a single lineID. This in turn implies that the data buses bus to and from each of the even and odd banks is 16*2B wide, not 16*3B wide (since how could a bus wider than 32B ever be utilized?)

Stores

What now of stores?
Let's add a single store, at offset x3, so we're dealing with byte offsets [0123]. Incrementing x5 by zero, we get 3+1 load/stores per cycle, as we were promised.
Incrementing by 128 we get the same sort of timing (~523 cycles for 400 iterations) as without the store.
Likewise incrementing by 256 gives us the same sort of timing as for just loads.
Clearly the data bus between the LSU and the cache must be split with separate read and write buses that can be used (for the same lineID) in the same cycle.
If we move to [012 131] (so that the store is now being performed the next line down) once again performance is as you'd predict.
Finally switching to two loads and two stores behaves again as you'd expect -- limited by two lineID's per cycle, but able to perform coalesced loads and stores to the same line in the same cycle.

Putting this in the context of bandwidth , as usual, the common cases where you'd want maximum bandwidth (either blits or walking through an array) will usually involve alternating between the even and odd banks, in which case you will hit the maximum 48B/cycle (or even 48B +16B store, depending on the address pattern).
But, again as usual, this will drop if your address pattern is limited to one bank, and drop again if your separate loads and stores do not share a lineID.

Latency issues

Bandwidth and multiple loads per cycle are, of course, important, but an additional important dimension to loads is latency.
In an ideal world, we'd be able to use the result of a load the cycle after the load executes, just as we can for many integer instructions. In reality this is way too optimistic, but how bad is it?

Let's consider a class pointer chasing loop, with the baseline probe LDR x2, [x2]; and with x2 initialized to point to itself. This takes 3 cycles per iteration. Pretty impressive!
Let's try a few variants.
First make the addressing more complicated. Initialize x5 to 0, and use LDR x2, [x2, x5]. Still 3 cycles per iteration.
How about LDR x2, [x2, w5, sxtw #3] (sign extend w5, ie 32 bits not 64 bits, and shift the result). Still 3 cycles per iteration, so a maximally complex addressing mode does not hurt.
How about LDP x2, x3 [x2]? This is an interesting case! The pair aspect of the load pair, we know is non-problematic and easily handled by the LSU and the cache. More interesting is whether this

structure (unusual, but justified as a CAR+CDR sort of thing) still gets the fast pointer chasing path. Yes it does! Yay Apple!
BUT this is a fast path! Suppose we tweak it just slightly, to LDP x1, x2 [x2] (this obviously means we also need to store the address of x2 in x2[1]). This case takes 4 cycles per iteration.

More generally suppose we want to use the result of the load. The classic case is something like (LDR x2, [x2], EOR x2, x0, x2, EOR x2, x0, x2).
We might expect this to take 3+2 cycles per iteration, but in fact it takes 4+2=6 cycles. Why is it slower? Because it's using "standard path" load, not "fast path" load.
The way Apple have implemented their fast path is that if the destination is
- an int register
- the first of a load pair
- sitting in the load queue as an address input
then the load can be completed in three cycles, otherwise four.
Of course there's a patent, from 2014! But before we get there, look at the much earlier patent (2007) https://patents.google.com/patent/US8364907B2 which shows you how things were done in the simpler world of seven years earlier. The essence of the patent lies in two points
- the cache (as of 2007) is written to as lines (not as the bytes we see later). The cache is also ECC protected on 4-byte boundaries. Meaning that an unaligned store that crosses these 4-byte boundaries (or is shorter than 4 bytes) needs to engage in cycle of (load the surrounding data, merge in the new data, write the whole merged lot back to the cache). The patent talks about doing that while using one of the memory request buffers also used to hold line cast-outs or various other interactions with the memory system.
- the second aspect of the cache is the (apparently trivial and uncommon point, but necessary for correctness) of performing the above operation correctly when the store of interest hits in a line that has already been chosen for cast-out, and so the entire line is already in the writeback buffer.
Reading the patent is (IMHO) fascinating in terms of showing how much detail has to be considered (even for a basic cache store mechanism), while also showing how many points of friction there are in such a basic mechanism, all these points that have been removed over the years (for example the stuff I've described about performing stores in parallel with loads, as bytes can be handled, or a much slicker pipeline with more half-cycle and overlapping stages, and less repeated work).
It's an interesting question what's happened to ECC for the byte-based cache. My guess (only a guess) is that the cache is assumed to be robust enough (some combination of design, experience, and modeling) that a per-byte parity bit or weak 2-bit checksum is good enough?

For the specific (2014) patent: https://patents.google.com/patent/US9710268B2 , which includes some interesting information in the context of everything we've been discussing, specifically this

diagram:

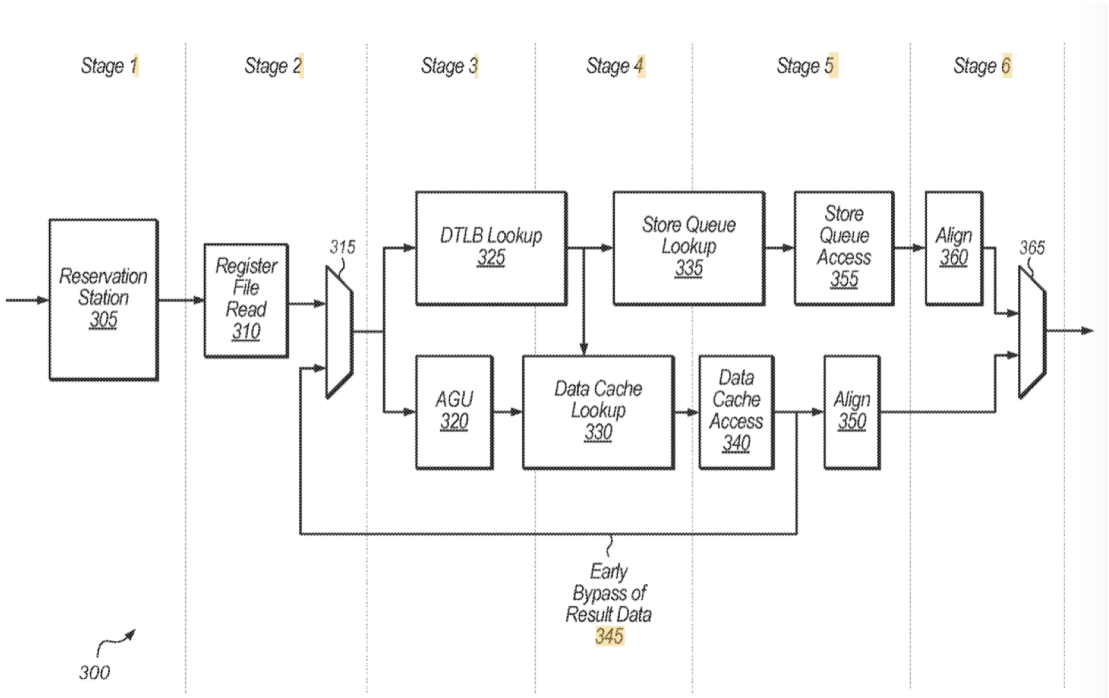


FIG. 3

See how the TLB lookup is performed in parallel with the AGU (presumably assuming the base register

will provide the page and the index will not overflow, as we mentioned for Intel).
This lookup of TLB in parallel with AGU raises questions about how these two are performed in parallel!
One traditional way to handle this has been to perform the sum as a pseudo-sum -- ie generate something that's not exactly the address sum, but which holds all the information of the address, at least good enough for lineID prediction /lookup purposes.
This paper, although based on very old CPU models, is an explanation of the idea: https://www.researchgate.net/profile/Dionisios-Pnevmatikatos/publication/2650802_Streamlining _Data _Cache _Access _with _Fast _Address _Calculation/links/0912f5113b46f98b06000000/Streamlining-Data-Cache-Access-with-Fast-Address-Calculation.pdf , using *or* as their pseudo-sum. Alternative versions of the idea might use xor.
In a sense what's being done is the use of a very simple hash to perform a two-index lookup.

This patent by IBM, (1995) https://patents.google.com/patent/US5532947A , shows an alternative version of the idea (splitting the "sum" into the middle bits of the first operand, the middle bits of the second operand, and a carry from the sum of the lower bits) and why it is useful. To understand the patent remember
(a) IBM number bits backwards, so lowest order bits have large bit numbers, and high order bits have small bit numbers
(b) in the context of semiconductor RAM, a "decoder" is something that converts an n-bit number into an activation of one of 2^n lines.
(Amusing side issue. Look at the name on the IBM patent: Terence M Potter. Guess where Mr Potter has been working since 1998...)

Once you understand the IBM patent, you can at least appreciate the idea behind this Apple patent (2009) https://patents.google.com/patent/US8171258B2 *Address generation unit with pseudo sum to accelerate load/store operations*. Consider eg a TLB lookup (the same idea could be used in a few other places).
Initially it would seem that we have to generate the full address (ie address the index register to the base register) before we can perform the TLB lookup. But suppose we split the two registers into 14 lower bits (the offset) and the higher bits (the page number) and sum these separately. So we have the lower bits (which will be used to access the cache), the higher bits (which are "basically" the page number) and a possibly carry into the page number. How does this help us?
The trick is to remember that the first step in accessing an SRAM is a decoder that converts an n-bit number into one line of 2^n address lines and then, *effectively*, we shift that entire vector of address lines by one if the carry bit is set. (The actual patent is about doing the "shifting" using not a shifter but the appropriately selected output from two different decoders.)
Normally this would be a crazy way to add one, but since the decode to 2^n has to be anyway, we might as well exploit the representation we have.

Although the standard is a physically-tagged, logically-indexed set associative cache, with some form of way prediction (or way knowledge), and that appears to be what Apple implements, other alternatives have been suggested.
Some of these, like the skew-associative cache or the direct+victim cache try to get you most of the win of high associativity while retaining a cache that's small and "mostly" direct mapped. Unfortunately they're no longer relevant once you have the budget to create a large cache (substantially larger than a page).
Another direction to go is to give up on physical tags and rely on logical tags. Although this was tried in the 90s (and widely hated by OS and software folks) IBM revived a much more sophisticated version of the idea for the z14. Their scheme still requires a way predictor, and it's not clear to me that it's any faster than the traditional scheme; but it does save a fair amount of energy.
Essentially one wants lots of ways (not least for capacity), but one would prefer not to pay the cost of multiple ways (ie figuring out, one way or another) the way of a line within a particular set.
In the early 2000s the interesting idea of the Selective Direct Mapped cache was introduced http://www.cs.cmu.edu/afs/cs/academic/class/15740-f02/public/doc/discussions/uniprocessors/power/micro01-way.pdf . This utilizes the (non-obvious) fact that most lines can actually be treated as direct-mapped in a cache; only a few lines collide in a given set necessitating multiple ways. (This is of course the same insight that validates the L1 Victim Cache, or the Skewed Cache). The idea is to store most lines "directly", only using associativity for those (~30% or so) that need it. The Way Predictor now predicts one of "direct mapped" or "this one of n ways". This scheme is more accurate than a normal way predictor because the direct mapped case is easily

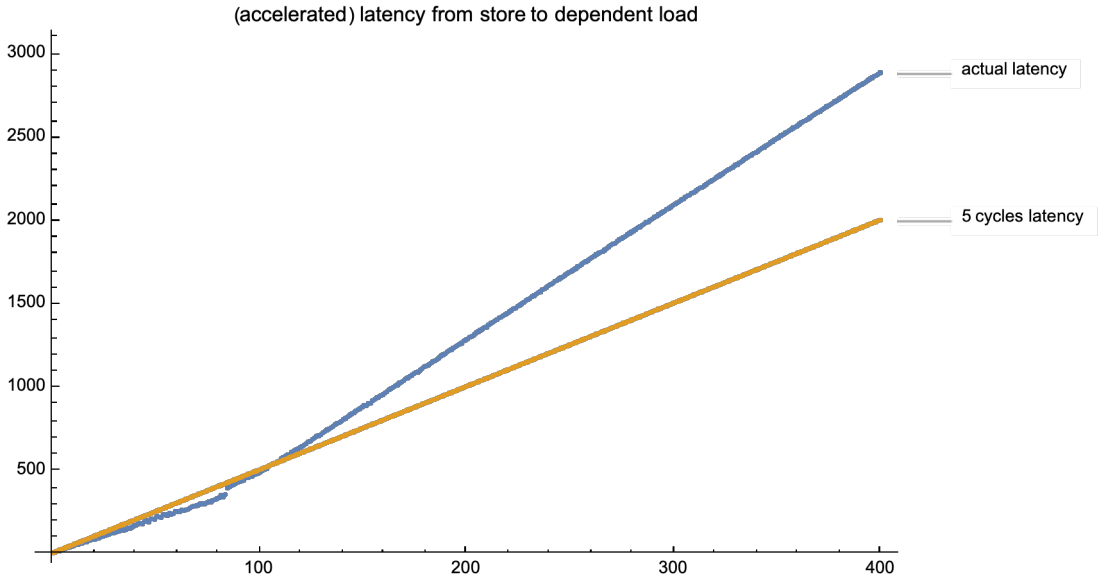
predicted and is very common.

Bottom line is that there's still plenty of scope for experiment beyond the standard cache structure, at least for companies like IBM and Apple that control the whole stack and can, if necessary, modify their firmware and OS's to deal with minor changes entailed by a new such model.

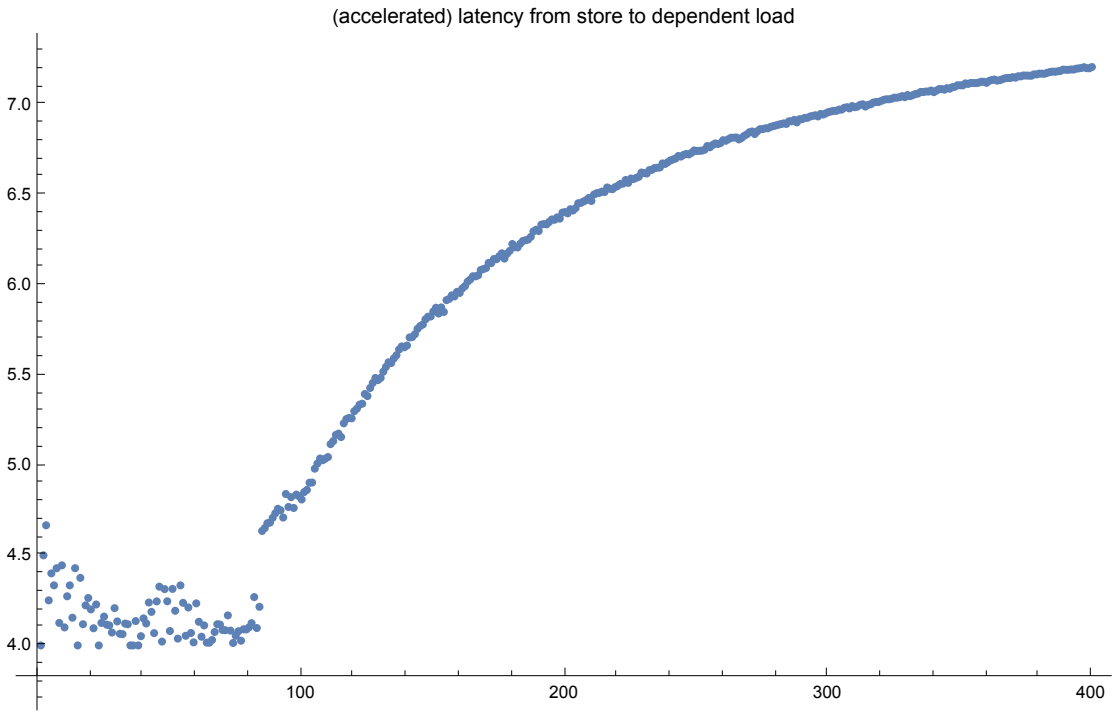
Also the store queue is looked up in parallel with the data cache lookup. This seems energy wasteful (recall one of the papers I referenced suggested saving energy by using the store queue as aggressively as possible as a cache, since it had to be looked into anyway) but the patent also refers to a predictor as to whether a load will hit in the store queue, so presumably the optimal strategy is
- use the store queue to hold as much as possible
- look first in the store queue IF the predictor suggests such a hit, and then in the cache
- look at both in parallel IF the predictor suggests no such hit

This is not the only low-latency case that Apple treats.
Consider this probe (STR x3, [x3]; LDR x2, [x2]) where we initialize x2 as above, and x3 equals x2. This takes 7 cycles per iteration. That's still very nice (the load/store alias detector is doing its job) but that's not ideal. We're taking something like 3 cycles to get the store into the store queue, and another four cycles to perform the load from the store queue.
Now modify it slightly, to (STR x3, [x2]; LDR x2, [x2]). This still takes 7 cycles.
Now modify again, to (STR x3, [x2]; LDR x3, [x2]). This is still effectively the same set (pointless!) operations, storing a value to a given location, loading that exact same value, and repeating. But this time it takes only 4..5 cycles per iteration! Whats the difference? The difference is that the address register (x2) is not being modified. And because of that, this patent, https://patents.google.com/patent/US20130339671A1 , can kick in. Recall our discussion of zero cycle moves (handled by fiddling the Remap table) and zero cycle immediates (same thing, using dedicated Immediate registers)? Suppose you know that
- a recent store wrote register xS AND
- generated an address based on register xA AND
- a load is loading from register xA AND
- xA has not been modified AND
-pS (the physical register corresponding to logical store data xS) is still available
Then you can perform the same sort of zero cycle game, servicing the load by simply Remapping pS into the destination register for the Load.
The patent goes into some detail regarding how this is done, and the various failure conditions.
It's interesting to note that we can see this prediction in action. Look at the curve below. For <~110 load/store pairs we run at the fast pace of 4..5 cycles/iteration, after that we run at 7 cycles/iteration. Presumably the predictor table holding the load/store dependency pairs is sized at ~110 entries.

Out[221]=



Out[222]=



It seems that the ideal case (everything lined up just right) is in fact 4 cycles/iteration, with a fair bit of scatter (predictor sometimes, but rarely, is not invoked?)
Not bad! It's natural to consider this to be a dumb case -- why store data you immediately read again? But It's honestly not that uncommon for it to be simplest for one block of code (a function, or even something isolated within an if block) to store a result to, say, a field of a structure, and the next block of code reads from that field (unaware of whether or how the if block modified the field).

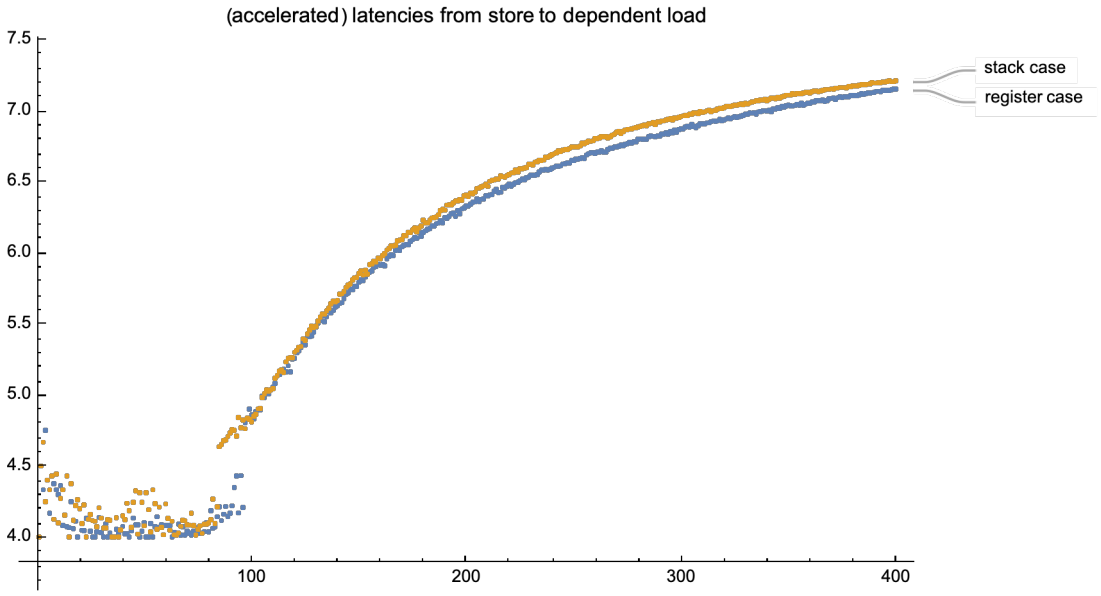
This zero cycle load captures one type of pattern. A different type of pattern is captured in this patent,

https://patents.google.com/patent/US10838729B1 , which looks somewhat like the Intel Stack Engine. The idea seems the same (service a load, that matches an earlier store address, from the physical register file) only specialized to using the stack pointer as the base register. So why a separate patent? I guess the two main differences are

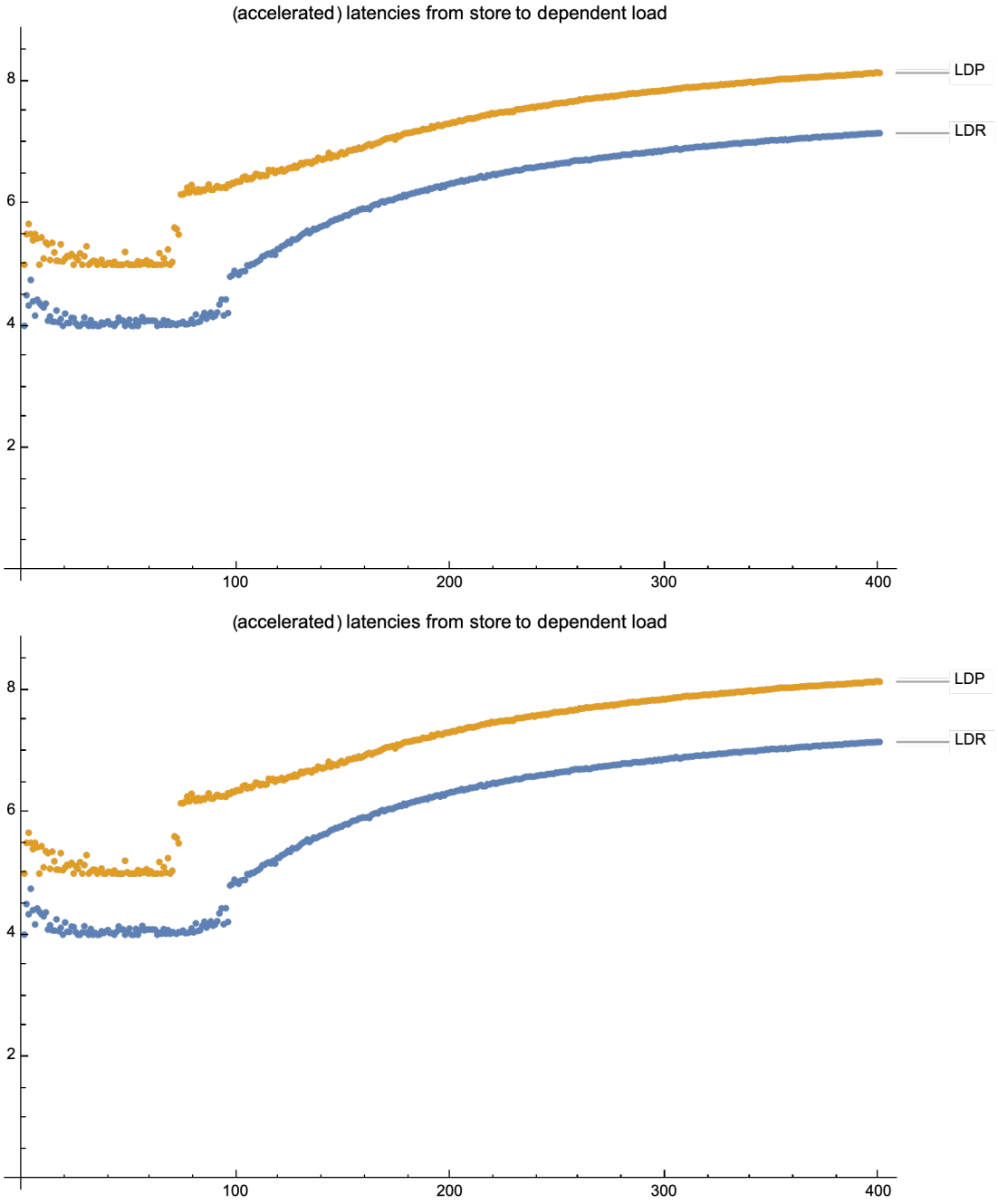
- the stack case can omit the base pointer, thus we get a separate pool of predictor storage that uses fewer bits and matches a common case (not storing non-volatile registers in function prolog/epilog, but also eg register spills)
- the stack case seems more aggressive (because of the register spill case) in allowing for load pair to match, not just single loads.

Regardless of the internal history, we now seem to have a common predictor, with common behavior, for both stack and register addressing:

Out[224]=



Out[226]=



Store pair / load pair takes one cycle longer, and to only have ~70 storage slots, but apart from that the same pattern. (The above is for stack, but register based was no different.)

Suppose we run a mixed probe, with load pair/store pair using a base pointer, and load/store to the stack? In that case we get the lower-latency case up to about 45 probes. This suggests there is a common pool of storage used for zero-cycle loads regardless of the details of whether the base pointer is stack or not, and whether the result to be loaded is a single register vs a pair.

One final question is how many of these loads we can perform per cycle. Suppose we set up two inde-

pendent probes of the form (STR x10, [x0]; LDR x10, [x0]), and likewise with x10 replaced by x11, x0 replaced by x1, and pointing to different addresses (otherwise we're serialized by the common memory address!)

This runs at the same latency, a best case of 4 cycles, (but obviously dropping to the slower rate at ~45 distinct pairs).

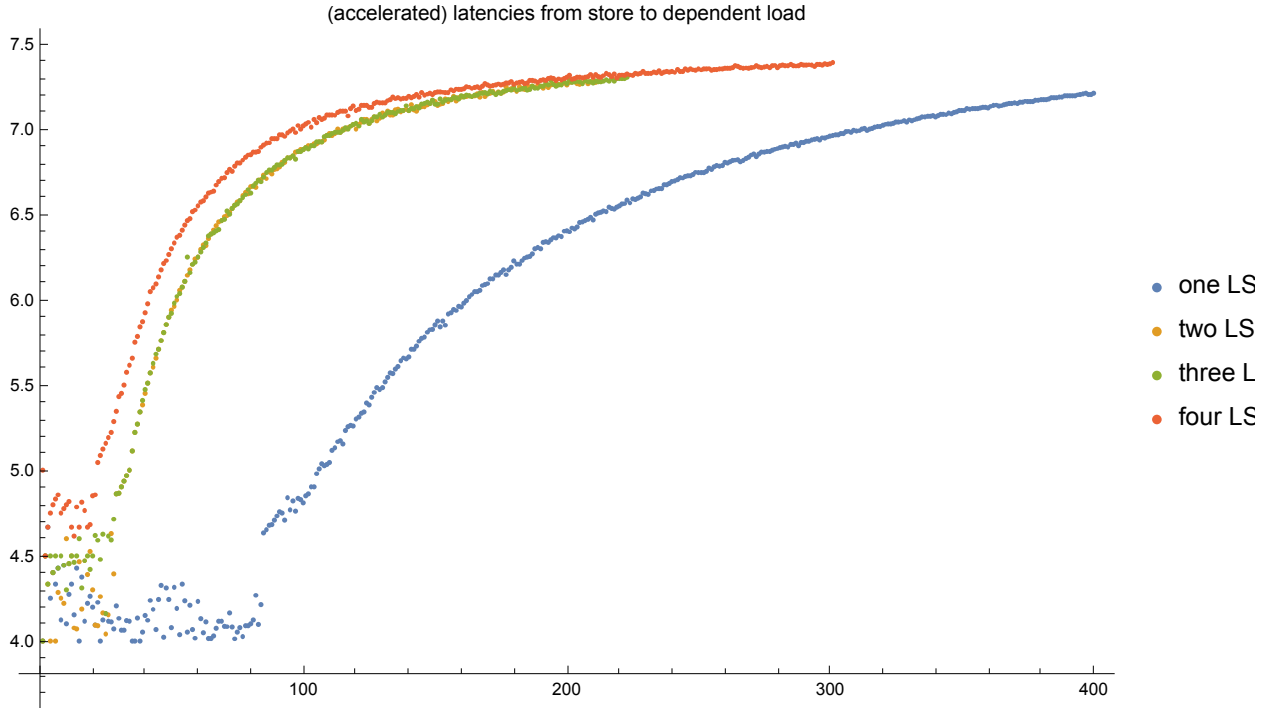
What if we now increase this to 3 such probes? This runs at a (noisy) latency of what looks to be about 4.5 cycles. Slower, but not a full cycle slower.

4 such probes runs at about 4.8 cycles.

If I had to guess, looking at the graph, my guess would be that the system is specced to perform up to three such zero-cycle loads per Rename.

But honestly, I'd be also accept two such loads per Rename as plausible, the graph is not especially compelling either way. One could probably figure this out with a more sophisticated probe but honestly right now I'm too exhausted to bother.

Out[230]=



What further can be said about the L1 cache? The interaction of the L1 cache with the rest of the system requires various storage. Some of this storage is just an address (for example snoop requests coming in from the rest of the system, or load requests from the core), some also requires data (for example line castouts, or stores to an uncacheable address).

Patent (2005) https://patents.google.com/patent/US20070050564A1 describes using common storage for all the various operations, along with multiple queues to prioritize these requests, and a credit system (to ensure that no request type can use up too much of the available storage). To my eye it mostly looks like common sense, and I don't know how advanced it was compared to the state of art at the time, or even now.

However the one thing that caught my eye as interesting is that, as I understand it (the implementation details by now may have changed, but not the overall idea) when a store is made by the CPU to a line that is not present in the cache, the cache does not immediately allocate the line and request it from the system. Rather the store is placed in one of these common storage buffers in the hope that subsequent stores will fill in the line. In the ideal case, the entire line will be filled in and the cache only has to send a "line modified" message out to the rest of the system, not a request for the contents of the line. Even in the non-ideal case, if obvious easy cases (like half the line) are present in this store buffer, then some bandwidth pressure (and energy) can be reduced by only having to transfer half the line.
Variations on this idea are common across ARM cores (and I assume are also present on POWER) and are among the sort of wins possible when your memory model is not especially strong. For example is has been noticed that, even as essentially a first *serious* attempt (a second learning attempt) Graviton 2 has remarkably high memory bandwidth, https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd/3 , and at least part of this is the ability to aggregate stores at the L1 level and present the system with complete line requests, rather than the read-modify-write cycles required by x86. x86 can achieve some of this via special instructions that tell the system to ignore the ordering rules, but that's one more hassle for the programmer. (Of course the ARM programmer then has to be more careful than the x86 programmer when writing to shared variables. IMHO this is the correct tradeoff, writing to shared variables *should* be something you approach carefully, not just quickly written code that you assume works because it looks like it does.)

Also interesting, in terms of principles, is (2005) https://patents.google.com/patent/US20070113020A1 . This is from Apple's early PA Semi days, when PA Semi was still primarily a PPC company. But the exact details of the patent are less interesting than the principle. PPC has an instruction, DCBZ, that allows you to fill a cache block with zeros. This is useful both for rapid fills of zero's and as an ISA-based way of avoiding the RMW cost of storing to a cache line (as described in the previous paragraph).
But one has to be careful. DCBZ, like an any store instruction, can't be allowed to modify the cache (and the rest of the system) until the instruction is non-speculative. So how to make it faster? What the patent describes is that if DCBZ is non-speculative, you execute it as usual, including, specifically, sending out commands to all other caches that they invalidate the line that will be filled with zeros. But if the DCBZ is speculative, you send out alternative cache commands. For example you could send out probe requests asking if anyone has the line in their cache. If no-one says yes, then at the point that the DCBZ becomes non-speculative, you can perform the allocation of the line without any further activity regarding other caches in the system.

Apple has a number of variants of this idea; another is (2014) https://patents.google.com/patent/US9501284B2 . Here the speculative instructions are loads, which are piled behind some sort of memory barrier (in this particular case a Wait For Event). Again the exact instruction (load) cannot be performed, but the specification does allow for prefetching to be performed, so the relevant prefetches are emitted and (one hopes) when the loads actually execute, the data of interest is in the L1D and no further delay is required.

Questions:

Obviously there is a lot about Apple's L1D that's unusual. Let's recalibrate.
The **standard** SRAM model includes these aspects:
- a "block" of SRAM (call it a bank) has a single set of address and data lines, and R/W line. It can support one operation (one read or write) per cycle
- it's as close to square as possible (I don't understand why)
- a read/write operation requires an initial step, called pre-charge, which "excites" a row of bits, once this is done signals can be sent down column lines and, based on the bit stored at the intersection of a bit line and column line, a small voltage will be established which can be amplified by a sense amplifier and the bits that were probed can be read.

So imagine a block of SRAM 128x128 (bits) wide.
We will interact with it via
- a 7 bit address bus (which will pass through a decoder to be turned into one of 128 vertical word activation lines)
- a data bus of the maximum width we wish to read or write in one operation (could be as high as 128 bits, but let's say it's 64 bits)
- some byte enables or whatever that tell us which specific bytes (or bits) of the 128 bits in a line are of interest
- some command signal lines
Operation will consist of sending the 7-bit address along with a precharge command, then, some time (maybe a cycle) later the data bits, byte enables, and a read command.

Now the amount of energy used by the precharge is essentially proportional to the length of a row, as is the time taken for the precharge.
So an obvious next step might be to split this 128x128 block into four 64x64 blocks. Since each row length is halved, it should take less energy and less time to activate just the bits of interest. The downside to this is slightly more complicated design and routing (to feed all the lines appropriately between the four different blocks), a duplication of some of the machinery perhaps the decoder, probably the sense amplifiers). But those are small costs.
So why not keep doing this to get even smaller (and faster, and lower power).
The standard answer is that keeping all these subbanks in sync requires a H clock tree, and that comes with its own costs in terms of power, design, and difficulty in keeping it behaving correctly as you split it finer and finer. And so there's been a (fairly coarse) limit to how small you make your banks.

The above explains the standard model of L1 access.
Supposed you have a standard 8 way set associative VIPT cache. That means that any piece of data
- lives in only one specific set, based on the lower (within-page) bits of the address
- because those lower bits are within-page, they are known independent of TLB lookup
- but that set holds 8 lines (each line in an SRAM row), and the data could be in any of those lines

The original version of the model goes, essential
0: TLB lookup | precharge all 8 rows
1: tag lookup (compare the physical address from TLB with the line 8 tags)
2: depending on which tag matched, read the data from that row
This means 8 energy precharge costs.

So the next model was
0: TLB lookup | read guessed way from the way predictor
1: tag lookup (compare the physical address from TLB with the line 8 tags) | precharge the predicted way (1 row)
2: hopefully, read the data from that row, but if (step 1) indicates a mismatch precharge the *correct* way (1 row)
3*: (maybe) read the data from the correct row

But many aspects of **Apple**'s design suggest that they are operating with a different mindset. I don't know which of the above set of standard assumptions they have modified. It's possible that they are willing to spend 10 or 20% more area or routing wire to design something with very different assumptions.
I cannot really understand there (quite a few) SRAM patents, but they seem to be operating on a mindset that
- isn't quite as concerned with squareness of the SRAM arrays
- uses much smaller SRAM sub-arrays than seems common for other vendors
- these smaller sub-arrays seem to allow them to perform precharge in half a cycle rather than a full cycle, meaning the second half of a cycle can perform read or write, meaning in turn a different sort of timing that allows for sequential tag access then precharge. Maybe this is how they avoid a way predictor (recall that I could not find any timing evidence of a way predictor no matter how randomly I tried to jump around addresses)?

We have as very early patents (2005) https://patents.google.com/patent/US7355905B2 *Integrated circuit with separate supply voltage for memory that is different from logic circuit supply voltage* and (2005) https://patents.google.com/patent/US20070002650A1 *Recovering bit lines in a memory array after stopped clock operation*. Both are low-level circuit patents that don't mean much to me, but seem to indicate a willingness to add additional wires and logic to the standard SRAM array for the sake of better behavior.

Barriers

The issue of barriers is one worth considering.
There is a general pattern one sees repeatedly in computing. We start with operations that are in-order and synchronous. (In order CPU, or synchronous disk writes to a floppy disk.) To improve performance, we convert these operations to asynchronous and allow them to be re-ordered. Which is OK, except

that there are always special cases where we need to preserve ordering.
In the case of IO this happens for databases (including the file system as a whole). To maintain transaction guarantees (ie either all of a set of changes get persisted to disk, or none get persisted) even in the case of power failure partway through the transaction, databases do things like write an initial outline of the changes that will be made to a log, then make the actual changes, such that if there's a failure in writing the actual changes, the transaction can be either wiped from the database, or reconstructed, using what was previously (by definition) written to the log. Obviously for this to work, the changes to the log need to be persisted before the actual changes!
In the case of CPUs, an example is two processors communicating about shared data. Again one wants transactional-type interactions -- I make a set of changes to the shared state, then I flip a variable that says "my changes are done", you see that changed variable, and read the changes I made. Sounds good -- but again it only works if the system does not re-order memory transactions. I need to be sure that once you see the signal variable has changed, any reads you make of the variables I modified will be of my changed values. (This is not the same thing as cache consistency. Cache consistency is about a single memory address holding the same apparent state for all CPUs. What we are discussing is the relationship between changes to two or more different memory addresses, and the order in which I make them vs the order in which you see them.)

Confronted with these types of problems, there is a standard solution which always seems to be the first solution attempted -- and it's always far from optimal! That solution is to add some sort of "sync" operation to the IO API, or to the CPU. The theory is a database will make a set of changes (eg writes to the log) then issue a sync command, then make the next set of changes (eg changes to the actual database file). The sync will force out the first set of changes to storage before the second set even get started. You can imagine similar versions of this for inside a CPU, a sync that eg forces all pending loads and stores within the LSQ to commit all the way to the L1D before the CPU can continue. And in fact Intel have something like this that flushes out cache lines all the way to DRAM for supporting Optane DIMMs (specifically for the database problem we already described).
The problem with sync is that it does far more (and so is far more expensive) than you usually want. All you actually want is that the first set of transactions must all occur before the second set of transactions. You don't care about when the transaction occur, and you have no particular desire to make the whole world wait while you flush all sorts of pending disk sectors or loads/stores. or cache lines, out to a much slower memory/storage tier.

The next level of doing things is barriers. A barrier implements the distinction we described; it says that things that happened before the barrier must complete before those after the barrier, but no more than that. Apple (but not Linux) provides barriers of this sort for IO, and ARM (but not x86) provides them for memory operations. There are different ways you can implement barriers (including treating them as flushes), but the obvious sensible way is to
- mark requests as they come in via some marker that changes when a barrier is encountered and
- not allow any request with a marker later than N to complete until all requests marked as N are complete.

Apple have a patent for this (2012) https://patents.google.com/patent/US9582276B2 *Processor and method for implementing barrier operation using speculative and architectural color values*.
Even with this idea there is something especially cute about Apple's implementation. The obvious place to implement such a barrier (at least to me!) is in the LSU, so that it controls everything is correctly written out to L1. But what Apple appear to do is implement the barrier between the L1 and L2, ensuring that the outside world (ie L2 and beyond) see correct ordering, but allowing more flexibility, and thus more performance, in how the operations occur between LSU and L1! I assume this ultimately controls the ordering of MESI state changes and response to snoops, which is closer to what you *really* want. (Ultimately you don't really care how the loads and stores are ordered to your cache, what you care about is how the ordering associated with those loads and stores is conveyed to the other devices in the system.)
This sort of generational segregation is the best one can do if barriers are your API or ISA, but you can do better!

The problem with barriers is that they still state more than you actually want! You don't actually care about *every* request before the barrier completing earlier than *every* request after the barrier; all you care about is your particular requests (your stream of database changes, ignoring all the other IO on the machine; or the particular changes you make to a shared structure, ignoring all other load stores that happen as you read/write your private storage in the process of changing that shared structure). What you want is a way to tag the specialized requests, and then execute a barrier that only applies to your tagged requests. This is in fact the IO API that Apple provides (IO tagged by a particular file handle). And it is something that ARM is investigating as part of their set of instructions for ordering cache lines out to persistent storage (ie Optane-type DIMMs). https://community.arm.com/developer/research/b/articles/posts/relaxed-persist-ordering-using-strand-persistency.

Once you understand this pattern, you see it everywhere! Consider, for example, this, apparently unrelated patent (2009) https://patents.google.com/patent/US8032673B2. The issue is communicating with peripheral hardware (which can things like sensors, IO equipment, radios, whatever). Naturally for large data transfers one wants to use DMA, but these devices usually need to be configured at boot time (and, much more frequently, maybe every time there is a sleep/wake transition), and that's usually done by PIO, ie writing configuration values to what look like memory addresses but which are actually routed, eventually, to registers in the peripheral.
The obvious way to do this is synchronously, to write a config value, check there was no error, write the next value, and so on. But this is a slow process with long waits between each write.
Slightly better would be to have the PIO controller buffer the writes in some queue and feed them on to the device at whatever speed it can handle. This at least prevents the CPU from having to wait a long time -- but it still means each peripheral is brought up sequentially, so there's some delay until all the peripherals are ready. However the PIO controller can't randomly reorder the PIO transactions because they are designed to happen in a particular order, one item of functionality being brought up at a time!
So we have the same sort of issue as above -- what we want is strand ordering, so that we can label all transactions to a particular peripheral and retain the ordering of those transactions, while allowing

transactions to other peripherals to be interleaved -- each peripheral gets its instructions in order, but the PIO controller sends them out to the peripherals as soon as any device is able to accept a new transaction. In essence that's what the patent is about, a PIO controller, and a labelling of PIO transactions, that allows for reordering between strands but not within strands. (At least this is my interpretation. The patent talks about the transaction ID as being a source ID, but to me that makes no sense; it only makes sense as a destination ID.)

A different way to optimize barriers is seen with (2010) https://patents.google.com/patent/US20110208915A1 *Fused Store Exclusive/Memory Barrier Operation*. The details are doubtless obsolete, but the big picture intuition remains. The problem to be solved is that a common OS pattern involves a spinlock (ending with a Store Exclusive operation and a branch loopback if that store failed) followed by a memory barrier to "publish" the changed state of the spinlock. The intuition is that both operations (the store conditional and the barrier) are expensive because they require a trip out to the *point of coherency*, think the last level cache. The solution is to fuse them together to create a single op (understood by the caches and the fabric) that only requires one such expensive outward trip. (In a way it's very similar to the techniques used by SPDY to try to piggy back as much connection setup as possible into just one or two packet exchanges.)

Equally interesting is when you consider some details this implies.
- One is that the memory barrier, at the point of the fusion, is probably speculative. When you define the precise semantics of this fused operation, you have to be sure that it behaves correctly in that it doesn't do anything that's incorrect if that speculation turns out to be wrong.
- Secondly, one thing this implies is that in response, even to speculative barriers, and even at early detection of such barriers (as early as in Fetch...) one could start performing whatever "push" operations might be required to force out various cached data as required by the barrier. Such early detection and execution somewhat ameliorates the pile-up of such scheduled work that can be caused by such barriers; and it's feasible as long as all that's done is essentially pushing out cached state that's going to be pushed out anyway at some point.
- Third is that the operations being fused are separated by at least one instruction (a branch) and possibly more than one (cleanup after the spin loop). One thinks of instruction fusion is something performed in Decode by observing back-to-back pairs of instructions, but here the fusion has to happen at a later point; the patent suggests performing the fusion in the LSU in which case it will happen so long as the two operations are not separated by an intervening load or store operation.
Using colors to implement barriers obviates the need for some of this technology, but one place that seems like it might still benefit from these ideas (both early "push" and fusing expensive "far" operations) is TLB/page table manipulation.

Memory Ordering Issues

We've described barriers and where to use them, but there are other unexpected issues for the CPU designer arising from multiple cores.
Suppose we have two loads, load A followed by load B, both loading from the same address. Remem-

ber that this is an OoO machine, so load B might execute first, with various other loads and stores occurring between the execution of load B and then load A.
What can go wrong?
First if there are no significant memory events between the two loads, who cares? They return the same value so let them happen in whatever order.
Second, the case we have already discussed in great detail is where there is a store C occurring in program order between A and B. We know how this is handled (that's what the load-store queue is all about) and we know the basic idea: when load B executes, it will check the various stores in the store queue, see that store C affects its address, and wait until store C executes. Likewise store C sees that it has to wait until load A executes. Things can go wrong at a mild level (slightly unfortunate timing) meaning that some of these instructions "collide", the collision is detected, and they have to replay. Or things can go very wrong (some of the addresses required are not available and are mispredicted) in which case at the point where an affected instruction retires, the misprediction is detected, everything is flushed, and we restart at that point. All covered before.
But there is a third thing that can go wrong! What if the following happens:
- load B executes
- because of activity in some other core, the relevant cache line is replaced in the L1D
- load A executes -- and loads different data because it's loading from the changed cache line storing changes made by another core

Note the issue here. The issue is not one of barriers -- we are making no claims about how one address has changed relative to another. The issue is one of timing -- load A is supposed to occur before load B and we have broken that promise. Yes, we broke it in a stupid way, because A sees "new" data as changed by some other core, whereas B sees "old data" before the other core changed it, but that just makes things worse -- our program doesn't expect time to jump backwards between two loads!
So there is an rule in the memory model that seems so dumb that you hardly need to say it, but it's needed for cases like this -- an older load can never see newer data than a younger load.

This is handled by yet concept! Every load has associated with it a "poison bit" and if the cache line from which the load read is modified before the load becomes non-speculative, then the poison bit is set and we need to recover (which might be possible by replay, or might require a flush). This is described in (2013) https://patents.google.com/patent/US9383995B2 *Load ordering in a weakly-ordered processor*; which is followed up by (2016) https://patents.google.com/patent/US10747535B1 *Handling non-cacheable loads in a non-coherent processor*, which generalizes the idea.
(It should be obvious that this mechanism is rather coarser than strictly necessary, but this should be such an uncommon case that correctness is what matters, no point in spending extra resources to make it finer-grained and faster.)

Obviously these load and store queues, specifically the comparison of a new load or store against all the other items in one or other of the queues, eventually costs energy, no matter what smarts we use. However some of these tests are sufficiently rare that we can often avoid having to perform them. The

above poison case is an example. Once a load queue entry is poisoned, we have to test all subsequent loads against the load queue to check that we aren't hitting a poisoned entry. But this is an unnecessary waste of energy in the usual case that nothing in the load queue is poisoned. Thus the LSQ tracks a single bit as to whether any entry in the queue is poisoned, and behaves differently in these two cases. This is described in (2019) https://patents.google.com/patent/US20200264888A1 *Content-Addressable Memory Filtering based on Microarchitectural State*, which includes a few other similar such energy saving possibilities (for example tracking a single value representing something like "oldest age of the valid loads in the queue"; if a store address now comes in that is older than this age, then there is no need to probe it against the load queue for collisions.
The common theme in both of these (and all) the cases is to find a way to collapse the usual (or at least a common) state of the queue into a single number than can be tested, rather than having to test every entry of the queue.

Prefetches

Prefetching is, of course, a multi-dimensional subject with so many variations possible! Even so, we can see some of Apple's thinking by working through patents.
The first patent, (2006) https://patents.google.com/patent/US20070294482A1, is very simple but even so is interesting. The impression I get of most CPUs is that Prefetch is added as an afterthought to an existing design, so that the Prefetch logic is associated with the L1D, perhaps on the far side (ie what is observed is the stream of L1D cache misses).
The Apple design places the Prefetch unit in the LSU. This has the advantage that it sees the entire load/store stream -- and the disadvantage that it has to filter that entire stream to extract relevant prefetch data (but of course it's easier to filter away data you don't need than to try to extrapolate data you don't have!) This early design also reuses the LSU to submit the actual prefetch requests, by waiting until the LSU is not busy, at which point the prefetch request (presumably encoded as a page address and an offset) is submitted to the AGU. The interesting consequence of this (obvious, though apparently not exploited at the time of the patent) is that the address stream is seen in virtual, not physical, space, so that it's trivial for the stream to cross page boundaries.

Along with standard (data+instruction) prefetches, one can also imagine prefetching TLB's. This should be complemented by a caching system specific to page tables (ie holding various elements of the tree of page tables from the root down) and such techniques are surveyed here: https://www.cs.rice.edu/CS/Architecture/docs/barr-isca10.pdf; presumably Apple uses one of these but I have so far found no evidence. Orthogonal to caching page table intermediate data is the issue of caching the leaf PTE's, which is more or less equivalent to prefetching PTE's, and this is discussed in a very primitive form in (2009) https://patents.google.com/patent/US8397049B2. At this stage, the prefetch consists primarily of loading a block of PTE's into the MMU on a single PTE miss, and looking through those prefetched entries on a TLB miss, along with a rough idea of defining a basic strided stream of PTE entries.
Mostly interesting insofar as it exists, but one more interesting issue is mentioned on the side, namely that the translation requests seen by the MMU/TLB are tagged by type (as code or instruction, but also

perhaps as type of GPU item like texture, tile, or pixmap) and the MMU can decide, based on this tagging, how large a block of PTE's to request as a load from the page tables (and retain as prefetched data).

By early 2012 we have advanced to https://patents.google.com/patent/US9098418B2. We now have
- an L2 TLB (not present in 2010)
- prefetch that's co-ordinated between L1 and L2.
We still have a dedicated prefetch unit in the LSU, but this unit is now looking for more types of patterns (still apparently only streams, but with some flexibility in terms of the stride rather than fixed stride). This prefetch unit is also communicating with the L2 cache to co-ordinate loading streams into the L2 and then into the L1. This communication now includes temporality knowledge, ie the extent to which a line is reused after its first use. (This can be used to decide whether or not the prefetched lines should only be retained in L1, or should also be retained in L2 on the way to L1; or, even if the lines are retained in L2 [which possibly means casting out pre-existing useful lines in L2] doing so in a way that they are first to be cast out when a new line needs to find a location).

By late 2012 this advances to https://patents.google.com/patent/US9047198B2, which adds (like we saw earlier for the L1) TLB translation for the L2 prefetches, something that has to be done in advance of the L1, since the L2 is loading a few cache lines ahead of the L1. (The patent states that this same co-ordination could be extended up to the SLC/L3, but most of the wording suggests that this was not in fact being done for the implementation at that time.)
One interesting question regarding this fancy L1 machinery is the extent to which it was, at this time, co-ordinated with I-prefetch, and how that was done. Perhaps there was an independent I-prefetcher associated with the Fetch unit, which likewise co-ordinated with the I-TLB and the L2, but the two ran separately apart from the obvious fact that you'd prioritize I prefetches above D prefetches?
In the most modern cores, one would hope that I-prefetch is in fact co-ordinated with branch prediction and aggressive run-ahead fetching.

We get a glimpse of a new prefetcher, (somewhat) beyond stride, in (2016) https://patents.-google.com/patent/US10133571B1. This patent is about something very different but includes the throwaway remark that the prefetcher can be an AMPM prefetcher.
What's that, you ask: (2011) https://jilp.org/vol13/v13paper3.pdf *Access Map Pattern Matching for High Performance Data Cache Prefetch.* Essentially it still looks for stride patterns, but the *mechanism* by which it detects such patterns is much more robust to varying the order in which the loads/stores walking the stride pattern are performed.

Cache to rest of system

L2 snoop filtering of L1's

The first patent giving us some feel for the L1 caching setup is (2006) https://patents.google.com/patent/US7752474B2, which suggests about as simple a (snooped, coherent) cache as you could imagine. The patent is mainly interesting insofar as it compared with (2010) https://patents.google.com/patent/US9317102B2. By this second patent the following improvements are visible
- we now have a Coherence Point on the fabric. For the earlier design, all snoops routed directly from one core to both L2 and the second core. Now the L2 holds duplicate tags for the L1's of the various cores attached to it.
The obvious consequence is that a snoop from one core can be handled by the L2 (which can filter out most snoops as being irrelevant to any other core, and so can discard them); this saves a little energy and maybe even allows for omitting some of the extra snoop handling machinery from the L1 (if snoops are now rare, maybe it's OK to handle them via a slower path?)

- a second improvement is that the L1 now has various counters of things like how many valid blocks and how many modified blocks it contains, and presumably these counts can be used to inform decisions as to how rapidly to allow the cache to be put to sleep.

One problem I saw with the 2006 design was that flushing the cache seemed to require cycling through every way of every set (presumably one per cycle) asking that way to flush itself if it is modified. I don't see much of an improvement to this scheme in 2010. Of course you can exit slightly earlier (at the point where the count of modified lines goes to zero) but beyond that the same mindless walking through every lineID seems to be necessary.

Let's return to snoop filtering and power saving.
The less obvious consequence of having L2 performing snoop filtering is that snoops can be handled by the L2 while the L1 sleeps, and mostly without having to wake the L1. This is a somewhat tricky business.
For super-short naps (rest of CPU is working, but no load/stores this cycle), you'll probably want to save power just by freezing the clock going to the cache, but you'll still be paying leakage power.
For slightly longer sleeps, the CPU might sleep but you don't yet want to pay the cost of losing your cache data, so you allow the cache to remain powered up (just not clocked). Under these circumstances, you really want the L2 to perform snoop filtering so that the CPU and cache don't have to be woken up.
But at some point the sleep looks like it's lasting long enough that we might as well flush the L1D (ie write out all the modified line) and cut power completely. At this point we don't want to wake up the main CPU to flush cache lines; that's definitely not ideal.

So by 2013 we have https://patents.google.com/patent/US20140195737A1. Now we have an asynchronous engine that allows the main CPU to partially power down while the engine copies all modified lines out to L2, before fully powering down the core. This might seem an obvious addition, but it's not as simple as you might think because of that eternal problem of coherency. Consider, for example, what happens if you receive a snoop on one of the modified lines that you haven't yet written out... You

can't *just* walk through all the modified cache lines, you have to maintain and handle some minimal level of snooping by the L1D and async engine right up till the moment of power down.
Interestingly the flush engine is associated with the L2 core rather than the L1 cores. Thus, in a sense, it pulls modified lines from the L1, it doesn't push them to the L2.
How does it know what lines to pull? Because (as mentioned) the L2 maintains duplicate tags for the lines of all the L1's, which it also uses as a snoop filter!

There are two further tweaks you can make to this system.
The first is that if *all* the clients of the L2 go to sleep, then after a while it makes sense to also shut down the entire L2. Once again you have the issue that you need to flush all the modified lines up to SLC. So the flush engine gets repurposed for that job, now walking the tags of the L2 looking for modified lines, until it has matched the count of modified lines that has been maintained.
The second tweak is that once an L1 client has lost power, there's no point in snooping for that client any more, it's like that cache does not exist! So Apple pulls power on the duplicate tags that were being maintained for that client: (2013) https://patents.google.com/patent/US20140189411A1.

drowsy L2

The L2 as a whole is managed similarly but with finer control, as a so-called drowsy cache; banks that haven't been accessed for some time are put to sleep, enough to retain data, but requiring a cycle or two to wake up on access.
The unit that is actually put to sleep is a way in each set. Think about this. If the L2 is, say, 8-way set associative, then each set holds 8 ways. If, physically, the appropriate ways (way one, way two, etc) are each stored in one of eight independently powered banks, then one of those banks can be put to sleep (thus making the set fully active for seven ways, while to access a line in the eighth way will take an extra cycle to wake up the bank).

A specifically interesting thing about how Apple does makes this choice to sleep is that they
- characterize the L2 by how leaky it. This will be one of those things that varies with manufacturing, some wafers just having all the transistors slightly better quality.
- measure the temperature just before the bank is to be put to sleep.
Based on the temperature and the leakiness of the L2, a an idle count is established. The more aggressively the L2 is leaking current (sub-optimal transistors, or running hot) the lower the idle count is set. So you can think of it as rather than saying "we will wait 100 cycles of idle before we sleep a bank" it's like Apple is saying "we will wait 1 microjoule of energy wasted in idling before we sleep this bank". (2014) https://patents.google.com/patent/US9513693B2.

More aggressively, you can power down (rather than sleep) that whole bank and have the cache look 7/8th as large (as many sets as before, but only seven ways in every set). And of course you can take this further, eg powering down two ways, and sleeping three ways while three ways of every set remain

active.
Like everything, there are complications to this once you start thinking about how to actually implement it! As detailed in (2014) https://patents.google.com/patent/US20150309939A1.

Details have surely changed since then, but at that time Apple's L2 had 12 ways, and these were split into three separately powered collections, so that either one third or two thirds of the cache could be powered down.
The first complication is deciding whether to power down a third, and if so, which third? Remember that lines go into whichever was the least used way of a set, so that after some time the four least used ways of this set may be very different from the four least used ways of some other set...
There are two orthogonal sets of three counters. One set counts the number of references to each of the three physical banks. The lowest value tells us which bank will be powered down.
Separately we have three counters that count how many references have occurred to the four MRU lines, the 4 LRU lines, and the four midRU lines across all sets. If these indicate that usage is concentrated in one, and two, thirds of these three ranges, then (if other conditions are also satisfied) the decision is made to power down a third.
Once you decide which third to power down, you of course have to flush all modified lines before the power down. Is that all you do? In principle you could imagine something like, for each set removing the least used lines from the way groups that will not be powered down, and copying across the most used lines from the way group that will be powered down. You could also modulate this with via tweaks like "if the line exists in an L1, then what the heck, let's just toss it, it'll bubble back to the L2 at some point as a victim".
The patent suggests that a cost is calculated for each way group that incorporates how much flushing is required, how many lines are in L1's, and how busy this way group has been, to decide the optimal way group of the three to power down, but does not suggest this next step of moving MRU lines that are unlucky enough to be in the power-down way group. Of course this is an obvious future improvement...
Finally, at some point you have to decide when to re-power-up the pieces of your cache that were powered down. Obviously the first signal is that you are missing frequently in the L2 (ie your L2 is too small). But there are a bunch of subsidiary tests that essentially try to figure out if more capacity would really help (ie if you are missing because of compulsory misses, or because of streaming type behavior, increasing the cache size won't help much; what you want to detect is misses that are occurring that are somehow associated with reused lines).

There's an additional less obvious power saving available. Just like the L2 snoop-filtered the L1, the SLC snoop-filters the L2, using a set of duplicated tags stored with the SLC. These duplicate tags are also powered by way, so that when a way group of the L2 is powered down, then the same way group of those SLC duplicate tags can be powered down. (2015) https://patents.google.com/patent/US9823730B2.

compressed cache (sorta...)

Even better than a sleeping cache bank is a powered down cache bank! Consider this patent (2017) https://patents.google.com/patent/US10691610B2 *System control using sparse data*. The patent suggests multiple things but the starting point is

- detect “sparse” line writes
- don’t perform the write, instead note the line (eg by setting a bit in the cache line tag)
- service reads from the line by not performing a read and instead providing the “sparse” data.

So, obvious use cases
- tag all-zero lines in L2 and SLC/L3. (They talk about generic memory, but the diagrams, to my eye, look L2-'ish. Apple’s L2’s tend to have a 3-way replication, as we saw in the drowsy cache patent.)

- they open the way for a compressed cache (at L2 or SLC/3). It’s not a great fit once you get past “all zeros” and perhaps “all-1’s”; because compressed cache really wants something different (eg mechanism to read/write half a cache line). But, baby steps.

- in principle you can extend this up to RAM. In practice, I’m not sure -- I can’t see a feasible way of recording the sparsity (bloom filter?) But maybe a flag in the TLB/page-table that records “all zeros”?

Equally interesting is the storage side. Essentially
- maintain one cache bank as the “loser” banks holding the invalid lines and the “all zero” lines
- as long as this holds, that bank can be kept powered off (even better than just sleeping!)
- once you have to break this (ie store real data in that bank) keep track of stats and, when it makes sense, repack the good data to a different bank, and re-power-down.

This is a more ambitious and more sophisticated version of the previous drowsy cache. Obviously powered down is better than sleeping; and providing machinery in the cache controller to move lines between “active”, “schedule for sleep”, and “powered down” banks makes the sleep and power downs last that much longer.

It’s probably misleading to call this a real compressed cache; but it does put in the place the first half of the sort of machinery one would like for real compressed cache support. As soon as you here the term *compressed cache* you surely get the idea, but here’s one example of the sorts of ideas being floated: (2012) http://www.cs.toronto.edu/~pekhimenko/courses/csc2231-f17/Papers/BDI.pdf *Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches.*

shared L2 and its consequences for shared frequency

This is small and no longer important, but it’s interesting history. Look at (2017) https://patents.google.com/patent/US10147464B1 *Managing power state in one power domain based on power states in*

another power domain. This sounds horribly technical, but in fact it describes the A10 (and only the A10). The idea is we have performance cores with an associated L2, and efficiency cores that use the L2 of the performance cores as their L2.
Doing this reveals a number of problems. The most obvious, and easily fixed, is that you need to be able to keep the L2 powered up even when the performance cores are asleep, as long as the efficiency cores are working. But the bigger issue is at what frequency do you run the L2, and the performance cost of having to move traffic between the L1 frequency domain and the slightly different frequency of the L2 (meaning buffering across the domains), a problem with no great solutions.

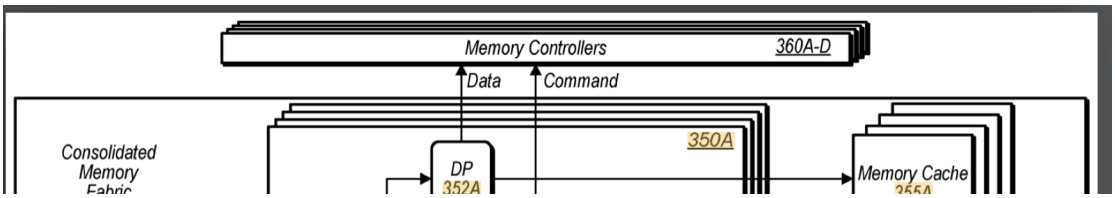
All this is historically interesting, but raises a meta-issue. When cores share an L2, either they all run at the same frequency, or the cost to access the L2 is substantially higher because of buffering across clock domains. So what's the right tradeoff? Later below we'll see a very complicated (but clever!) Apple patent for optimal OS scheduling given the constraint of multiple cores having to run at the same frequency. But more generally, as we move to from the simple world of one performance and one efficiency cluster, is four the optimal number of CPUs sharing an L2 (and thus sharing frequency)? There are clearly advantages to a large L2, and a shared L2; but there are also costs. Is a better tradeoff perhaps three cores sharing an L2, and the future low-end being something like two performance clusters (ie six performance cores)? Or two cores sharing a substantially smaller L2, and all the two-core clusters (including the efficiency two-core clusters) sharing a large CPU L3 (distinct from the SLC), operating on its own frequency domain, and we just accept the cost of crossing to that frequency domain?

non - inclusive non - exclusive

Also in the field of snooping we have (2018) https://patents.google.com/patent/US20200081838A1. To me this seems most interesting insofar as it clarifies the following issue:
Intel have mostly used *inclusive* caches, eg the L3 contains all the L2's. The downside of this (cache space wasted on replicated data) is obvious, but the reason for it is that it makes your snoop filtering easy -- the L3 can easily filter snoops for the L2, in that if the snoop misses in the L3, then there is no need to send it down to the L2 and test that maybe the line can be found there.
The next easiest alternative is an *exclusive* policy, as used by AMD. Depending on how this is done, you may or may not be able to snoop filter in the outer level cache, but what is easier is that the cache line state modification, because the cache line (and so its state flags) can only exist in one place. Exclusion won't cost you space, like inclusion, but it may cost you some power to enforce that a line is always *moved* (not just copied) from one place to another.
What Apple do, however, is neither exclusive nor inclusive. To implement this, while retaining the advantages of snoop filtering by a higher level cache, the tags of lower level caches are duplicated in the higher level caches. As far as I can tell this happens in both the L2 and SLC, and the idea seems to be something like:
- core to core snooping will mostly occur within a core complex, and can be handled well by having the

L2 for that complex snoop for all 4 processors of the complex
- snooping between different types of IP (like NPU snooping changes in a GPU cache) will be filtered at the SLC level
- left uncertain is whether snoops between the Performance Core Complex and the Efficiency Core Complex are treated as a special case or handled by the SLC.

Regardless, the consequence of this is that there are lots of duplicate cache tags! Obviously it's a fair bit of design work to ensure that they are kept in sync between the "original" cache (eg an L1) and a duplicating "higher cache" like the L2. One can see the appeal of the simpler inclusive or exclusive models!
Many of the Apple cache patents have to do with various aspects of these duplicate tags, for example the one referenced at the beginning of this paragraph has to do with sequencing issues. When a snoop comes in, the first thing a cache will do is compare the snoop to its tags, to see if the snoop matches a cached line. But does it first look at the cache-native tags, or at the duplicated tags? The patent says "look at both simultaneously", with a bunch of complications that then result depending on if both tags hit (eg the line state has to be changed in both the L2 and (perhaps multiple) L1's), if the line is locked (eg through a load exclusive instruction), and whether the instruction has passed the "global ordering point".
And interesting side aspect to the patent is that its Fig 3 shows what is probably something like the current design of the SLC.



The area within the dotted line is essentially a single pool of common queues which all requests flow through. Four arbitrators (in principle there could be more, but four seems reasonable for an M1 class SoC) route these requests to one of four "slices". No-one ever says exactly what they are doing here, but an obvious way to think about this is to consider (physical) addresses as having bits 0..6 specify a byte within a 128B line. Then in the simplest case, use bits 7 and 8 to choose one of the 4 arbitrators and slices. If you're willing to spend an additional few delays, you can do a more complicated address hash and (hopefully) still get decent spreading across slices even if your lines are sequentially separated by 256B or 512B. But I don't know how this spreading of the address space over slices is done -- maybe there is a good reason to spread at the page level rather than at the line level, so that all lines of

We can be a little more precise about this inclusion/exclusion. If we trust (2007) https://patents.google.-com/patent/US7702858B2, what that says, among other things is that the L2 is
- inclusive for instructions (I don't really see the point of this. I can make a justification, that code that's loaded by one core is often code that's used by other cores, either via a multi-threaded app or in shared libraries and the OS; but is that common enough to be worth doing things this way?) There are other ideas in academia for how best to handle instructions in an L2. Because it's much harder to hide the delay from an I-cache miss than a D-cache miss, you want to prioritize I lines over D lines in L2 one way or another, for example be slower to "age" I lines down from MRU to LRU.
- victim for data. In other words a data line is loaded directly from its source (let's say DRAM) into L1, without being stored in L2. At some later point, when the line is removed from the L1D, because it's become the lowest priority line and a new line is loaded, the line is then moved to the L2. Subsequently it could be reloaded by the L1, at which point it would be supplied by L2 and would be in both caches.

(The actual content of the patent is interesting, though surely now changed and essentially irrelevant. Suppose a requester makes a request. Recall that this is before the (2010) first snoop filtering patents. So the request goes to both the L2 and the two L1's (and any other caches on the SoC). Theoretically the system should wait for everyone to respond to the snoop before the next step of having one of the agents, if possible, supply the data. The patent outlines circumstances under which the L2 can return the line from its contents before having to see the responses by the L1's underneath it.)

BTW it seems likely that, from at least the A6 and A7 days, Apple was using MOESI as their cache protocol. You can find the details on wikipedia, but one line summary is that MOESI augments MESI with an Owned state which allows the cache-to-cache transfer of m*odified* lines. In other words CPU A asks for a line, and if that line is held (modified) in CPU B's cache, then the protocol allows for the transfer of the line. Compare this with MESI (which allows no cache to cache transfer), and a protocol which allows *unmodified* lines to be copied from one cache to another (this was introduced by IBM as MERSI, and then used by Intel as MESIF).
Obviously best of all would be a protocol that allowed (as much as practical) all lines, modified or not, to transfer cache to cache rather than having to be pulled in from a higher cache or DRAM, and such a MOESIF protocol is possible.

You can keep adding more and more states if you're willing to pay the complexity cost; primarily to try to clarify things like "yes this line is shared between multiple cores, but they're all on the same socket" vs "this line is shared between multiple cores on different sockets" -- essentially think about all the different levels of cost involved as you might have caches that share an L2, then that share an L3 (so presumably are on the same socket) then that are on different sockets. The more your protocol clarifies these different degrees of sharing, the less you might have to wait in response to any particular broadcast of "I am about to do something with this line; does anyone out there care?"
If you can't get enough of this stuff, IBM is the master, with extremely complicated protocols given their big systems that go up to L4 caches, and consolidate multiple packages across multiple boards!

Even Apple has got into the game: (2009) https://patents.google.com/patent/US20100235586A1 *Multi-core processor snoop filtering*, though I've no idea what this was used for -- presumably not for anything iPhone related, so some sort of custom cache controller for the dual-package first generation mac pro's?
The idea is, however, interesting and sensible. Most pages are never shared, so suppose we indicate in the page tables, propagated to the TLB and then to the cache, that pages are not shared. This would mean that interactions with cache lines of those pages could avoid many snoop broadcasts, saving energy and bandwidth.
This seems like the sort of thing very appropriate to Apple – co-ordinated changes to the OS, APIs and HW ,that would be impossible for most vendors. So maybe it's present in the M1?
This is followed by (2011) https://patents.google.com/patent/US8856456B2 *Systems, methods, and devices for cache block coherence*, still mac rather than iPhone related, but specifically discussing the cost of coherency between CPU and a high-bandwidth device like a GPU... The extension beyond the previous page-based system is that many lines can, in principle, be shared, but in practice they are not; eg they are constructed in the CPU, loaded by the GPU, removed from the CPU's cache, and the CPU never cares about them again. Essentially from the pattern of snoops, and other shared info, each device builds up info not only about what lines it holds, but also something about what lines other devices hold so that, for example, if it knows that no other device can be holding a particular line, snoop broadcasts related to that line can be suppressed.
There are also some nice details utilized by the scheme. For example initially (non)sharing is tracked at a high granularity, but stats are maintained and if these warrant it, the granularity associated with an address range is reduced, to allow for more fine-grained tracking of (non)sharing.

maybe move here the stuff about SLC dual -processing piepeline?

manually managed cache

When we use the word cache, we tend to think of a *transparent* cache, in other words one that automatically decides which lines to retain and which to remove. But manually managed caches are another option, and useful in a variety of situation. One case is structured data access, where the pattern whereby you will access the data is well-defined (common in DSP code, or image processing code). Another case is code that may be called infrequently but you want it to be low-latency when called (Apple give the example of interrupt routine code).
To handle these Apple have provided a variety of manual controls over the years. The traditional way of doing this is to allow some lines to be locked in an L1 or L2, easy to implement.
But Apple, as usual, has a much more interesting setup:

The most recent scheme is (2019) https://patents.google.com/patent/US10922232B1. The idea here is to allow a cache (they seem to imagine this as a possibility from L1 up to L3, and give examples using L1) to have certain address ranges mapped (via in-cache registers) to some lines of the cache.

An obvious question is “what is the difference between locking a line in a cache” and “mapping an address range to a line in a cache”, except that the second seems more complicated?
I think the win is for *ephemeral IO* data. Consider a standard network stack. Data comes in from some hardware, hopefully via DMA, into DRAM. That DRAM buffer (owned by some low-level software like a driver) gets copied to the network stack which may copy it a few time each time stripping off headers and performing some level-appropriate (IP, then TCP, then HTTP) processing, finally copying it across the OS boundary into a user buffer. There’s a lot of copying, and so there has been a continual push to reduce the number of copies to zero, basically for every level to agree on a protocol for how to allocate a buffer, how to transfer ownership between levels, and how to strip off headers or tails by manipulating pointers associated with the buffer. This all sounds great, but once this was all implemented, the results were disappointing; an improvement yes, but not nearly as much as hoped. The essential problem is that the first step, moving the data off the hardware (disk or network) is done via DMA -- which dumps the data in DRAM. Every subsequent copying step is minor in cost compared to the initial cost of moving the data from DRAM to cache. If you eliminate all the copying the OS-specific part of the operation looks fast; but your app is not much faster because now, when it starts processing the data buffer it receives from the OS, every line of that buffer causes a miss in L1 -- you have just moved the primary cost from occurring within the OS to occurring within the client :-(
But suppose that I can designate the buffer address range as part of a cache (either L1, L2, or SLC) and have all the pieces along the way (the memory controller and all the cache controllers) understand what has been done. Now when the network or disk device performs its DMA, the data is dumped directly into a cache, and we no longer have to pay the cost of the initial copy from DRAM to cache!
(One can imagine wilder use cases for this -- future macs will have multiple processor complexes, all with large L2’s. When only one core complex is being used, one would want to derive some sort of value by using those additional L2’s...)

The technical focus of the 2019 patent is the fact that both memory and IO can simultaneously enqueue requests into this cache which is also providing a “memory range”, and that the ordering in which these requests are serviced has to be handled carefully otherwise deadlocks can ensue.
With this in mind, we can look at the much earlier (2009) https://patents.google.com/patent/US20110010504A1. Superficially this looks like the same idea, marking part of a cache as an address range. But there are many more restrictions (and some interesting background ideas mentioned).
As restrictions, the idea seems limited to the SLC, and is apparently not available to IO (except as very restricted DMA from DRAM into/from the SLC). The idea seems to be like I suggested at the start, provided for structured data patterns. So you can request a block of “fast memory” of a certain size and have that allocated in SLC rather than RAM for your temporary use case.
As interesting ideas, the patent makes three points.
First is that this storage isn’t used like traditional cache storage, and so doesn’t require tags; rather a few range registers in the SLC can route addresses as appropriate to this storage. So while this is nominally part of SLC, it kinda lives on the side, invisible to code that doesn’t explicitly take advantage of it, and more dense because of no tags and many fewer (no?) associated line state flags.

Second is that while the requester can ask for memory that matches an existing address range (with flags that will optionally copy the data in DRAM into the SLC at the start and/or copy it back to DRAM at the end), an alternative is to ask for an “invisible” address, namely an address that’s in the middle of the address map, above the address range that’s mapped to DRAM, and below the address range that’s mapped to various bit of IO and OS business.
Third is that one initially tends to think of this as something to be used by CPUs, but in fact Apple appears to have in mind as primary clients things like the GPU and the ISP (as I said, structured data access...)

Ultimately the use case for this earlier patent is complementary to the use case for the 2019 patent, and I wouldn’t be surprised if both are present on the M1.

Now put together some of these various ideas for unusual ways to use the SLC. How further can we save energy? Look at (2013) https://patents.google.com/patent/US9261939B2.
The standard model for a computer (mobile or desktop) has the GPU generating data that is written out to DRAM, and the display reading that data from DRAM, both doing this say 60 times/second. So both directions we are paying the energy cost of reading/writing off-chip to DRAM. What if we could avoid that cost?
The 2013 patent does this in a limited way. When the system detects that the contents of the display buffer are static, the display buffer is copied to the SLC, and the display controller runs out of SLC rather than DRAM.
And there’s even more that can be done! (2013) https://patents.google.com/patent/US9396122B2 points out that now you are using the SLC as framebuffer memory, you know how the data will be accessed. And you can time the sleeping of banks so that only the specific bank that needs to be read by the display controller while painting a particular section of the screen is fully powered up, all other banks are put to sleep.
(How do you discover the screen is static? You test the obvious things like changes to compositing buffer addresses, and the compositing queue instructions, but the main one is you calculate a CRC from the bytes read as you paint the display, and ensure that this CRC remains unchanged for N frames. (2013) https://patents.google.com/patent/US9058676B2. The rest of the patent is uninteresting in that it’s an early version of the static frame idea, before using the SLC; this early idea was to store a lightly compressed version of the static frame in DRAM, so that loading it and decompressing each time cost less than loading the data from multiple compositing buffers and merging them together.)

This is a nice way to get auxiliary use of the SLC under conditions where it’s not otherwise being used, but one can surely do much better. I would not be surprised if modern Apple SoC’s have permanent dedicated on-SoC storage for the primary display of the device, from at least Apple Watch up to MacBook Air, something like a block of memory-addressed (untagged) SRAM within the SLC where some part of it is configured to act as display storage (sized as appropriate for the device), and the rest is available to the OS for the sorts of use cases suggested above.
(2020) https://patents.google.com/patent/US10972408B1 is hardly dispositive, but it does include this

language “The graphics output information corresponds to frame buffers accessed via a memory mapping to the memory space of a GPU within the video graphics controller 140 or within one of the processors 150A-150D”…

bandwidth to caches and DRAM

Point is we see the SLC as split into four "units of addressability" meaning that this probably determines the maximum bandwidth -- four requests to be serviced by the SLC per cycle. Each slice is surely aggressively sub-banked for purposes of power (allowing the individual subbanks either to sleep or to be fully powered down).

Note this doesn't mean the SLC can provide 4 different lines to 4 different clients per cycle! I expect the design point is that most of those 4/cycle requests are expected to be address-only requests (ie snoops and line state changes).
We know, https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested, that the M1 can provide about 64GB/s from either SLC or DRAM, and that this number doesn't change whether one, four, or eight cores are hitting SLC/DRAM. 64GB/s is 20B/clock cycle. This probably means the NoC can "really" transport 32B/clock cycle (once you take into account various NoC overhead), but I would assume the NoC is running at some fraction (half? 2/3?) of the CPU clock cycle, so the actual NoC data width is at least 32B (at a 2/3 rate and extremely low overhead), or maybe 64 (at a half or 1/3 rate)?
It's interesting that Dick Sites, (2015) https://www.youtube.com/watch?v=QBu2Ae8-8LM, talks about the main thing he wants from a dedicated data warehouse CPU is the ability to move 16B/cycle at a duty rate of about 25%, and he spends the talk complaining about how far every vendor is from that. Apple essentially hit that (if you consider only the large cores...)!

Also interesting is think about this. Imagine (just as a thought experiment) a mesh NoC with 8 cores (and various other devices) on it, so let’s imagine say a 4x4 mesh.
In theory, then, a multi-sliced L3 should be able to provide ~4 requests per cycle, and these should be able to find their way via different (and hopefully non-overlapping) routes to different CPUs, giving you a substantially higher aggregate bandwidth. We don’t see this, or anything close to it. Why not?
Presumably a Core Complex appears to the NoC like a single unit, at least for most purposes. So perhaps you can get that level of higher total bandwidth from the SLC if you are running not just the performance cores at full speed, but also the GPU, the NPU, and the ISP?

Second it is interesting that I can’t do better by using both the Efficiency and the Performance Core Complexes. My assumption would be that they are treated as different destinations by the NoC, and if the NoC can provide higher mesh bandwidth (by simultaneous routing from different sources to different destinations along different mesh segments, as I suggested above for allowing aggregate SLC bandwidth to be higher once GPU, NPU, and ISP are also making requests) we would see it in that case.

Perhaps my assumption of available higher SLC bandwidth to different clients is incorrect?

Third within a Core Complex, the L2 is apparently split the same way, into four independently addressable units, each of which is capable of providing close to 32B/clock cycle, so we do see total L2 aggregate bandwidth increase linearly with the number of cores. My guess is the drop in bandwidth by ~25% has to do with writing the line into the L1D, something like
- for four cycles I can read 32B/cycle from L1D (16 B from each half of the L1D)
- during those same four cycles each of four quarters of a 128B line is transferred from L2
- during the fifth cycle I can't access the L1D because the fully-accumulated line is transferred from a line buffer into the cache.

cache replacement using LRU or FIFO on a line-by-line basis

We can see more cache design choices in (2009) https://patents.google.com/patent/US8392658B2. The idea here is in a cache (conceivably everything from L1 to SLC) the cache tags include, along with all the other status bits like MOESI, a bit specifying the replacement policy for this particular cache block. The explanation is clumsy, but the idea is that while LRU usually works well, there are access patterns (specifically streaming through data) for which a limited FIFO replacement works better. (Limited FIFO meaning you restrict the streaming data to just a few of the available ways in any set, in the most extreme case just one way, but more generally perhaps two or four ways in the hope of some short-term reuse.)
There are papers available suggesting how a cache might dynamically figure out which of these techniques (LRU vs limited FIFO) is currently better, but the patent is not that ambitious. Instead, the cache is informed, in each request, at to whether the request should be treated as LRU or as part of a stream. This information in turn is, either associated with pages (ie set up as part of the pages tables) or is held in range registers. These range registers appear to something of the equivalent of PPC BAT registers, situated in MMUs, and being consulted in parallel with the contents of the TLB. Presumably they are used for the obvious IO cases (like Display DRAM) but the patent also suggests that (some of them, somehow) are available to SW. We saw these same range registers in the earlier referenced 2009 TLB prefetching patent, where they were used to describe different memory classes being accessed by the GPU (texture, pixmap, etc) along with whether they would benefit from stream-like prefetching.
For 2009 the system is pretty impressive, and I expect the various cache subsystems have only improved since then.

cache telemetry

Consider the M1 and an L1 cache miss for a particular core. That miss may be serviced from a variety of other places, for example

- another L1 or the L2 of this cluster
- a cache (L1 or L2) of a second cluster
- the SLC or DRAM

If we track cache fills by these different sources, we can use that information in a few different ways. (Precisely which options are optimal in particuar conditions will depend on other details). For example
- if we see an above average number of fills from DRAM we might increase the frequency of DRAM and the SoC communications fabric.
- however if those are already at maximum, and we are still frequently waiting on DRAM, we should reduce the speed of the core since it is mostly burning energy waiting for memory

- if we see an above average number of fills from the other cluster, we should increase the frequency of that cluster.
- alternatively we should have the OS reconsider how threads are being allocated to clusters.
The baseline heuristics the OS uses for scheduling are static, things like
- allocate threads/processes by QoS
But statistics gathered during execution can augment this by telling us that certain threads (in the same process or different processes) are constantly communicating through memory, in which case those threads should be scheduled together on the same cluster.

These ideas, describing per-core counters that track these cache fill sources, are the content of (2019) https://patents.google.com/patent/US10942850B2 *Performance telemetry aided processing scheme*.

However there's a second, stranger, aspect to the patent! By now we've seen many Apple patents, and they tend to show a few common baseline designs, primarily an approximation of the the 1st gen up to A6 design, likewise for the second gen A7..A10 design, then the third gen A11..M1 design.
Not this patent! It describes a design with
- two clusters
- four cores in each
- a per-core L2 and
- a shared per-cluster L3 (which the patent calls LLC)
Now this could just be fancy of a different law firm writing up the patent. But it makes one wonder if we are seeing an aspect of the 4th gen design.

There are obvious advantages (sharing material across multiple cores) to a consolidated L2 if that larger design does not hurt cycle time. But there is also a (non-obvious) disadvantage, namely that all cores have to share a common frequency with the L2; otherwise you need to pay a few cycles transferring information across a frequency boundary, which is a noticeable additional delay given the 16 cycles or so to get to L2. As far as I can tell, the cores of Apple's gen 3 designs (so A11..M1) all operate at a common cluster frequency.
By switching to a large per-core L2, backed by a common cluster LLC, one gets most of the larger size

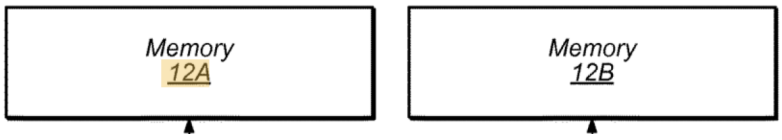
benefit for each core, while still allowing the cores to run at separate frequencies. This sort of flexibility in per-core would seem to be necessary to really take advantage of this cache telemetry.
(Of course, ultimately, there are many ways to slice and dice the precise details. For example one could pair cores together, sharing frequency and a common L2, and cluster three or four of these pairs together behind a common cluster SLC. One could even imagine something like a one core+L2, two core+L2 and 3 core+L2 all sharing a single LLC – such a scheme could allow one, two, or three closely tied thre
ads/processes to work closely together while being fairly flexible in handling other workloads.)

consolidated address translation unit

Given our experience with x86 (and similar designs that grew from a single simple CPU) we tend to treat the terms TLB and MMU as more or less synonymous.
However, as far as I can tell, Apple have likewise deconstructed this machinery. I can't be sure of the details, but approximately the TLB is something like a simple L1 cache, while the rest of the MMU (page walkers and suchlike) appears to be a unitary entity living up at the SLC/memory controller level (and shared across all agents).

There are many ways to slice the problem of page walking, but an issue one constantly has to remember is that Apple's concern is very much an entire SoC, not just CPUs, and this has constant ongoing implications. For example one likely has translation required for all IO elements (not least for security), which means something somewhere that's providing a page walker so as to populate a TLB.

Look at this diagram from (2011) https://patents.google.com/patent/US9652560B1 *Non-blocking memory management unit*.



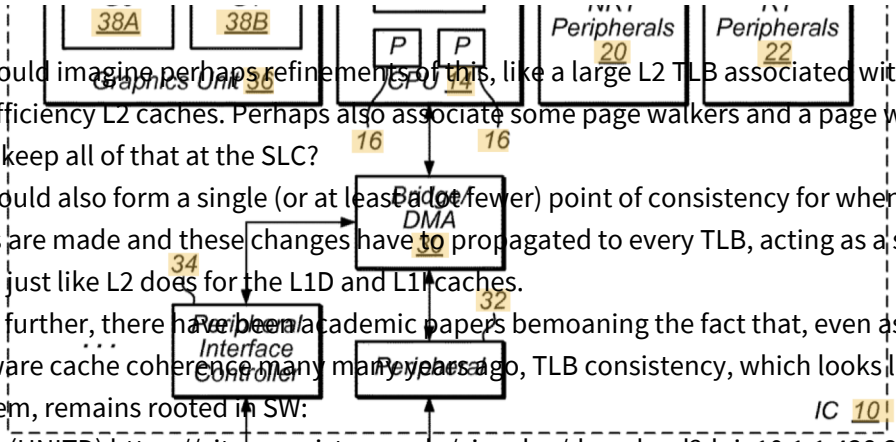


FIG. 1

One could imagine perhaps refinements of this, like a large L2 TLB associated with the Performance and Efficiency L2 caches. Perhaps also associate some page walkers and a page walker cache with the L2, or keep all of that at the SLC?
This could also form a single (or at least a lot fewer) point of consistency for when changes to page tables are made and these changes have to propagated to every TLB, acting as a snoop filter for the L1 TLB's, just like L2 does for the L1D and L1I caches.
Going further, there have been academic papers bemoaning the fact that, even as the industry adopted hardware cache coherence many many years ago, TLB consistency, which looks like much the same problem, remains rooted in SW:
(2010) (UNITD) https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.438.2661&rep=rep1&type=pdf *UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All*
(2011) (DiDi) https://www.doc.ic.ac.uk/~lvilanov/publications/files/pact11_didi.pdf DiDi: *Mitigating The Performance Impact of TLB*
Shootdowns Using A Shared TLB Directory

(BTW, though more GPU than CPU, the 2011 patent is interesting in itself. As far as I can tell, at that time [remember, PowerVR GPUs...] the GPU internally operated purely in virtual address space [with the implication that GPU caches are flushed on context switch to a different GPU user address space].
So it's only when requests leave the GPU that they need to be translated. These requests flow through the MMU as in the diagram above, on their way to servicing the request from the SLC or DRAM.
One consequence of this is that this "GPU-external" MMU can service page faults even though, nominally, the GPU has no support for virtual memory. What's required is that the MMU detect that the translation request matches a non-present page, and route a request to the CPU to fault in the page from storage. Once that is done, the GPU memory request [now translated, and the data serviced from the faulted in page] can be sent back to the GPU.

This is nominally different from a CPU, where a page fault is accompanied by a context switch, to give the time to some other process while storage services the fault. But it works because a GPU, of course, has thousands of active threads, all ready to step in as soon as a cache miss occurs; and to the GPU this page fault just looks like a cache miss that too longer than usual.

Why page faults anyway? I thought iOS had no VM?!
Not correct! iOS doesn't [usually] have page-outs and so what looks to the app like unlimited memory. But it does support other aspects of virtual memory including memory mapped files, and it is not uncommon to use memory mapping for things like large texture atlases.)

With all this speculation in mind as to the economy and efficiency of having a single locus of truth for all TLB tables and all page walkers associated with the SLC, (2019) https://patents.google.com/patent/US20210064539A1 *Unified address translation* is very interesting.

The patent itself is essentially about APRR/SPRR, a sort-of security/sort -of performance feature described in

partial detail here: https://blog.svenpeter.dev/posts/m1_sprr_gxf/
The basic idea is that for tighter security one wants to be able to flip the permissions of some pages fairly frequently. For example for JIT pages, one wants to be able to flip the page between Write mode (no Excute) and Execute mode (no write) fairly frequently. But traditionally modifying page permissions is an expensive process. The page table entry in RAM has to be changed, then a broadcast sent to every TLB on the system (and remember that include the GPUs, NPUs, and various other IO devices) telling each to flush the relevant page from its TLB, followed by a wait till every TLB responds that the flush is done.
The Apple alternative is that a page no longer has a set of permission bits, rather it has a "permission index", which moves with the page translation into the TLB. This index is split into two parts. Under "A" conditions, the first part of the index is used to index a (short, 4 entry) table giving the appropriate permissions; while under "B conditions" the second part of the index is used. The usage model, I assume, is something like
- normally the A conditions are valid and represent safe page usage
- if a temporary change is required (eg to write to a JIT page) the B conditions are established (by writing to a register or whatever)
- as soon as the change is done, conditions flip back to A
I *think* the idea is that the B conditions are only set up for the specific CPU that is making the (temporary) change to the page, so no other TLB needs to be informed of this change, avoiding all the cost of (two!) standard TLB teardowns for what will be a temporary modification.

Honestly, security is of no interest to me. But if its your thing, that 2019 patent builds on (2016) https://patents.google.com/patent/US9852084B1 *Access permissions modification*, which describes an early set of augmentations to the MMU to provide various security improvements. Among other things, these augmentations include
- a BAT register describing the range of genuine OS code, so that code outside that range cannot run with OS-level permissions
- a lock register for locking the above BAT, the page security remap tables and various other things after they have been constructed so that no attacker can modify them after a very short boot window
- ways to limit the abilities of OS and hypervisor code when they are accessing user pages.

More interesting however, IMHO, is the material that is hinted at in the 2019 patent. The existing state of the art describes CPUs as having a TLB and an ATU; the new design talks of the ATU being optional in the CPU and moved up to the processor complex.
What's the difference? The point is that this is what I described above: a CPU has a minimal L1 TLB, but no ATU (Address Translation Unit, ie all the extra machinery to perform table walks). If the L1 TLB misses, the request goes up to the Processor Complex (ie L2 cache) which will have a large L2 TLB, shared across the cores that share the L2 cache, the page walkers, and so on.
So it seems like versions of this idea in one form or another have persisted from 2011 till now.

A different aspect of the TLB/page walker is optimal performance. (2011) https://patents.google.com/patent/US9009445B2 *Memory management unit speculative hardware table walk scheme* is surely obsolete in many details, but the essential idea remains interesting. The patent describes a limited machine (think PA Semi, even before Swift) with the following interesting features
- there are the usual separate I and D TLB's, D-TLB tightly associated with the LSU; but there is a consolidated MMU that provides a second level TLB and a consolidated page walker. In other words even at

this early stage we separate the high performance cache aspects of a TLB from the other “overhead” aspects of operating page walkers.
- the page walker provides a queue for TLB requests.
- the precise details don’t make much sense to me (and may be an attempt to lower the energy/transistor budget) but the end result is that the page walker prioritizes all “definite” page walking (which could be on behalf of either I or D cache) before any “speculative” page walking. The target CPU appears to have been so restricted that speculative means literally what it says, load/stores were segregated by speculative (not yet latest in ROB) vs non-speculative!

One could imagine a future system with multiple page walkers, and an obvious prioritization scheme with I misses as highest priority, then D misses, then I prefetch misses, then D prefetch misses as lowest priority. (Apple, unlike most microarchitectures, runs their I and D prefetchers in virtual space, so they can, and do, cross page boundaries, generating TLB requests along the way.)
My primary takeaway from the patent is that the more you have page walking centralized to a single locus of control, the more you can engage in this sort of prioritization of different classes of page lookup.

Memory controller

This (2010) https://patents.google.com/patent/US8510521B2 looks very technical, but there’s some interesting stuff that’s described though not part of the main patent.
The system (ie early iPhones) has five independent ports into the memory controller, two for the two “GPUs” (I would expect by now these are consolidated into a single port behind an L2 cache), one for the CPU system (two CPUs behind an L2 cache, with some peripherals connected to this subsystem), an NRT port (non-real-time, primarily for media encode and decode) and an RT port (real-time, primarily for display controller stuff).
The requests through each of these ports are tagged with various levels of QoS (at the time, three levels of QoS for RT, two levels for NRT, and everyone else gets whatever is left) along with a flowID. The requests are routed to one of two memory channel controllers (interesting in that it looks from the outside like the iPhone has a single memory controller handling a single 64-bit wide channel. Is that an illusion, with really two independent 32-bit channels, or a cleanup from these older designs which were based much more on third party IP?)
The focus of the patent is optimizing for bandwidth which still servicing the QoS appropriately, and the way this is done is rather clever (at least to me, as someone unfamiliar with this field!) There is a first round of sorting of requests based on the requesting port and the QoS level.
These first round requests go into a queue in each channel controller which then sorts them according to traditional memory controller criteria. This is described in some detail, but essentially as you would expect: reads and writes are segregated, and then those which will hit in the same DRAM are page aggregated into what are called affinity groups. These second level queues are scheduled according to QoS+affinity criteria. These are complicated, but essentially they expand on the idea of “schedule the affinity group with highest QoS, but include all the items of the affinity group not just the highest QoS

item".
The requests scheduled from these second level queues go into third level queues which are then sent to the DRAM according to DRAM criteria (ie meeting timing requirements and suchlike). If you're unfamiliar with memory controllers, it's worth working through the whole thing to see exactly how the details I'm glossing over are handled.
(Apple seem to *really* like this patent. I found four version of the patent that all look identical! I've seen a few others where there are two copies of the patent [essentially same words, just the title is changed to emphasize feature A rather than feature B] but this is the only one where I've seen four different features they liked so much they wanted each to get its own title!)

Another way the flowID is used is for bandwidth allocation. The above tries to optimize total bandwidth, but within that total bandwidth we may not want one greedy client using up everything. In principle that sounds fine; in practice the problem is always how to implement the idea without too much overhead.
(2018) https://patents.google.com/patent/US10437758B1 gives one part of the solution. Many details are omitted (but can obviously be imagined); the idea that matters is that every "request stream" gets a pool of credits, you use up one credit when you make a read request, and the credit gets returned when the read request delivers the loaded data from DRAM. A stream can be fairly flexibly defined using some combination of the agentID, the flowID, and the QoS.
The mechanics of this are fairly clear for reads, and the net result is that a quota can be enforced in a distributed fashion at the source of read requests, rather than in a centralized fashion in the memory controller. For writes it's less clear what to do because there is no natural object that is returned when a write completes. The patent is somewhat vague on this. The idea seems to be that
- there is a natural reply to the write *request*. Maybe every NoC packet automatically gets a reply validating that the packet was delivered?
- this reply gives some sort of information about the number of write credits left. (Which helps, but not that much, because how do you know when a write has happened and so you have an additional write credit?)
It's possible that the write issue is just not very important if the memory controller is using a *virtual write queue*. This is another academic idea that we know has been implemented in IBM POWER for a few years now: rather than having the write queue being of limited size in the memory controller, allow it to be of more or less indefinitely large size , with the overflow stored in the last level cache. Conceptually we integrate the memory controller with the LLC, so that all stores are placed in the LLC, at which point they will then hang around indefinitely; but when the memory controller switches to write more it will aggressively try to drain as many dirty lines in the LLC as possible: https://www.cs.cmu.edu/afs/cs/academic/class/15740-s17/www/papers/skdhj10.pdf. But I haven't yet found definitive evidence that Apple is thinking along those lines.

Obviously one of the main jobs of the memory controller is, on a cache miss, to move the data from RAM to the requesting cache/core as fast as possible. Consider what that entails.
A cache line is 128B. Let's say the width of the NoC (Network on Chip) that's connecting different

blocks, like the Memory Controller and a Processor Complex, is 32B wide. That means it will take 4 beats (ie transfer operations) to move a line, and those beats run at the NoC speed which is probably half or so of the CPU speed! So just getting the entire line from here to there can take a non-negligible number of cycles.
An obvious solution (ie this was already being done back in the late 90s) is Critical Word Forwarding which means exactly what it sounds like -- the Memory Controller ensures that the exact "word" (whatever that means in terms of NoC/bus width) requested is transferred first, then the rest of them, and the L1D is set up to forward that Critical Word to the LSU before assembling the entire cache line.
That's great, but there's a secondary issue, as our CPU's get more complex, of the scheduling on the other side -- how does the LSU know when to have that particular load lined up and ready to execute to grab the word it wants from the L1D? We've already mentioned the concept of Replay, and how dumb replay might retry a failed load every N instructions, while Apple's Replay adds a fake dependency to the dependency vector of the load instruction, with that dependency only satisfied when the L1D has the data available.
Again that's great, but it means we have a cycle or two delay between when the L1D receives the data and when the load executes. So by 2010 Apple have https://patents.google.com/patent/US8713277B2 which has two pieces to it.
(a) The memory controller signals to the appropriate L1D that, in the next cycle, it will be dropping the critical word on the NoC.
In an ideal world this would achieve two things:
- all the machinery that's sitting between the memory controller and the L1D will prepare itself (fully power up bus interface units, buffers, and suchlike) so that we don't encounter any one or two cycle delays for powering up
- the L1D knows (because it knows the speed of the NoC, etc) when the data should arrive, so it can signal the Scheduler as far in advance as necessary to ensure that the load is ready in the same cycle that the Critical Word data is ready.
(b) But sadly the world is imperfect. The above sounds great, but in reality even though the Memory Controller planned to drop the Critical Word on the NoC the next cycle, the NoC is a shared resource, another client might have grabbed it, there might be routing congestion along the way, there can be a frequency mismatch between the CPU/L1D frequency and the NoC frequency necessitating a cycle or two clocking delay at the transition between the two, etc etc.
So the second part of the patent is we have a little agent sitting between the Memory Controller and the L1D who tracks the discrepancy between when the data was promised and when it actually arrived, and keeps adjusting the delivery time passed on to the L1D to match a best guess based on recent delays.

That all sounds pretty good. Can we do even better? Well, if one load was waiting on DRAM, what if there's a second load also waiting on DRAM (either on the same CPU, or elsewhere in the system)? Doesn't that second load also want to get its critical word ASAP? Of course it does, and so we get the followup patent a few months later (2010) https://patents.google.com/patent/US20120137078A1, where the memory controller does as before, for the first critical word, puts the rest of the line aside,

sends out the critical word for a second request, repeats as necessary, then eventually gets round to providing all those clients with the rest of their cache lines.
Even this is not the end of the line. Once you start sending the remainder beats for a line, are you locked into that transaction until the line is done? Or can you interrupt it if a new critical word becomes available. The patent covers this second possibility.
Even beyond this, once you start sending the rest of the beats, what order should you use? If address overhead is not a factor (ie every beat has to contain an address, there is no implied address incrementing of subsequent beats) then it seems to me you can make two improvements:
- the order of the beats. Ideally the beats will be sent in the order required by the CPU. You don't know this, but a reasonable heuristic, it seems to me, might be something like "if the request was in the first 3/4 of the line, give the rest of the line in incrementing address order, then wrap around; otherwise in decrementing address order then wrap-around"
- interleave these non-critical beats across all requesters, rather than sending the entire line of a given requester.
Obviously both of these need to be simulated, but intuitively they both seem like they could improve overall performance slightly at low extra complexity.

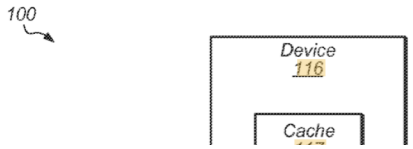
There are other things a memory controller can do. A device frequently has two or more memory "banks", by which we mean simply standalone DRAM storage devices that can be independently powered down. Under some circumstances ("performance mode") one wants all banks to be active, both
- to store as much as possible (less use of compressed RAM, more cached file blocks), and
- to stripe successive memory ranges over the different banks so that, insofar as possible, all banks contribute to providing immediately useful storage, thus lowering latency (more active DRAM pages) and increasing DRAM bandwidth.
But there are alternative circumstances ("efficiency mode") where one might use lower energy (at the cost of slightly lower performance) by using fewer memory banks and allowing those not in use to power down.
Apple have a patent for doing precisely this, in the context of Intel Macs. Once again I've no idea if this was ever used. It's basically support for hot-plugged DRAM; at the time of a switch down to performance mode, some data is flushed to disk, some is copied from active ranges in the victim bank to available ranges in the non-victim banks, and the memory controller is reprogrammed to map physical addresses from the old to the new mapping of address->(rank, bank, line). This seems bizarre and something I've never heard of hot-plugging DRAM apart from data centers, but the patent is here:
(2010) https://patents.google.com/patent/US8799553B2 *Memory controller mapping on-the-fly*.
M1 comes with two DRAM chips so certainly, in principle, could engage in this sort of thing. Does it?

System Cache (SLC)

(2012) https://patents.google.com/patent/US9218040B2 discusses the SLC. It's frustrating in the way so many of these patents are, in that it throws out phrases that suggest something important, without

ever explaining them. Some examples:
- on multiple occasions Apple patents refer to the L1D as part of the LSQ. One can of course see what they mean by this, but is there anything significant to this phrasing implying a difference of emphasis from the traditional model, where one imagines the LSQ as part of the CPU, and the L1D as something slightly separate from the CPU?

- in the above patent, Apple refers to (and diagrams) the SLC as part of the memory controller; again one is left to wonder the significance of this phrasing.
However in multiple places Apple seems to suggest that the System Cache should be thought of more as a front-end to DRAM than as a back-end to the L2 (ie more like what is often called an L4, than as an L3). In some patents they even refer to it as a Memory Cache rather than a System Level Cache.
This difference is not mere semantics; it has implications for exactly how coherence is handled, and it allows the memory controller to be tightly integrated with the SLC (as in using it for a Virtual Write Queue, as I've already suggested).



- likewise in the above patent Apple draws the SLC as, in fact, composed of two caches that behave like one cache. Once again, one is left to ponder the significance of this. I *assume* it boils down to something like the even-odd split we saw in the L1D -- each half cache can simultaneously service a request from a different agent?

- the patent, and the diagram, suggest that iPhone SoC designs use two memory controllers, and two memory channels. The DRAM bandwidths sustained by iPhones mean these channels must each be 32-bits wide. I raise this point because the concept of the DRAM channel is one of those computing words that has become useless for most communication purposes :-(
In the PC world, people seem to think a channel means a 64-bit wide connection to DRAM. In the ARM Android world they seem to think it means a 16-bit wide connection. Apple seems to think it means a 32-bit wide connection. Sigh.
Regardless the idea seems to be, once again, splitting by address (the obvious choice is selected via address bit 7, so that even 128B lines and odd 128B lines go to alternate SLC caches and then alternate controllers, but who knows? there may be reasons to split by bit 8, by 256B units, or even something

large like by 16kiB page units? or perhaps the fundamental size is determined by the DRAM page size, so that each controller and thus its SLC works with even vs odd DRAM pages?) Each memory controller then maintains its own queues of operations targeting DRAM.
Compare this with the iPad design which doubles this, so we have four SLC's and four memory controllers.

As mentioned above, memory requests are tagged with a flowID (the patent suggests there are 16 of these, 4 bits, though of course this may have risen in more modern SOCs), and the SLC makes use of these. Specifically the SLC has a concept of *sticky* lines and (I probably have the details wrong, but the basic idea appears to be)
- some SLC lines are non-sticky, and these are the ones that are overwritten first when space runs low
- each flowID has a quota for how many lines it can store in the SLC. Its sticky lines then fight against each other for occupancy, once there are no non-sticky lines to pick on, but a given flowID can't hurt the lines of a different flowID
- it's unclear to me if or how the CPU can get in on this flowID mechanism. The GPU is in on it, and in fact part of how it works is that all data associated with a particular frame has the same flowID. The next frame, two things can happen:
+ the GPU can tell the SLC to relabel all flowID X lines as now belonging to flowID Y
+ alternatively, if a request is made (by flowID Y) for a line that hits in cache as belonging to flowID X, the ownership will change to flowID Y
It's unclear why both these mechanisms exist, and it may be that Apple wasn't sure quite how this would all play out, so provided both for the OS/app teams to experiment with?

But the generic thinking behind the cache appears to be that it should be though of primarily as a means to facilitate communication at lower energy. Any latency savings are nice (and the system will happily use the SLC as an L3 cache in the absence of anything else going on) but it's designed, and provided with this extra machinery, to expedite communication across space and time. In particular the flowIDs and sticky bits are an attempt to ensure that data required for a purpose, then not required for a while, but which will be required later, remains in the cache rather than being aged out. Think eg graphics textures required for the construction of a frame every 60th of a second. You can think of it as somewhat like a manually managed cache (where you lock lines in the cache so they can never we replaced), but more flexible, delegating more of the detail work to the cache rather than having the developer worry about every little detail.

The actual focus of the patent is the same sort of thing we saw earlier with the L2
- track how large a fraction of the SLC is "really" being used
- limit the number of active ways to match that fraction
But the details differ. In particular the power-down granularity is by-way, not by one third of the cache; likewise different are the details tracking when to grow or shrink the cache. One final point is that, when it comes time to replace a line, of the active ways, and of those that are eligible (taking into account stickiness and flowID) a random line is chosen. Given the constraints, this was probably the

easiest choice at the time, though today who knows how many of these design decisions have been tweaked.

As with any cache, a question of interest for the SLC is how it allocates lines.
The Apple pattern appears to be that, to first order, CPU computation and caching happen more or less as expected in L1s and the L2, with lines allocated in L1 and L2 as victim. (And with unclear details as to how data moves from the P- to the E-cluster and back). In this model, the SLC is mainly there to service the rest of the SoC, not the CPU. But if the rest of the SoC is doing nothing, clearly you want to get some value out of that large block of SRAM, using it as an L3. But how exactly? As a victim cache for the L2? Unclear.

(2012) https://patents.google.com/patent/US20140089600A1 appears to answer the question, but I suspect it's an answer only relevant to non-CPU agents.
Consider any cache that allocates lines on a miss, when a new request comes in. What do you do with the pending request while you wait for the data?
Well the awful decision is to block until you have the data, but clearly that's no longer acceptable.
Next option is to put the request in some sort of queue while you wait for the data. But this has the consequence that for every subsequent miss, you have to check against that queue (because if you've already sent the request to DRAM, you don't want to send it again). That means checking a large associative structure, with power issues.
One could imagine different solutions to this, but the solution Apple chose (at least, as of 2012 -- with all these old patents there's the likelihood that new designs and new processes make a different architecture more appropriate) was to allocate the line without the data. Rather than just being marked invalid, the line was marked as "pending". Then any subsequent request for this line would also see it as pending. All requests to pending lines go into a replay buffer where they wait till their line arrives. (Presumably they are awoken by some sort of selective wakeup, same way we have seen this done for instruction Replay. An obvious scheme, for example, would be to hash the desired address down to an appropriate length -- 8 bits? -- and wake up all requests matching that hashed address when the line comes in.)

Another unexpected aspect of the SLC is that it can act as a "puppet" controller for other caches.
Consider (2013) https://patents.google.com/patent/US20150143044A1 *Mechanism for sharing private caches in a soc*. The idea is that
- multiple IP blocks on the cache have caches
- sometimes those IP blocks are using none or only a fraction of their cache (eg the block is powered down)
- in which case, why not reuse that cache for some other client?
The SLC controller is tasked with doing this and making it work.
Your immediate reaction is probably to think of this as something like the ISP or Media Engine is being unused, so maybe the CPU gets to use some of their cache as an augmentation to the L3 of the SLC. But Apple actually give as example a rather different case, somewhat backwards from that, suggesting that

the GPU might want access to more cache, and the CPU may be using only some part of the L2 and can let the SLC/GPU use the rest.
Presumably the primary win in this is energy (it's always more expensive to go off-chip) but there's also some latency win, maybe 50 ns or so (?) to get the data via a trip all the way to the SLC then rerouted to a private cache, rather than 100ns to DRAM.

You might wonder how this is actually done. At least part of the answer (which has been alluded to in some other sections above) is that the SLC maintains two sets of tags, one set being the traditional cache tags for the SLC, the second set being tags for all the other "relevant" caches in the system. These two are tested in parallel.
In some sense the point of the second set is coherency, to allow the SLC to act as a single coherence point rather than requiring every transaction to visit every cache, each one testing whether the transaction matches what's in that particular cache. But as Apple have realized, if you are performing this sort of test, you can add a little more data to those "remote" tags to both redirect a request to the appropriate cache, and even to puppet the cache and so populate it with data as you see fit.
This set of parallel tags and how they are used is described in (2018) https://patents.google.com/patent/US20200081838A1 *Parallel coherence and memory cache processing pipelines*. This looks scary to read, until realize that the "parallel coherence pipeline" simply means this second set of tags that is probed in parallel with the normal SLC/ memory cache tags.
(Other systems may do this by forcing the outermost cache to be inclusive of all inner caches, which has the same effect, providing a single location to test coherency. But it's clearly more efficient to allow the caches not to be inclusive, and simply have the outermost cache replicate the tags of all inner caches, not the tags *and* the data.)

large vs small core issues (some A10 history)

There's not much particular to say about Apple's use of large vs small cores. However there is some interesting historical trivia.

You will recall that the A10 had the interesting design that the large vs small CPU was essentially invisible to the OS, meaning that a single pseudo-CPU presented itself to the OS, but toggled between the large or small core depending on circumstances. This sort of pairing (large tied to small, rather than a cluster of large's tied together and a separate cluster of small's tied together) has been considered in the literature in other contexts (for example to save power by having the CPU switch to the small core when executing low-IPC code [either lots of unpredictable branches, or lots of misses to DRAM), but it's not clear that there's any point in doing this when OS-transparency is not essential.

One suspects Apple did this as a one-time experiment but with an understanding that clusters were the long-term goal. Given this, they seem to have considered the A10 small core as something they could experiment with. What sort of experiments? Well look at (2014) https://patents.google.com/paten-

t/US20160147290A1 *Processor Including Multiple Dissimilar Processor Cores that Implement Different Portions of Instruction Set Architecture*. The idea is simple enough – put a limited ISA on the small core, and swap to the larger core when it's necessary to implement the ISA that's not on the small core. (Seems simple, but you'll note that Intel seems incapable of doing something like this for Lakefield.)
The complication is that the swapping has to be done by the cores themselves because, remember, the OS only see's one core!
But that's not the interesting part. The interesting part is what's the part of the ISA you want to omit? We know, for example, that Firestorm and Icestorm apparently both implement the full Apple version of AArch64, even up to both implementing AMX. So what's omitted in the A10? The very cool answer is 32-bit support! That seems to have been part of the additional value Apple extracted from this one time-experiment, a core that ran only 64-bit on which they could presumably test OS modifications, see how much design could be simplified by omitting 32-bit complexities, etc etc. One wonders the extent to which the A10 Zephyr core (small core) was in fact a prototype for the micro-architecture of the A11 and subsequent family, rather than being a simplified version of the A10...

There second patent in this space is (2015) https://patents.google.com/patent/US20170068575A1 *Hardware Migration between Dissimilar Cores*. This tells us essentially two things:
- the dual-CPU presents itself to the OS via its power states. The OS decides (based on its reasons) upon a power state for the pseudo-core, and the cores decide which one should be active based on this power state. This means they have to transparently move state between them; obviously the user visible registers, but also a few SPRs. The bulk of the patent describes how this is done, but it's not especially enlightening or generalizable.

DMA

As we move further away from the CPU, we're covering territory about which I know less and less. Even so, there are still interesting things one can learn. For example consider
(2005) https://patents.google.com/patent/US7620746B2 and (2007) https://patents.google.com/patent/US20080222317A1, followed by (2008) https://patents.google.com/patent/US20090248910A1. The first two are from PA Semi, the third is from inside the iPhone team but covers essentially the same material.
The 2005 patent covers the idea of adding functionality to a DMA controller, so that the DMA can perform some function on the data as it streams through. The ideas suggested for this include data transformations like crypto, or reductions, like a CRC or a hash.
The 2007 patent builds on that by allowing the chaining, ordering. and dependency, of DMA descriptors so that the previously mentioned functionality can be stacked. The example given is something like a network stack where the TCP layer creates a checksum, the IP/Sec layer encrypts, and calculates a hash, and the ethernet layer calculates a CRC.
The point, of course, is that even though one doesn't think of it there has been a TCP-offload engine in your iPhone from the early days.

It's not clear to me if this mechanism is an ideal way to perform transformations that grow or shrink the data (ie compression/decompression, or codecs) as opposed to a more traditional accelerator scheme (ie pass an in buffer pointer and an out buffer pointer to the accelerator, and wait for it to give you an interrupt once it is done).

The obvious next stage, after these bulk transformations, is the appending/prepending and removal of networking headers and the other paraphernalia of a full TCP offload engine, and we get that in (2008) https://patents.google.com/patent/US8359411B2. An interesting point that the patent takes pains to point out is by placing this machinery in the DMA engine, not the networking hardware (eg in an ethernet chip) it becomes available to all network interfaces that use TCP. This is obviously nice insofar as it shares the HW across WiFi, cellular, even a Lighting/USB ethernet connector, but is especially cute when you think of something like Thread (Bluetooth radio, but using IPv6) which refits TCP/IP to a protocol that never before used it and is unlikely to have this offload on any chipset!

An especially interesting variant on this idea of "active" DMA occurs in (2008) https://patents.google.com/patent/US8610830B2 which essentially has the DMA engine between a local video decode buffer and display RAM reorder the data during the transfer so as to handle the four different (portrait vs landscape, top vs bottom) possible orientations of the display, though one suspects every aspect of this particular design, up to and including the swizzling DMA transfer, is now obsolete.

We discussed earlier DMA that can route data directly into the cache. We've discussed above DMA that provides for faster networking. Why not glue these two ideas together? 🤣
(2006) https://patents.google.com/patent/US7836220B2 discusses networked DMA. There is a standard for this already called RDMA, but what Apple proposes is lighter-weight. (RDMA builds on TCP/IP, so it can be routed across networks. But if you don't need that degree of routing, you just want to work on the same ethernet you can strip out the IP and TCP routing stuff.
The idea seems to be that machines A and B will each create a range of memory visible to the other machine, and essentially be able to DMA from one cache to the other and back via smart DMA with no CPU involvement! Is this of any value? Did they ever do anything with it (or plan to)? Was it part of PA Semi's business plan that was abandoned after the acquisition? Who knows.
Maybe we'll never see it in a consumer product, but it will be part of whatever they plan for their data center hardware (and of course you know they are moving to their own data center hardware!)
A constant theme in computing has been the joy of making an X look like a Y. You see this in every virtualized device driver, in virtual memory, in RPC's, in the very concept of virtualization. Engineers seem to delight in the idea that if I hook up things just right, I can make this machine think it's doing X even though I'm reinterpreting everything it does.
Some of these fakes have had long and successful lives (like virtual memory). Some have crashed and burned (if we're being honest) like transparent RPC, or the general idea of proxy objects and all that 90s DCOM/CORBA stuff. One of fakes that crashed even harder than remote objects was transparent networked virtual memory, and one could argue that networked DMA is much like that, but I'm not sure that's the correct analogy. I think the failures tend to be the result not of the functionality per se, but of

trying to make the aspects of the implementation invisible. People (of course) really like networking; but the gap between real network issues (latency, lost packet, random connect and disconnect) and hiding a network connection behind a procedure call (or, heck, a memory allocation) was a bridge too far, the network failure reality intruded too often to upset the abstraction layer. But if this networked DMA is

NoC issues

I'm no expert on NoC! This is my rough summary of what I think is the significant part of some patents. (Unsure if I will keep this section.)

A constant concern is transferring data across clock domains. We have
(2005) https://patents.google.com/patent/US7500044B2
(2007) https://patents.google.com/patent/US20080198671A1
(2015) https://patents.google.com/patent/US20160328182A1
which all seem to be fancier ways to performing this task with lower latency.

A second concern is arbitration. The big idea here seems to be unify arbitration with routing. As I understand it,
(a) you have something like a "local" connection from every agent to, let's call it, the central fabric, with easy, fast, lightweight arbitration over this local connection.
(b) this local connection feeds into a *queue* in the fabric, not directly onto the fabric.
The end result of this is that every agent can (mostly without drama or contention) throw a succession of requests at the fabric and have them immediately queued in a per-agent queue. Then a combined operation of arbitration and routing will extract requests from these queues and send it onto the fabric. This is as opposed to a bus-like scheme where you first ask for access to the fabric, then, once it is granted, throw out your request.
Or, to put it differently perhaps,
- the per-agent queues are placed in a central location rather than at the agent, meaning that scheduling can be done optimally across all queues; and
- there is no need to request access to the shared fabric (wasting latency waiting for the reply, and fabric bandwidths), that's implicit in the decision as to which queue gets the next scheduling slot.
(2005) https://patents.google.com/patent/US20070038791A1

There are some patents about enforcing (or relaxing) ordering between different buses, eg (2005) https://patents.google.com/patent/US7412555B2 for communicating with PCIe, and (2012) https://-patents.google.com/patent/US9229896B2 for communicating with AMBA. No idea if these are interesting or, if so, why.

This one (2007) https://patents.google.com/patent/US8284792B2 is about the design of a PCIe con-

troller. To me (knowing nothing about the subject) it looked pretty neat; same idea as we have seen with the cache designs where certain hardware is shared, other hardware is replicated, to get you two unit's worth of hardware with one unit's worth of cost. Presumably this was inherited from PA Semi and slept for a few years, until the iPhone 6 (2015) shipped with a PCIe SSD.

I've mentioned before that, right from, the start it seems like all bus/fabric/NoC transactions were tagged with QoS, which was then used in various ways. In the early days there was also a concern with realtime vs non-realtime IO traffic, and with coherent vs non-coherent devices. I'm not sure if any of this is relevant anymore. We have things like (2010) https://patents.google.com/patent/US8683135B2 which seems to be about allowing prefetch even from a non-coherent data source, and (2011) https://-patents.google.com/patent/US9176913B2 which, for reason I do not understand, seems to think it important to equate non-coherent traffic with non-realtime traffic and treat these two in the same way at the memory controller.

Another thing Apple do (which seems like overkill! but presumably this is born of experience) is that all requests sent over the NoC have a QoS attached to them. That seems not *that* surprising in the case of basic memory requests, and we discuss below, in memory controller, how this QoS is used to order requests in the memory controller. More surprising is that this QoS is also used to prioritize snoop requests! Perhaps in future the mechanism may become more aggressive, but as of 2018 https://-patents.google.com/patent/US10795818B1 the mechanism is fairly simple, mainly that each snooping machinery has a queue of incoming snoop requests, and once a priority snoop enters the queue, only a limited number of non-priority snoops are allowed to be serviced before the priority snoop is serviced.

The end point of all these ideas is (2020) https://patents.google.com/patent/US10972408B1, where adding to the central arbiter controlling the fabric, and all the per-agent queues, we now have a whole lot of per agent counters counting different things, and a programmable matrix that can choose from cycle to cycle which queue has highest priority based on things like age, recent bandwidth, read vs write, etc. The object, of course, is to achieve simultaneously (as far as possible) both never a wasted cycle on the fabric and latencies as low as is required across a wide variety of agents.

Everything we have described so far is built around a QoS identifier, and superficially this seems like enough. Fire off a transaction that's marked as PRIORITY (in fact Apple seems to use three QoS levels that are frequently referred as Red [realtime], Yellow [non-realtime or if you prefer, soft realtime], and Green [best effort bulk transport]) and it gets sent to its destination as fast as possible. What more do you want?
Well the reality is that packets often have to cross multiple buses and boundaries to get from here to there. For example a packet may originate on a PCIe bus, be moved onto the NoC, then moved off the NoC to a USB bus. At every transition between buses (and possibly at some internal locations of each of these buses) the packet may be placed in a queue. Even though the packet is sitting in the PRIORITy queue, and will be sent to the next stage as soon as possible, it may be delayed while prior transaction(s) complete.

Now the packet is on the next bus, it needs to be routed, and will be treated as PRIORITY again at the queue, but again there may be a delay.
The point, you should note, is that if all you have is a PRIORITY flag, that does not record how much an individual packet was delayed at intermediate stages. In an ideal world, you might want to move a long-delayed packet ahead of all the other PRIORITY packets for all subsequent steps of a multi-step journey; but doing this requires additional information attached to the packet. What you want, in fact, is something like a timestamp. But that's difficult to implement in a distributed fashion, across multiple devices and buses all running at different frequencies. Easier to implement as a practical matter is a concept of "urgency" which is filled in (and utilized) at the buffers and scheduling between buses, rather than being established at the end points. This urgency concept (essentially "this packed was delayed in a buffer for an unreasonably long time, so make up for it in subsequent routing/scheduling/arbitration decisions") is the subject of (2018) https://patents.google.com/patent/US20200057737A1.

Power issues

I'm no expert on power! This is my rough summary of what I think is the significant part of some patents. (Unsure if I will keep this section.)

Along with all the other stuff above, we have a few power-specific patents.
The on-going idea seems to be to centralize ever more power control in one place. This may seem obvious, and some of the patents may seem ridiculously dumb; I think they have to be placed in a context where a phone (even an iPhone) didn't consist of a single SoC with (almost) everything on it designed by Apple, but consisted of multiple different chips from different vendors that had to be tied together as best possible.

We start with (2007) https://patents.google.com/patent/US8645740B2 and https://patents.google.com/patent/US7711864B2, where we have two big ideas
- central management
- management based on measuring the actual power draw of each component rather than some sort of spec or theoretical model.
But this central management is all done by SW on the CPU.
There's also (2007) https://patents.google.com/patent/US20080168285A1 which is surely obsolete, and has to do with ensuring that a CPU or GPU has finished executing all the current instructions before it's powered down.

By (2009) https://patents.google.com/patent/US7853817B2 we've evolved this to
(a) perform (some) power management using a dedicated power management controller. (As opposed to having the CPU power up and down various pieces of hardware, which means paying the energy cost of constantly waking the CPU up).

(b) make this power management controller aware of various dumb ways in which attached hardware needs to be put to sleep/woken up, because so much hardware hasn't yet transitioned to the modern unified way of handling sleep/wake.

This is followed by (2010) https://patents.google.com/patent/US8271812B2 where this power manager now sees the world as a set of domains (like the CPU domain, the video decoder domain, the audio playback domain), each domain has associated with it a set of performance states (which at the high end of degrees of speed, and at the low end are degrees of sleeping, with more sleep meaning a slower wakeup). Each domain, and each performance state, have associated info describing the control registers and their values to force that state.
Obviously this gives more centralized control, and is more flexible to set up and going forward, but a particular concern Apple has is to ramp up various separate functionalities in lock step. So that, eg, if we slow down the CPU we also slow down the L2 cache and maybe the bus interface -- we don't want a situation where any subsystem is running faster (or slower) than is appropriate for the rest of the subsystem.
Also, slightly later in 2010, we have https://patents.google.com/patent/US8806232B2. Now we are giving the power manager a little more intelligence (not CPU-level intelligence, think more finite state machine-level intelligence) along with some non-volatile storage. The power manager can now save (and then restore) the state of some devices using that non-volatile storage, and can engage in more sophisticated power control without having to fall back on the CPU.

Next we get (2011) https://patents.google.com/patent/US20120185703A1. The patent is not at all clear(!) but I think the new feature here is that performance control has moved beyond on/sleeping/powered down to DVFS, and so the power manager needs to be extended to know a co-ordinated set of voltages to apply to each performance domain to move all its sub-components up through a set of frequency regimes.

(2013) https://patents.google.com/patent/US20140208135A1 seems like another patent based on working around sad third party hardware. The idea is that you may put an agent to sleep, but some other agent may (inappropriately) generate the wake code for the agent. How to fix this? Well, the fabric sits between every agent and every other agent, so you turn the fabric into a censor, blocking any inappropriate wake-style requests until the central power manager confirms that, in fact, this device is allowed to wake up.

But that's old school! By mid-2013 we've moved from the old era of Apple having to integrate third party HW to the new era of an Apple SoC. With this comes a much more designed power approach exemplified by https://patents.google.com/patent/US10303238B2. At a high level we see here both a PMU and a PMGR. As I understand these, the Power Management Unit
supplies actual power (ie it is what ensures that x amps flows into the SoC at a given time, that this rises or falls as demand changes, but never exceeds danger limits) while the Power Manager implements all the power saving stuff we've previously discussed like twiddling registers and changing voltages while

ensuring everything happens in the correct order and remains in spec.
More significantly we see the existence of a CPU Complex, and an addition to this complex of two important elements, the APSC (automatic power state controller) and the DPE (digital power estimate). In a way these are not primarily about saving energy (unlike the PMU and stuff we have previously discussed), they are about ensuring that the maximum power draw never exceeds what the battery can supply. The APSC controls things like how how many CPUs can be powered up at a particular voltage/frequency setting, while the DPE tries to track instantaneous power usage and ensure it's always within bounds, if necessary reducing the issue/execution rate of instructions for a few cycles. The APSC is programmed so that "normal" code running on all cores will not exceed limits, but in theory power virus code could exceed limits, and the DPE is there to catch when limits are exceeded and signal to the offending CPU(s) within a cycle or two that they need to throttle instruction issue or execution. (I already mentioned the patent for the CPU tracking the count of "heavyweight" instructions and throttling those if they are too dense. That's in 2011, but note that it's internal to the CPU. These mechanisms I'm describing move the management to the core complex and to the entire SoC, so they can allow eg one core execute vector code at full speed if the other cores are powered down, or are running undemanding integer only code.)

An interesting side note is that when a new core is powered up (a process that takes up to 10 μseconds), there's a concern that even though the new voltage/frequency settings will "eventually" be safe, there might be overload spikes during the transition, and so the effective clocks to the CPUs are halved (easy to do, as opposed to the more complicated manipulations of a general frequency shift). All in all it looks like they tried to think of everything -- and mostly got it correct except for the one small detail of forgetting that the maximum power a battery can source will eventually drift below spec...

The companion patent https://patents.google.com/patent/US9195291B2 describe how the power estimator works. It receives ongoing counts from each CPU of "significant" events, eg how many vector instructions executed per cycle, aggregates these to a per-CPU instantaneous power estimate, and sends per-CPU throttle requests as necessary. The number of "power events" over some period of time of time is tracked, and if this is too high the fact is reported to the PMU and PMGR, and the frequency/-voltage settings for the offending core(s) are reduced; and similarly if a core has been operating within spec for some period of time, its frequency/voltage is allowed to rise. I assume this, essentially, is what Apple has meant by saying they perform DVFS is hardware as opposed to having an OS driver make these modifications (obviously at a far slower rate, and with less detailed info).

This 2013 model of DPE was concerned with not exceeding the allowed power draw. But you can look at it from a different direction. Rather than worrying about a power virus, what about code that uses less power than we modeled, for example code that is purely integer with no use of FP or vectors? This is the subject of (2014) https://patents.google.com/patent/US9195291B2. We use the same digital power estimate but if the DPE detects a pattern of substantially lower estimated power than has been budgeted for, the PMU+PMGR are again informed, and the voltage for that core is allowed to drop slightly (while maintaining frequency), or frequency is allowed to rise while maintaining voltage.

Can we scavenge some voltage in any other way? Well a non-obvious fact about digital circuits is that there is a (not unrealistic) temperature+frequency range for which they run faster at higher temperatures (ie you can run them at the same frequency but slightly lower voltage if they are hotter).
(Don't confuse this with power issues! Running hotter is not something you especially want to aim for because it increases leakage current; but if the world around you has heated up your chip, can you make use of this fact?)
(2012) https://patents.google.com/patent/US8766704B2 does the same sort of thing as the previous patent; it detects if the SoC is at such a higher temperature, and if so, slightly reduces the voltage applied to the things like the CPU while maintaining frequency.
This is an extension of the idea in (2009) https://patents.google.com/patent/US8169764B2, which talks generically about more precise temperature compensation for the f vs V curve rather than just assuming a single curve at all temperatures.
If this sort of stuff excites you, there is more of it covering things like how to calibrate thermal sensors, how to run the control loops that ensure the SoC never overheats, how temperature affects radio frequency components, how these power and temperature measurements are communicated to the outside world while debugging/optimizing the device, how to extrapolate a few local sensor readings across the rest of the SoC, etc etc.
Some patents seem like refinements of an older idea (like a way to use smaller voltage guard bands) that should be irrelevant given these more sophisticated new ideas like measuring circuit behavior, or power modeling. I *think* what much of this boils down to is that there's a lot of IP on a SoC, and it's not all amenable to the sophistication and refinement of the CPU power reduction schemes. For something cruder like a memory PHY you may be stuck with older techniques like guard bands, either because the problem does not allow for a better option, or because there are enough higher priority issues with this particular IP block to postpone anything but refinement of the existing technique.

The culmination of this is (2019) https://patents.google.com/patent/US10948957B1. The earlier patent estimated power by multiplying event counts by a fixed weight. Now
- the weights can vary, and
- the exact energy (over some time range) is measured and compared with the estimate based on these weights.

The weights are then updated the bring the two in sync.
Obviously this allows for a more accurate estimate (not least because, again, the energy used by parts of the SoC will vary with the temperature of the SoC, and with its age). The patent also points out that, apart from using the estimator to limit instantaneous power (given battery constraints), it is also used to limit total energy released over some amount of time, to maintain thermal limits.
This has a companion patent https://patents.google.com/patent/US10969858B2 which is obviously (though it never says as much) about AMX. With AMX everything we've said about power draw becomes that much more of an issue because a single AMX operation can involve so many simultaneous floating point executions. The patent describes an even more sophisticated power estimator for each cycle of the AMX engine taking into account

- (at least approximately) how many operations will be performed, based on the sizes of the input vectors/matrices
- a model of the AMX pipeline
- not just absolute current levels, but also the inductive power resulting from rapid *changes* in the current sourced by the AMX engine.

So let's consolidate.
- We have the OS deciding on a generally appropriate performance level for each core based on what it knows (like number of processes that want to run and their associated QoS).
- The OS tells the PMU which tells the PMGR to put each core in the appropriate DVFS state.
- The APSC and DPE estimate from the overall activity level of a core, and its temperature, whether the voltage level can be shifted slightly down (to save power) or needs to move slightly up (to service many expensive instructions)
- The goal, at a per-core level, is to keep the voltage right on the edge of acceptable, to save reduce but ensure that power events (when execution of vector instructions has to be halted for a cycle or two) are at the correct level -- not zero, but not too frequent

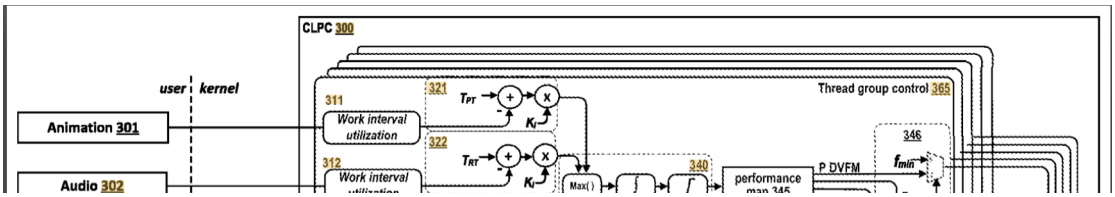
OK that sounds great and we have high-level control from the OS, and low-level per-cycle/per-core control. What's missing is something in the middle. In particular we don't want a situation where all four cores simultaneously either want to run vector code, or decide that they're doing too much vector work and want to stop; both will generate large current swings and noise. We won't get this from the above per-CPU DPE control mechanism.

The solution for this is (2016) https://patents.google.com/patent/US9798375B1. One part of the solution is to give each processor a pool of "energy credits", which are used up as the CPU engages in more heavyweight work. In principle if you stagger the times at which each core gets given its credits, and each core runs till it has used up all its credits, you'd do something to offset the times at which multiple cores all want to run all their vector pipelines simultaneously. But in fact Apple do something more sophisticated than that, making it probabilistic, as your credits run low, whether an instruction is allowed to execute in this cycle or not. A random or pseudo-random value is compared to the credits available, and if smaller, execution can proceed. This will both throttle the cores that have been using excess credits as their credits deplete, and will throttle them at different times, and with different timing patterns, thereby avoiding large current spikes.

This is followed up a few months later with (2016) https://patents.google.com/patent/US10452117B1 which extends the above idea to allow either core complex (P or E) to transfer any unused credits over to the other core complex, so basically allowing either the E- or P- processors to run a little more aggressively if the other complex is not being very aggressive (or, to put it differently, sharing the credits across all processors, rather than maintaining two independent pools of energy credits).

There remains the issue of how the OS decides on "appropriate performance levels". This (2016) https://patents.google.com/patent/US20170357302A1 is a basic explainer, with (2017) https://patents.google.com/patent/US10884811B2 giving much more detail. (This now very much straddles like line between OS/API functionality, but helped out by a lot of SoC/CPU hardware.)

It’s hard to convey just how glorious this second patent is! The ideas are (once you finally understand them) pretty impressive, but the whole thing looks like a briefing from the Pentagon, complete with thousands of acronyms and impossibly complicated diagrams like



Before even trying to understand this patent, understand the problem to be solved.
OS scheduling has always consisted of trying to balance off various desiderata.

On traditional (but fairly recent) desktop systems, the goal is to maximize responsivity while keeping throughput reasonable. This tends to be done by heuristics as to what are "user interacting" apps, and giving those priority boosts. There was a time when VM concerns were an essential part of this, and much of the scheduler's prime responsibility was to prevent thrashing (ie running simultaneously more than one app that was using a large amount of DRAM); presumably this is still a background concern but of much less immediate import.

Along a different dimension we have server scheduling. In this case throughput may be considered highest priority, and the most important trick is trying to pack together on different cores applications that each stress a different part of the overall machine, hence a few high memory bandwidth processes together with a few low memory bandwidth processes, perhaps a high FP/SIMD process on one hyper-thread together with an integer-only process on the second hyperthread, and so on.

Both of these versions of the problem are trying to select from a set of runnable processes, with a set of (somewhat known, depending on exactly what the SoC and the programmer provide) characteristics onto a set of cores, to optimize for something like performance. As a second tier goal, the OS will futz around with DVFS, but overall without much intelligence beyond a few very basic points like "don't exceed a certain temperature or power draw" or "if all the processes are known [somehow...] to be background, then reduce DVFS. Those readers who follow Linux will know of the never-ending travails of Linux and Android scheduling in the face of DVFS, a matter that's still not really satisfactory after all these years.

Before the truly complex solution, let’s consider the easier 2016 patent which is appropriate to A7..A10 class hardware. Neither clusters nor *visible* E vs P cores are part of this hardware.

Apple's goals are to hit performance targets (eg smooth UI) at minimal energy expenditure. The tools available are
- how many cores to power up
- DVFS of the cores
- less obvious, but also relevant, modifying the frequency of the SoC fabric and the DRAM

The basic flow of control is to have a first stage of scheduling that establishes performance goals. Think abstractly of "achieve state X by time Y" where X is some approximately knowable fraction of the total amount of work we want to achiever by a deadline.
How do know how fast to run the machine to achieve those goals? This is the role of the CLPC (closed loop performance controller) which essentially means you keep measuring your progress, and if you are going faster than necessary to meet the goal, slow the machine down; otherwise speed it up.
This means in turn that (to a first approximation) we describe performance as a monotonic map from n cores running at maximum frequency down to 1 core running at minimum frequency. Depending on how we're meeting the target, we continually move up or down this performance map.
[Of course there are two extreme cases of "as fast as possible" which is the default for third party code where nothing special is indicated, like, eg Geekbench code; and "as low as energy as possible" which will be the case for almost all the time the screen is dark and only background code is running. However most code that actually runs is media decode, animation, things like that, which fit this deadline model.)

But that just gives performance, without energy concerns. So the output of this stage passes through a second level of control which worries about energy and performance issues.
One part of this concern is easy. Things like device skin temperature, temperature at various extreme parts of the SoC, and short-term power draw are tracked and high performance levels which might cause these to be exceeded are dialed back.
Beyond that the efficiency controller is tracking an energy per instruction metric and trying to optimize this while not impairing the performance goals.
An especially interesting part of this is that, although there is an override mechanism (which forces the energy/instruction cost to 0 as long as it is active) most of the time the controller is trying to maximize energy efficiency. This might sound bad, like it will hurt performance, but mainly what it's saying is
- if you're constantly waiting on DRAM, then running the core at high frequency does you no good anyway
- if you're not running very wide (hard to predict branches, or long dependency chains) you can't take advantage of the big core anyway, so why waste power keeping you there rather than on an E-core?

To make all this work well
- one needs good metrics, and a lot of the patent is about how the energy is measured (over "long" time intervals) and estimated (over "short" time intervals) to inform these decisions
- one also needs a good control framework. Anyone who has ever tried to write control loop software knows that it's very easy to have the system oscillate wildly between too fast and too slow; or alterna-

tively if you try to prevent that, to respond to changes so slowly as to be useless. So another large part of the patent is about that control framework.
- not stated, but an obvious and easy extension would be metrics that are somewhat orthogonal to these primary metrics. For example performance might be lower than we wish because either the CPU is too slow or DRAM is too slow. This makes our control problem two dimensional, but the basics remain the same – change one or the other of the DVFS knob and the DRAM frequency knob, see how performance and/or energy efficiency changes, this allows us to calculate scaling coefficients (ie derivatives) telling us the response to both of these knobs, and we can use those scaling coefficients to calculate an optimal point (which will of course keep changing as the code keeps changing).

This is an adequate first start but we need something much more sophisticated for the A12 and later.We have the same concern of course, for responsivity and low energy usage, but we now have the complications of
- there are both P and E cores
- these are grouped into clusters and clusters run at a fixed frequency. So we can, eg, use three cores of a 4-core cluster, but we can't run two of the cores as fast as possible and two as slow as possible

To do this as well as possible Apple introduce a few new concepts.
• One is the *Thread Group*, a group of threads that are scheduled together, so in a sense the cluster is given a unit of work rather than each core being given a unit of work. Thread Groups are dynamic. They are constructed at the start of execution from various data (heuristics, indications by the programmer, ...) but threads can constantly move between Thread Groups if this makes sense. Various OS technologies allow the OS to see when two threads (perhaps from the same process, perhaps from different processes) are working together, and these threads will be united in a Thread Group to schedule together as much as possible. (One thing I never saw an answer to is what happens when one member of, say, a four element thread group, is required to yield, eg while waiting for IO. Perhaps the answer is that along with the "large" thread groups there also threads with no obvious affinity to any other thread, ie singleton thread groups, and these are opportunistically executed whenever such an occasion arises?)

• Another concept is the Work Interval Object which is a way to describe a task (perhaps worked upon by multiple threads) which has a known deadline and a known progress towards that deadline. The point is that for tasks for which this model is a good abstraction (think eg of audio playback or UI animation) the task (ie the set of threads) can let the OS know of its performance as it progresses, and the OS can see that the threads either need more performance, or can relax a little, to meet the deadline.

Neither of these are meant to describe all code under all circumstances. But they do cover a lot of types of code, and they can be used by the OS to perform substantially better scheduling (ie fast enough, while saving a lot of energy).

Bearing that in mind, lets look in more detail. From the parts I can understand, I think the essential idea

is
- the OS constructs an object (let's call it a *task*) for every thread that wants to run, along with an associated performance goal
- a performance goal is anything that ultimately turns into some sort of measurable rate (so many *x* per second). Certain tasks that are closely tied to the OS can provide precise versions of this performance goal (for example IO may have a goal of so many blocks/second, threads handling animation will have a goal of 60 frames/second). Deadline goals (ensure this frame is decoded by time T) can be converted into a rate goal as long as there is some reasonable proxy for what fraction of the frame has been decoded so far. A task can have more than one goal, so both a "throughput" type goal and a latency type goal.
- as code becomes more and more "normal" code, it's probably going to provide less and less of this info, with the most non-detailed levels being tasks that are simply associated with a QoS, and I *think* the QoS is associated with an amount of energy per instruction as the performance goal; so at one extreme we have "infinite nJ/ instruction" for maximum performance, at the other extreme we have "nJ/instruction as low as you can get" for background work, and in between we have specific "nJ/instruction" levels.

There are also a few non-obvious goals whose cleverness only becomes apparent after some thought. For example one of the goals is (essentially) time spent runnable but not running. Who cares? Well, all the goals mentioned earlier act to optimize a single task, but if we have more tasks than resources, we don't just want each task running at optimal CPU speed, we want the CPU speed to be higher, so that the core can do the work of more than one task over an interval. This runnable-but-not-running metric covers this aspect of sharing between tasks.

- the scheduling machinery ultimately wants to achieve that each task gets as close to its performance goal as possible, while
+ never exceeding power limits
+ never exceeding thermal limits (both of these are hard constraints)

The "main OS" provides the tasks, the hardware provides a whole lot of sensors, and a low-level subsystem (which may be part of the OS, or firmware, or hardwired, or a microCPU on the SoC) tries to balance all this. The primary way of doing this is via Closed Loop Engineering which is a fancy way of saying
- continually compare the vector of goals to a vector of results (eg actual blocks/second vs desired blocks/second, actual energy/instruction vs desired energy/instruction)
- depending on how closely they match, move a "control effort" up or down.
The control effort is a scalar indicator that is supposed to indicate (as best possible) how much "performance" to throw at the task. It varies between 0. and 1. Each value of control effort is mapped to a table of "performance effort" which we can think of, for now, as essentially starting at 1.0 with a P-core running at maximum frequency, and moving downwards till we get to at 0.0 an E-core running at 400MHz or whatever.

The control effort is mostly discovered by the closed loop process I described above. The thread starts at a control effort of 0 (so minimum E-core speed). Next time round, if the desired metrics are too far from where they should be, the control effort is bumped up a little; and so we go, until hopefully at some point things stabilize.

So this is the basic idea as of 2016, but there are extra complications which are the subject of 2017. Some of these include
- in many ways the object that executes code is the core complex, not the individual core. I *think* each core complex can only run at a single frequency.
At first blush this seems non-ideal -- you can, it is true, power down individual cores, but you can't run one P core fast and the other one slow. I suspect this is because of the tight coupling of all the cores to the L2. To access the L2 rapidly, it needs to be synchronous with each core, which means all the cores (and L2) are all running at the same speed…
We have seen various patents that allow for slightly shifting the voltage of a core while at a given frequency. If the cores run on separate power planes, this remains feasible, and may work well enough that Apple doesn't consider it too much of a problem requiring this equal frequency across a cluster? I can't tell from the patents I've seen so far if all the fancy stuff Apple patented about voltage adjustment are in fact per core, or complex-wide.

- the actual object Apple wants to deal with for most scheduling is something called the *thread group*. The idea seems to be, approximately, that we want to schedule what looks to the user like an "application", rather than scheduling threads independently. But we don't just consider the threads created by the application because a lot of work (animation, audio, decode, IO, ...) is done by OS threads on behalf of an app, so we dynamically move threads around into these scheduling constructs, as eg the audio thread does work on behalf of some app.
- each thread in one of these thread groups has its sets of performance goals, and these are all compared with measurements to generate a final control effort for the thread group as a whole
- this effort is again looked up in a table, but now the table is not just a list of single processor states, at maximum effort it might have 2 P cores at maximum frequency, down to 1 P-core at low-frequency or 4 E-cores at high frequency, down to 1 E-core at lowest frequency.
At various stages these desired performance levels are over-ridden either by the instantaneous power manager (which believes the battery, in its current state, cannot source the current required) or the thermal manager (which believes that if this performance is allowed the device will get too hot).

Along with all the above, also continually measured for each thread group are E- and P-complex residency (which is unsurprising, one can imagine ways this might be used).
More interesting is that also measured (presumably using counters deep in each CPU) is RS-residency (reservation station residency) which measures on average how long each instruction waits before being executed. Ultimately this is a measure of how much the code is actually using the capabilities of a large core -- if it is mostly waiting on DRAM, or has long runs of serial code that cannot exploit the width of large core, then, regardless of all its other goals, there's probably no value in running it either at high

frequency (waiting for DRAM) or on a wide core (serial dependencies).
There are also implications that the system is constantly measuring the types of instruction used, so that it knows if a lot of vector work is being done by a particular thread.
These measurements based on CPU counters appear to be attached to threads as extra attributes, and the main way they are used is to decide between "overlap" cases -- if the performance range is a good match to either a fast E-core, or a slow P-core, which should we choose?

So we have, for a thread group, a desired number of cores of a desired type at a desired frequency, and the threads are all marked as such.
They then go into either an E- queue or P-queue and are appropriately pulled out of this queue for execution.

Now if you have read all this so far, some things might seem problematic! In particular the system seems to really want to schedule a thread group as only P- or only E-. This would seem to have two problems.
The first is "what if I have, say, lots of hard-working threads, more than I have P-cores?". This is dealt with. The idea is to hold off on using E-cores for a while, but if this state of lots of P-threads persists, the system will give up and start moving threads placed in the P-queue across to the E-queue (presumably under these circumstance the understanding is "we are now trying for maximum performance" so issues of frequency no longer occur -- we have already dialed the P's to the maximum, and we'll run the E's as hot power and thermals allow).
I think it's important to recall the problem Apple is trying to solve! In a sense the "I have a large computation that uses lots of cores and goes on for many minutes" problem is very easy, with the obvious solution running every core as fast as allowed.
But that's not the hard problem; the hard problem is the one of "I have a constant stream of small, short-lived tasks; some of them have no performance concerns; some of them demand smooth animation; and while executing them all (on up to 6 cores) I want to use minimal energy". So the system is biased towards starting everything at the low end of "try running all the jobs, one after the other, on a single slow E-core, and heck, maybe that's good enough?" while climbing up from that minimum rapidly if circumstances indicate it's necessary.

The second issue is "what if my problem naturally decomposes into a few performance threads and a few supporting slow threads"? The answer seems to be that for almost all practical cases (where the slow threads are various Apple support threads) the OS knows how to handle the appropriately; and if you insist on writing a very strangely unbalanced app, there are APIs you can call to modify thread grouping.

I don't know how this works out, but there doesn't seem to be an answer. The answer may be at the level of the thread groups -- the thread groups are supposed to be constructed not just of threads

working together, but of threads that have similar performance demands, and any threads that seem to be drifting from the rest of the app maybe get moved to a different thread group after time? Presumably, like all this stuff, if the issues only exists for a second or so, who really cares? And if the issue persists for a while, some sort of high-level movement of threads to a better-matching thread group seems good enough.

On the other hand this scheme also has advantages. For example it naturally groups together (running at the same time, on the same core complex) threads that may be interacting (producing data for each other, sharing data, using the same locks), and such threads will have rapid communication through their shared L2.

So what we get from all this is a collection of threads, tagged by core and frequency, that are placed in E and P- runnable queue. Finally at each scheduling operation we try to pull out from these queues something that matches all the work we have done.
Once again, the *common* case, the one we are optimizing for, is that most cores are idle most of the time, and we're not in fact trying to pack every core as tightly as possible with work!
If we have four independent light-weight tasks lined up, the preferred scheduling may very well be to run them all sequentially on a slow-E-core, rather than fire up four E-cores, or run that E-core any faster. Obviously there are escape mechanisms whereby the system is tracking if the runnable queues are growing too large, at which point the obvious types of things will kick in, but the interesting stuff Apple is doing is primarily about keeping the system "feeling" fast while in fact most of it runs as slow as possible, and it ramps up to running more cores faster as effectively as possible. (Effectively meaning:
- know how fast you want to be, and compare that to what you're achieving. Measure, measure, measure!
- try to keep threads that are working together scheduled together)

Following this particular thread goes into a whole other world of OS technology, scheduling, transferring priority between threads, etc. Far far from our starting point of CPUs, so we'll leave it here!

Naturally all this power stuff has its counterparts on some other IP blocks, most obviously the GPU. For example an early version is (2011) https://patents.google.com/patent/US8856566B1.
This matches the thermal bounds tracking aspects of the CPU scheduling described above, but another way to think of it is that it's the equivalent of Intel's turbo'ing based on thermal inertia. The tech details differ, but the goal is essentially the same -- keep track of when the device is not running extra hot, and allow that to accumulate "credit" so that you can run the device hotter than optimal for a brief period, until the credit is used up (ie the thermal mass has absorbed all the heat generated, and continuing will raise temperatures too high).

- move what knobs you have available (primarily whether any thread goes on E core or P core, and the core frequency) to bring these two into better alignment
- while ensuring (obvious) that you don't go out of bounds and (less obvious) that you don't create state oscillations that grow larger and larger (eg run a CPU too slow, then panic and run it too fast, then panic more and run it even slower, ...)

This all sounds plausible enough as a rough start, but obviously many details are omitted. The part they seem to care about the most for the legal purposes of the patent is the use of control theory, which is not exactly the part we (as investigators of the system) are most interested in!
Things are improved (in the sense that we get more detail -- too much!) in
which of all patents I have read looks the most like a Pentagon briefing! God help you understanding this, but the new ideas that are introduced include
- the OS cares more about *thread groups* than individual threads. Thread groups appear to be something like a set of threads all working together to achieve the goals of a single app; but they are somewhat dynamic so that (I think this is how it works) OS threads like an animation thread or a media playback thread will join the group of an app while they act on behalf of that app

- these thread groups are treated as primary scheduling units (still using the previous ideas of a performance goal or two attached to each thread, ie task).
I think the idea here is to back away from the traditional OS model which schedules threads as the basic units, with a byproduct that an aggressively threaded app gets a lot more resources than a non-threaded app. All this dynamic thread group business seems to be creating a single object representing "all resources required by one app at this point in its execution", to be scheduled as such, and balanced as such against other thread groups (ie other apps).

- based on thread groups and matching target metrics to measured metrics, a map of the best performance schedule for this thread group is created (ie each thread gets mapped onto a P vs E core at this frequency)

- these maps are then read by the *thread* scheduler (as opposed to this earlier app/system scheduler) which puts each thread into an E vs P queue, with the appropriate DVFS info attached to it. At this stage there may be some slight juggling of the queue across different thread groups to ensure maximal occupancy of every core.

Along with all the above, also continually measured for each thread group are E- and P-complex residency (which is unsurprising, one can imagine ways this might be used). More interesting is that also measured (presumably using counters deep in each CPU) is RS-residency (reservation station residency) which measures on average how long each instruction waits before being executed. Ultimately this is a measure of how much the code is actually using the capabilities of a large core -- if it is mostly waiting on DRAM, or has long runs of serial code that cannot exploit the width of large core, then, regardless of all its other goals, there's probably no value in running it either at high frequency (waiting

for DRAM) or on a wide core (serial dependencies).

One thing to bear in mind as you read through these two patents is that I *think* each core complex can only run at a single frequency. At first blush this seems non-ideal -- you can, it is true, power down individual cores, but you can't run one P core fast and the other one slow. I suspect this is because of the tight coupling of all the cores to the L2. To access the L2 rapidly, it needs to be synchronous with each core, which means all the cores (and L2) are all running at the same speed.
We have seen various patents that allow for slightly shifting the voltage of a core while at a given frequency. If the cores run on separate power planes, this remains feasible, and may work well enough that Apple doesn't consider it too much of a problem requiring this equal frequency across a cluster?

A third issue that is not address is scheduling "closely interacting" threads that will be engaged in a lot of communication and lock-sharing. You'd want such threads to be scheduled on the same core complex (both E, both P, same of each when there Apple has larger SoCs with multiple P complexes) so that they can communicate rapidly through the L2. The mechanism as currently described does not seem to incorporate anything linking two (or more) threads together as "ideally schedule on the same core complex".

This all sounds good (once you see the big picture) and it certainly seems to work well for all the current use cases! iPhones run at low power, M1 macBooks feel very responsive, etc.
One thing that does seem to be missing is any sort of concern for
- programmer modification of thread topology. The Linux crowd go on about this a lot, the Windows a little. Apple seems uninterested in catering to them.
- affinity (ie retaining a thread on the same core across scheduling to take advantage of the prewarmed cache and branch predictors.) Apple's scheduler for intel seemed to positively fight affinity, constantly moving a thread, and I think this was to take advantage of thermal inertia when there were one or two high priority threads on a many-core system. This seems unimportant for Apple's cores. Perhaps they use affinity at the final stage of scheduling, but don't consider worth putting in the patent?
- (most important) scheduling nearby threads that will be engaged in a lot of communication and lock-sharing. You'd want such threads to be scheduled on the same core complex (both E, both P, same of each when there Apple has larger SoCs with multiple P complexes) so that they can communicate rapidly through the L2. The mechanism as currently described does not seem to incorporate anything linking two (or more) threads together as "ideally schedule on the same core complex".

- every executable object that can be scheduled (call it a thread) has an associated QoS
- each QoS has an associated performance price it is willing to pay (so many nJ/instruction)
- simultaneously the system is constantly measuring the instruction count and energy usage of each CPU so that the OS (or perhaps the PMGR) has an approximate idea of what config (P vs E core, at what frequency) will cost a given nJ/instruction; and these two are matched up.

But most of the patent is written in control theory language (which I don't speak), wwith system A modifying the choices of system B, and in turn being modified by system C, and I don't know what problem this complexity is supposed to be solving. I think maybe it's supposed to provide hysteresis so that each core isn't madly jumping from one state to another, but only moves after a few rounds of scheduling establish that it really isn't the best fit for the task?

Where does the coupling between frequency and voltage come from? Obviously these could be set at production time, but the settings would have to be sub-optimal, to handle both variations across chips and variation as the chip ages. Instead Apple has (2009) https://patents.google.com/patent/US7915910B2 which occasionally performs a self-test (slowly reduce the voltage while keeping a particular frequency, until the test returns incorrect results) and stores those voltage/frequency pairs. This is an extension of the earlier (2005), also PA Semi, https://patents.google.com/patent/US7276925B2. The earlier patent seems to test that a few types of circuit elements work as voltage changes, whereas the later patent seems to test that an entire digital subsystem (cache, adder, whatever) works.

The patent doesn't say as much, but I assume a similar idea was used, for example, to establish the minimum voltage at which cache banks can sleep without losing data. Ultimately for cache banks we move to this fairly amazing patent (2016) https://patents.google.com/patent/US9922699B1. The idea is that each cache bank is powered through one of a set of (say eight or more) power diodes, each of different size and hence voltage drop. Based on temperature conditions, the system decides which diode to use for a given bank, thereby using the minimum voltage possible under current conditions.

Apart from the above ideas, a different power-saving strand is to use dedicated hardware as much as possible. This begins with (2008) https://patents.google.com/patent/US8359410B2 which talks about using a dedicated audio DSP and codec to handle either music playback or improve the audio quality of phone calls. Again this seems a no-brainer, but Apple say that the state of the art at the time is to perform this sort of work on the CPU where, of course, it consumes more energy.

branch

https : // patents . google . com/patent/US9626185B2 - diagram of branch pipeline and training stage
https://patents.google.com/patent/US20150039860A1 proof that Apple are using checkpoints
https://patents.google.com/patent/US20150293577A1 a few loop buffer details

Now let's consider Fetch and everything related to that. (More so than other sections, as I wrote this I kept finding that a concept I was explaining now relied on a future concept. I've done the best I can to order the material, but you may find it worth reading this once fairly rapidly, just to get the basic ideas, then again to see how the ideas all fit together.

Like so much, most people have in their minds a vague idea of Fetch that was maybe appropriate for the late 1990s but has little modern relevance. The way to think of Fetch is, like most modern micro-architecture, to start by thinking of the problem in abstract terms.

Program execution consists of a sequence of instructions, mostly short sequential runs of instructions (say five to ten instructions) followed by a new sequential run. The gaps between these sequential runs are, of course, branches of one form or another.
Now the *important* point is: a natural division of work arises. From Decode onward, the machine has no interest in the gaps between these runs of instructions. As long as we have branch prediction, the interior of the machine just sees a stream of instructions to execute. It's the job of Fetch to construct that stream as accurately as possible; it's the job of mispredict recovery to pick up the mess when something goes wrong; and it's the job of the interior machine to ignore both those two other jobs.
If you take this seriously, you should realize that we have exactly the same situation as much of the rest of the machine: we have two pieces that can operate mostly independently, and we should put a (possibly large) queue between them so that when one is blocked the other is not. Decode just wants to extract a steady 8 instructions/cycle from the queue. Fetch should try to deposit into that queue as much as possible per cycle. Some cycles Fetch puts nothing in the queue (I-cache miss), some cycles just a few instructions (the sequential run being fetched is not very long). To make up for this, when long sequential runs are encountered, Fetch should be capable of fully exploiting them, moving sixteen instructions or more from the I-cache to the Instruction Queue.

Next step in understanding. While Decode is going to operate as a blind engine that just grabs as much as it can (up to 8 instructions) per cycle from one end of the Instruction Queue, Fetch on the other end is going to operate as an asynchronous engine. This means that (in the absence of any indication of things going wrong) *every cycle* Fetch predicts
- where to load the next sequential run from

- how many instructions to load from that run
People talk of Branch Prediction (and we will get to that) but the more immediate concept you want to understand is Fetch Prediction, which refers to the two points above.

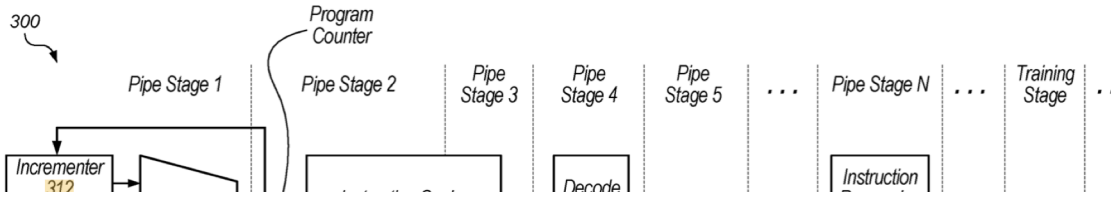
You may think this design is just obvious an common sense. And it's not new, it was proposed by Glenn Reinman in (2001) https://cseweb.ucsd.edu/~calder/papers/UCSD-CS2001-676.pdf *Hardware Optimizations Enabled by a Decoupled Fetch Architecture*, but you'd be amazed at how small the fraction is of people who get this and what it implies.
(The thesis is essentially two parts, the first describes Fetch, the second we'll cover later, and considers instruction Prefetch.)

Now let's consider further implications of this Decoupled Fetch design. Assuming prediction is correct, at any given time we have (distributed over multiple queues and pipeline stages) instructions that are in-order running from cache access stages through presence in the (deep) Instruction Queue through Decode, then Map and Rename. After rename the instructions become out of order.
Note a consequence of this -- if we realize that an earlier prediction was incorrect, we can correct it without too much pain a fair way down the pipeline. If we realize that instructions, even in the Instruction Queue, are incorrect, we can simply flush them and re-pack the Instruction Queue from the correct address. We lose a few cycle, but maybe not even those if we had enough queued instructions ahead in the Instruction Queue.
If we catch an error in Decode by then we've started to allocate resources (in Decode we allocate, at least, ROB slots) so we can still correct an error without a total machine flush, but doing so is more work. And so further on through Map and Decode – if we want to correct a Fetch error at these points, instructions are still in order, and so doing so is technically feasible, but requires a lot of care to deallocate all the resources that have been allocated.

So what, you say. Well what this means in practical terms is that we can utilize a number of different types of predictors, with different latencies. We need a basic Fetch predictor that can deliver a next fetch address within a cycle) but we can also have additional predictors that check the stream over the next few (perhaps two to even as high as five) cycles and can kill it, and reFetch, without too much pain and drama.

Here's the idea (from (2016) https://patents.google.com/patent/US10747539B1). Ignore the details, consider the big picture.
Pipe Stage 1 decides the next Fetch Address. This is derived from
- various single cycle predictors
- a multi-cycle predictor sending a correction signal (ie "flush the last two cycles worth of Fetch, and restart here")
- the Decode unit detecting an easy branch (direct, non-conditional)
- the eventual execution of the branch detecting a mispredict and forcing a flush
Ideally all but the last of these don't require much interaction with the bulk of the core, so they will cost

neither too many wasted cycles nor too much energy.



More consequences.
One average we have a branch say every 6 or so instructions. There have been designs in the past that were essentially driven by branches (imagine something like
- predecode [scan instructions and mark those of interest as a cache line is moved into the I-cache] marks branches
- Fetch pulls in the I-cache line up till the next instruction marked as a branch
- the address of that marked instruction is fed into some sort of predictor which indicates the next Fetch address)
This is easy enough to implement but it means Fetch halts at every branch, even non-taken branches. And good luck running 8-wide if most of your fetches pull in only ~6 instructions...

So it's simple math that to do better we have to stop treating branches in the stream as special; what matters as break-points in the stream; a branch that is predicted non-taken is of no interest to Fetch. This means that now, on average, maybe we're at ~10 instructions between taken branches. That's good, but a second issue is cache line boundaries. In the past it's usually not been worth reading two cache lines in a single cycle, rather you read till the end of a cache line, then pick up the rest of this sequential run next cycle. With 128B (32 instruction) cache-lines, that may still be feasible for Apple, the loss from runs that occasionally extend over a cache line may be small enough to ignore for now. But this once again reinforces the point that with all the things that can go wrong (I-cache misses, short instruction runs between non-sequential addresses, instruction runs that reach the end of a cache line) you really want Fetch to pull in as much as it can (much wider than the nominal width of the CPU) so that *on average* you maintain a flow of 8 instructions per cycle.
Apple is clearly getting close to the point where the sort of machinery I've described is no longer good enough. The next stage would be a predictor that spits out not one but two fetch predictions, the next sequential run, and the one after that, along with cache access machinery that can access two separate cache lines and move the results, appropriately ordered, into the Instruction Queue. A number of papers have talked about this, but I'm unaware of any actual implementation.

Now some general points about predictors, before we get specific.
In the early days of prediction,
- transistors were scarce enough that the same structure was used for both training and prediction. There may still be cases where this works well, but you'll notice with a lot of Apple predictors (not just branch prediction) that one structure is used for training, a different separate structure for generating predictions.

- people were somewhat fast and loose about the exact details of how predictors were updated. But prediction is now sufficiently accurate that you really don't want to pollute your predictor training with incorrect data. This has two consequences.
One is that you don't want irrelevant code (most obviously interrupts) being allowed to feed data into your predictors.
The second is that you don't want to train your predictors on speculative instruction streams.
This leads to a tension. On the one hand, you want every successive branch in the *speculative* stream to inform your prediction, because that is recent data that carries a lot of useful information about subsequent branches. But you don't want that same data locked forever in your long-term prediction data.
So the best predictors have a somewhat complicated structure (both storage and training) because they want to maintain one set of data that's informing predictions, but is provisional, along with a second set of long term data that's absolutely accurate, and they want to use both of these optimally.

- how do we handle context switching? Traditionally this has just been ignored, with an expectation that, sure, after you context switch things will such for a while until the predictor is retrained. Again, good enough for the old days, not acceptable if you want the best performance possible.
So how can one do better? I'll suggest three options I think are viable, and later we'll see what Apple does.
+ The (apparently) simple option is you just swap the branch predictor data every time there is a context switch. Add some appropriate instructions, get the OS to call them. Sure, it will work, but it suffers from the problem that there is no natural path from the branch predictor SRAMs to the rest of the machine. You have to add ways to extract, then replace this information and those paths have no value beyond context switching.
+ You could somehow tag every branch prediction entry with an ASID. When the predictor is referenced, we see if the ASID matches our current ASID. If so, we trust the predictor, if not we reset it to neutral. We would expect that in any given time slice, only a few branch prediction storage slots are used by a particular executable, so that, for the most part, at any given time the branch predictor holds relevant predictions (being used by the current app) plus predictions used by the last few apps, and if any of those apps are re-scheduled on this core, they'll be able to reuse those predictions. This should be familiar as the way many TLBs work (with an explicit ASID tag for each TLB entry) and it's also of course the way any physically addressed cache (ie pretty much all of them in machines of real interest) work, with the ASID being effectively embedded in the virtual-to-real translation.
+ Best of all, probably, would be have all branch prediction operate in physical rather than virtual space. Then you get the equivalent of every entry being tagged by ASID, but even better, you also get

sharing. Most code executed on Apple devices is code in Apple shared libraries, and while there are surely counterexamples, one would expect that prediction trained on one execution of such a shared library is usually a good fit to a different execution of the same shared library. Sounds good, but it does require substantial infrastructure in the design because the code, as written, presents all addresses (whether relative PC offsets for branches, or subroutine call addresses, or indirect branches for virtual calls and procptrs) as virtual addresses. So if you want to fight that, you need various back-channels or such sneakiness to convert the virtual address stream present in the instructions to a physical address stream.

So, still operating at the abstract level, think about how you'd implement such machinery. When the machine boots up you know nothing, so you have to bootstrap from there. The obvious easy solution looks something like
- by default we just keep executing forward, pulling in as many instructions as we can from our current position
- the machine downstream at the point of execution of a branch will eventually detect errors (ie the instruction after a branch doesn't match the PC that the branch calculated as being the next PC)
- these will be reported to the Fetch Predictor, the Fetch Predictor will record in a large table with entries like "at PC x2, there is a jump to address x3'"
- the Fetch Predictor is constantly maintaining 3 PCs:
+ startOfRunPC
+ branchPC
+ branchTargetPC
- using these we can also fill in, for previous entries, how long the Fetch Group should be, and what PC to look up for the next cycle of Fetch
- so eventually this table will be reasonably densely populated, so that each cycle we can look in the table for the address we plan to jump to

This gives the basic idea, now we need to fill in details.
First of all, in any given cycle what we try to look up in the predictor is
- the address of the next run of instructions, and the length of that run; both of which can be fed to the I-cache.
Suppose we don't have a hit in the Fetch Predictor? Then, in the absence of anything better, we keep going forward.
How many instructions to load for these unknown cases? One option is just load as many as you, the downstream will figure it out! But that may not be power optimal, if there's a good chance that you are stumbling blindly into unknown territory. Maybe a better option is to proceed one instruction at a time, slowly but cautiously, so that while you're in unknown territory you're wasting minimal energy as you build up a map? Or maybe simulations show the optimal length is 2 instructions of Fetch in this unknown case?

We want to correct errors as soon as possible.

One way to do that is at Decode (at which point you know where the branches are) you can handle various simple branches right away. For branches that are non-conditional (so you know they are taken) and for which the address is embedded in the instruction (eg branch to PC+offset, or the equivalent for branch and link [ie subroutine call]) Decode can check right away that the successor instruction matches the calculated target address and, if not, can flush everything after the branch and inform Fetch (including updating the Fetch predictor). This saves a few cycles compared to having to wait till the branch Executes or, even worse, Retires. We see that pattern in the Apple patents, in the Control Flow Evaluation block of Decode.
Even some more complicated cases could in principle be handled this way. Consider a branch-to-link-register or an indirect branch. Is it likely that these are directed to the instruction directly after the calling instruction? Of course not! So if the PC after a `BLR` (indirect call) or a `RET` is the current PC+4, that's likewise a strong indicator that we need to flush and train the Fetch Predictor for this PC. [should actually try to test for this case]

Another thing we can do is have certain especially difficult prediction cases (especially indirect call) offer a "halt" behavior. If we reach Decode and have reason to believe that a `BLR` is unlikely to be correctly predicted, we can just pause the Fetch pipeline until the `BLR` is executed and provides a value.
For simple taken/not-taken branches this is usually not worth doing because even if you guess the direction, there's a reasonable chance of being correct; but for indirect branches there are many ways to be wrong (wasting power) and only one way to be correct.

How much such a predictor actually be implemented? Essentially it's like a direct-mapped cache. Suppose we want 4096 entries. We'd take the 14 lowest bits of the PC (drop the 2 lowest bits which are always zero) and use those as index into a table. This is reasonably fast (it needs to operate faster than a cycle!) and reasonably accurate. But note one consequence of this will be aliasing, ie if we have two PC's that match in the 14 lowest address bits, then they both can't live in the Fetch Predictor table at the same time.
We can reduce aliasing by using more PC address bits. 14 bits are within a single page; beyond 14 bits we start using pageNumber bits. Do we want these to be virtual page number or physical page number? Virtual is clearly easier, but physical is probably doable and may be better in a theoretical sense.
We could also reduce aliasing by using cache type technology. The bare minimum is attach a tag (say a few more higher bits from the PC) and compare those to the current PC. A match indicates a good Fetch Predictor entry, a mismatch means ignore the entry. This will prevent using bad entries, but won't allow two aliasing entries to co-exist. We could attach two entries to each slot (ie now we've created a 2-way set associative cache for the predictor), but that's probably more energy usage than it's worth, and it introduces tag comparison into the critical path. (The previous validation tag does not have to be on the critical path, because we can kill a lookup that has already been sent to the cache if the validation tag indicates a mismatch.)

So we now have a single-cycle Fetch Predictor. Job over? Not even close!

We have accepted, for the sake of single-cycle performance, that the Fetch Predictor is not especially smart and not especially accurate. It's smart enough (basically do whatever worked last time) and good enough that most of its predictions are valid. But we want to augment it with a variety of specialized predictors that will take an additional cycle or two to validate the Fetch prediction, but if they disagree we can will the Fetch while the costs are still low, as previously discussed.

One obvious specialized predictor is the return address stack, since it's practically impossible to predict the address of a `RET` instruction without it, and trivial to do so accurately with such a stack. The main issue with a return address stack is simply making sure that you correct it (somehow) when something unexpected happens, most obviously an incorrectly speculated code path that generates an unmatched return or call before being corrected; but also a code path (using exceptions or longjmp or whatever) that breaks the usual rules of call/return.

Another specialized case is loops. Loops have the characteristic that they repeat the same code over and over – until they don't! Ideally you want to record whatever the exit condition for the loop is (usually, not always, a fixed count) and use that rather than being fooled by the fact that this loop has jumped backwards 1000 times so the way to be is that it will always keep jumping backwards. Loops also raise a whole set of possibilities for saving power. If you're confident in the loop, you can (for at least a few cycles) switch off the branch predictor, and the ITLB. Perhaps you can store the instructions in some sort of buffer and also switch off the I-cache?

Then there is what's traditionally called *the* branch predictor, the Branch Direction Predictor, the thing that guesses whether a given "branch based on condition" will or will not branch. The easiest versions of this are *local* predictors. Each branch is treated in isolation, and has, for example, a two bit saturating counter associated with it. The counter goes up each time the branch is taken, down each time it's not taken, and you guess the direction based on the current count value.
This is implemented like the Fetch Predictor – use some number, say 14, bits from the PC of the branch to index into a table of these counters. Like always, lookup can mispredict for two reasons – maybe the prediction was just wrong? or maybe the prediction would have been correct except aliasing meant that two branches were using the same predictor slot and kept confusing each other.

Even with this super-simple local predictor, there are improvements possible, but the big improvement is to use non-local prediction. This assumes that the best way to guess a branch's direction is not what it did last time, but what was the pattern of code that got us to this particular branch. This pattern of code is most easily encoded in a history vector that records, eg whether the last N conditional branches were taken or not taken. This history vector is hashed with the PC (eg use the last 14 taken/not-taken decisions xor'd with the lowest 14 bits of the PC) to index a two-bit saturating counter used as before. From this point on you can go wild with variations, but the current world champion (literally! https://jilp.org/cbp2016/program.html) is named TAGE, and it's basically an extension of the above history vector idea, but implemented in a way that allows for very long histories. The other trick you want to include is to generalize from the this basic history vector to a more sophisticated path history. Imagine

identifying the taken path of execution as, say a sequence of branch target addresses. Obviously this path uniquely identifies a path of execution (starting from some point earlier); just as obviously it's a lot of data! But as usual we can hash it down to something a lot smaller, but still (usually) identifying a unique path. So, for example, for every taken branch, we can shift the path history by 3 bits, then xor in the new branch target. And once again one can go wild with variations on this theme.
The end point of all this is that you want to be able to identify each unique path of execution (over some number of previous branch points), so that all your prediction statistics (eg your counters that go up or down) are associated with that unique path, and we can extract whatever is predicable and associated with that path.

Let's consider the above in more detail.
First suppose we have a local predictor, but we index into our local predictor using the *full* PC (this is a thought experiment!) Even with this crazy amount of storage, the best we can predict for any branch is essentially "what it did last time". We cannot track well structured patterns for this branch (it keeps alternating taken/not taken), and we cannot exploit correlation (whenever that branch is taken, this one is usually not taken). So prediction is adequate, not great, even before aliasing. Implementing this predictor in a practical way means we cannot use the full PC as index, just, say, 12 bits; which means beyond the theoretical inaccuracy already discussed, we now add in some degree of inaccuracy from address aliasing, where two different branches find themselves allocated the same slot in the table (because the lowest 12 bits of their PC's match), so they keep fighting each other over how to update the counter.

Now introduce branch history. Once again imagine an insanely unlimited amount of space. We can construct a perfect history for every execution of every branch – we know that to get to that particular branch we followed a path of fifty thousand previous branches in this order, with the path consisting of something like the address of every taken branch and the branch direction of every conditional branch along the way.
Now apart from impracticality in storing this lot(!) this does not actually help us! If we know that one (and only one) very particular path resulted in a taken branch, so what? The next time we reach that branch the path will have grown by a few entries, the branch corresponds to a different (not yet recorded) history, and we have no prediction.
Clearly we want some branch history – but not perfect branch history. How much?
What you probably want is something approximately like: "match the branch history backwards for as many entries as possible before the number of matching paths drops below a statistically significant level". Sometimes the closest matching path we can find (with, say, at least 8 data points of how the branch was taken) is a match over the past three branch events; ie for this case we'd want to use a branch history of three. But sometimes a branch 200 events back is tightly correlated with this particular branch and you want to use a constructed branch history that looks something like "use events around 200 entries back, then skip 180 events, then use the newer events" and this synthetic branch history will match a number of branches, and so will be a good predictor (will be tightly correlated with) the current branch.

Obviously this is still impossible to implement. But how can we use the idea?
Firstly we implement the branch history, as discussed, by extracting a few bits on each branch event, from some combination of the branch address, the branch target, whether the branch was taken, etc etc, and folding those into a path vector which might be something like 256 bits long.
Secondly we hash down this path vector into a variety of shorter indexes (let's say each 12 bits long) that fold in varying amounts of the path vector and the branch PC. At one extreme, we just use the lowest 12 bits of the branch PC and we have a fully local index. The next index may use the first two bits of the history folded in with 10 bits of the branch PC. We do this using geometrically more (that's the G, for Geometric, in both O-GEHL and TAGE") more of the history bits, sot 4 bits next time, then 8, 16, all the way to all the bits of the history. Once we get to the longer history lengths, we may drop some of the bits in the middle, on the assumption that the correlation is as described before, between an event 200 branches back and now, with most of the intervening events irrelevant.
This gives us something like nine indexes, into nine tables where for each table entry we maintain a 3 bit saturating counter. For each entry we also have 2 "usefulness" bits and a tag.
The usefulness bits are for updating the predictor; they mark entries that will be the best choice for sacrifice when we need to overwrite an entry.
The tag is a different hash of some combination of the PC and history bits from what we used for the index. The hope is that if we match on both the index (finding this entry) and the tag, then it's highly likely this entry is not aliasing, it really refers to a single combination of path+endpoint branch.

This seems complicated, but not really.
The index based on zero bits case you understand: every time this branch is taken, its counter (as determined by the lowest 12 bits of the PC) goes up, not taken the counter goes down.
The index based on 2 bits case is essentially the same: for each path that had a certain structure (eg two branches immediately prior to this branch were not taken) again we modify the counter up or down when this branch is taken or not taken.
And so so on across nine tables.
Your immediate response is probably one of two things:
- how can this possibly work? Isn't every branch history really unique? Well, yes, if you go out to the start of program execution. But you agree that the branch history going back two branches is probably not unique? There are just four possible taken/not taken patterns in the branches before this one, and in fact most of the time one of those four is strongly dominant, the other three almost never happen. And so it goes as you go further backward. The number of possibilities grows, of course, incredibly fast. But the number of actually executed paths, in most real code, grows much slower; most code follows a few, predictable, patterns of branches.
- then you worry about the reverse problem – doesn't this lead to crazy amounts of aliasing? Well again not really. Essentially we are hashing very long very different strings, and as long as our hash is not too dense (let's say we want to track 20,000 branches, but we are providing 36,000 hash slots) it's unlikely that two strings will match to the same hash slot; and we catch almost all those cases via the tag comparison.
So with those out of the way, the only question then is, we create nine indexes and perform nine

lookups. Which one do we use?
For prediction we use the longest match that works (ie matches tag as well as index).
For training, we use the usefulness bits to decide which table to update.

This should give you enough of the idea that, if you want details, you can read the TAGE paper, (2006) https://www.irisa.fr/caps/people/seznec/JILP-COTTAGE.pdf *A case for (partially)- tagged geometric history length predictors*, without being overwhelmed. I'd recommend it! It's a beautiful paper, easy to understand (once you have the basic ideas in place as above), and it covers the particular details of things like how you choose which table to update, and why.
This same idea (use long history vectors, and pull out the best match from matches at various length) can be used for other predictions, both indirect branches and value prediction, though in these variants you have to modify the "counter" that are using. For branches a single counter can track both the prediction (taken, not taken) and the confidence; for other case you need two separate storage items for these two tasks, eg a storage for a branch target address, and a separate counter of "how often has this address been correct vs failed".

What matters for our purposes is that humanity knows how to build remarkably accurate branch direction predictors, but these are slow enough that, to get real value out of them, you need a Fetch implementation like I have described – something fast enough to generate a prediction every cycle, plus a mechanism that can validate those predictions over the next few cycles. In particular for any particular Fetch Group, in parallel with the Fetch Group being loaded from the I-cache
- you want to know where the conditional branches are in the Fetch Group
- you want to test each such branch (before the last) to validate that it was not taken

- you want to validate that the last branch in the Fetch Group (if it ends the Fetch Group and is a conditional branch) was taken.
If you think about it, this is not exactly trivial! You could mark conditional branches in each cache line via pre-decode, but that doesn't help because, ideally, you'd be doing Fetch Group validation in parallel with cache access, to generate a correction as soon as possible.
I have no idea how Apple does it, but my guess would be that it involves the following elements
- the Fetch Predictor doesn't just store the next target address and the number of instructions to fetch; it also has a mini-map of the next Fetch Group. At the very least this would be a bitmap of which instructions in the Fetch Group are conditional branches; there may also be value in indicating indirect branches and returns (because if you know those are not present, you can save power by not activating those predictors?)

- even if you now know the PCs of the conditional branches that you now need to look up in your fancy TAGE predictor, there's a question of volume. Some Fetch Groups are short and have zero or one conditional branch, some may have many (all but the last hopefully not taken). How many conditional branches per cycle are you prepared to look up in your TAGE box? I suspect the optimal answer to this is to make the Branch Direction Prediction Validator machine yet another asynchronous machine that

is coupled to Fetch via a queue. So the Branch Direction Prediction Validator is fed (via a queue) a list of "PC's to check, and the expected taken/not-take status", processes some number (two?) of these per cycle, and generates results (validation succeeded, validation failed), to be given to a pipeline stage somewhere after I-cache to check their fetched instructions against this queue to discover a mismatch as soon as possible.
In the worst possible case (code that's just nothing but a sequence of non-taken conditional branches one after the other) if the queue holding these conditional branch PC's to validate fills up then, like when any queue fills up, upstream (ie Fetch) will be informed and will pause until the queue frees up as much space as required for progress.

The other common type of predictor is the indirect branch predictor. If all that did was guess the previous indirect branch target of this particular branch, that would be an adequate guess but would also add nothing to the Fetch Predictor. However TAGE ideas can also be used for indirect branch prediction, (2011) https://hal.inria.fr/file/index/docid/639041/filename/ITTAGE.pdf *A 64-Kbytes ITTAGE indirect branch predictor*, so you can likewise use a heavyweight but slow predictor for these uncommon cases.

There are also weirder specialized-case predictors that are possible, and if you look through the JILP Championship papers you'll see some examples of them, but it's unclear whether any industrial CPU uses them.

As usual, we now see what we can learn (via patents or experimentation) of the above hypotheses.
We start with (2012) https://patents.google.com/patent/US20140075156A1 *Fetch width predictor*, which validates much of the above. You should make the effort to read through it, because you will see just how much it matches my description above.

Beyond the explanations above, we also learn a few implementation details.
- Even as of 2012 (so around A6), Apple's maximum Fetch Width appears to have been 32 bytes. You may think of this as 8 instructions (already large for a 3-wide CPU) but remember this is ARMv7 days, so it could 16 Thumb instructions. Point is, Apple appreciated from the start that when you can gulp in as many instructions as possible, you do so, to make up for all the cycles where you're recovering from misprediction, waiting for cache misses, or whatever.
- Likewise even in that machine, Apple's Fetch Groups could straddle two consecutive cache lines.
- The index into the Fetch table appears to have a few higher order bits xor'd in with the low bits (which provide the primary entropy). My hypothesis for this is the same as when we saw the same thing being done for other predictors – it's possible that the low bits of an address are not perfectly uniform (linkers like to align things to page boundaries, compilers may believe it makes sense to align things to cache-line boundaries) and if the low bits are not perfectly uniform then you can boost the entropy slightly by mixing in a few higher order bits.
- Each entry in the Fetch Predictor is tagged by higher order bits of the PC. So, as I suggested, the

system can at least catch aliasing (when an invalid entry is pulled up) and try to do something appropriate for the “zero fetch knowledge” case rather than just believing the incorrect entry. This is probably most useful, as I suggested, for minimizing the damage after context switches.
- A technical concern with Fetch Predictors is what do you do with a very long Fetch Group? For example suppose that we have an unrolled loop body that is 30 instructions long, but we expect the usual Fetch Group to be 16 instructions long. There are at least two issues. One is how many bits do we use to store the Fetch Group length; more difficult is what if we want to store auxiliary information related to the Fetch Group (for example, as I suggested, the locations of conditional branches and perhaps also other types of branches)?
The standard answer in the academic literature for this is you allow “overflow” Fetch Groups. Basically for the address that corresponds to 16 instructions into the long Fetch Group, create a fake entry that corresponds to the next 14 instructions. It’s fake in the sense that it doesn’t represent the *target* of a branch, like most Fetch Group entries, but it’s perfectly legitimate in the sense that it represents a group of 14 sequential instructions, to be loaded from this address, and treated like any other Fetch Group. And Apple appears to be following this traditional answer.

Submitted at essentially the same time, we have (2012) https://patents.google.com/patent/US20140089647A1 *Branch Predictor for Wide Issue, Arbitrarily Aligned Fetch* filling in a few more details. These include
- The Branch Direction Predictor (as of 2012) was the O-GEHL version of the Perceptron predictor, which was considered to be the best predictor around that time before TAGE took the crown. (O-GEHL is an early version of the primary TAGE idea of how to handle long histories, while Perceptron refers to how to convert those histories into a tangible prediction).

- The Branch Predictor tracks its confidence level and uses this, among other things, to indicate whether further training is needed. (Most direction predictors calculate a number, maybe between 0 and 7, or maybe between -15 and 15. It's obvious that one end of the range vs the other end corresponds to taken vs not taken; and one can consider how far the number is from the extremes, say how close it is to 0 as opposed to ±15.). This training request is attached to the branch instruction early in the pipeline and propagates with the branch right up till Retire.
Suppose we have a branch for which we are very confident. Then it makes no sense to continue training the branch while it is predicting accurately; further training is just a waste of energy. Hence the value of this training request. Of course regardless of the training request, an incorrectly predicted branch has to transmitted to training!

- What about the issue of how to handle multiple branches in a Fetch Group? The Apple answer (at least as of 2012) is clumsy and inelegant, but not as bad as it first looks. Consider any of the direction predictors described earlier; easiest to imagine is the basic 2-bit saturating counter local predictor. We imagined our table indexed by the low bits of the branch PC, and the table holding a 2-bit counter for that PC.

But now assume that the table is indexed by the low bits of a cache line. In other words, suppose we were using 14 bits of the PC. A cache line is 128B, so 7 bits. So strip off the lowest 7 bits, and our table now has 7 bits (128) entries. Each entry is now a row of counters -- one for each instruction in the cache line, so 32 counters.
So if we want to access one particular counter, we would use the appropriate address bits (bits 7..14) of the branch's PC to find the appropriate row in the table, and then use the 5 bits 2..6 to identify the appropriate counter of the 32 counters. (Of course bits 0..1 of an instruction are always 0 and irrelevant to anything.)
The win in this design is that with a single Fetch Group PC we can identify the appropriate row of the Branch Direction Predictor, which holds prediction data for all possible branches in that line, and we can read out the entire row. At the point where we wish to actually validate the predictions in the Fetch Group (maybe at Decode, maybe just after I-cache access) we at that point know where the branches are in the Fetch Group , so we know which values from this set of 32 possible values to look at.

Your first impression on looking at that design is that sure, it may work, but doesn't it waste so much space? All those possible counter locations that correspond to non-branch instructions. No!!! What we have done is simply rearrange the same data that was present in the original (non-cache-line) design, with the same degree of aliasing, neither more nor less! A single row of the predictor does not correspond to a single cache line of instructions; it corresponds to all the cache lines of instructions that alias to this row (ie that have the same lowest 7 address bits of the cache line address). So while one particular I-cache line may use 8 of the 32 slots, another may use 5, and another may use 7. Aliasing exists – but no worse (or better) than before.

You second impression might be that no way this could actually work for a global (as opposed to a local) direction predictor, especially with the multiple tables and multiple weights of O-GEHL Perceptron. But it does! It all works as I described; just operate the predictor as you did before, but use as the address (to decide in which row to store a value) the 7 cache line bits, not the full PC address; and then use the remaining low 5 bits to decide which entry of the row to read from (when making a prediction) or write to (when training the predictor). The patent gives the full details.

So technically this is, IMHO, pretty cool. It uses a little more power than one would like, but solves the multiple branch location problem in a way that I never thought of.

By 2015 Apple have moved on to a proper TAGE-like system, but they add one interesting twist: (2015) https://patents.google.com/patent/US10719327B1 *Branch prediction system*. We described how, with TAGE, we construct a number of indices (in my example nine) which are used to look up in nine tables incorporating each incorporating ever longer runs of history. We also pointed out that the tagging of each entry means that aliasing should be rare. That's good, since it means two different branches won't be fighting each other over whether to increase or decrease the branch direction; but it means the usual problem with direct-mapped caches, ie that only one entry with the index hash can live in the cache at one time. We can fix this with normal caches via going to two- or four-way set associative, and

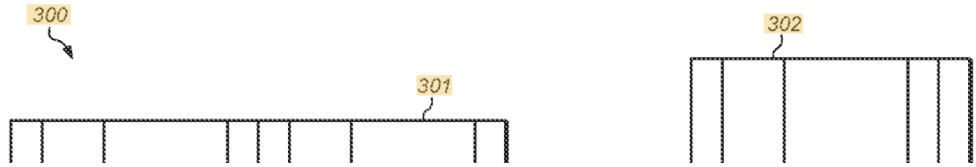
Apple suggest doing the same thing for TAGE. Each of the TAGE tables can be implemented not as a direct-mapped cache but as two- or four-way, with different tables having a different number of ways (and a different number of indices) as best suggested by simulations.

Apple also take interesting advantage of this TAGE structure for energy savings. The lowest power-saving modes stop the clock but continue to power memories (include the branch predictor memories), but more aggressive power saving modes start to cut power to SRAMs and thus lose memory contents. Because TAGE has multiple memories, one can make choices about which of these memories (associated with different history lengths) to cut first. The two main ideas in
(2016) https://patents.google.com/patent/US10223123B1 *Methods for partially saving a branch predictor state* are
- maintain a measure of how many useful values are in each of the tables. Obviously you'd prefer to cut power to tables with fewer useful entries.
- but the tables associated with longer histories take longer to build up, so we need to balance these, but preferentially cut power to the shorter-history tables.
The patent also points out that, once a table has power restored, it's still not useful until it has at least one entry, so it might as well be kept powered off until training deposits one (or more) valid entries into it.

Now let's consider indirect branch prediction. The most trivial solution would be a table indexed by some number of PC address bits, and holding the most recent target of the branch that matched these PC address bits. This captures one common case, where vptr's of some form use a value that changes slowly or never. But it doesn't capture other cases where there is a pattern to the changing value of a vptr, for example the vptr alternates between two different values. To capture this sort of structure once again you want to use a history/path vector for your indexing.
With this background, let's examine the baseline Apple indirect branch predictor (2013) https://patents.google.com/patent/US9311100B2 *Usefulness indication for indirect branch prediction training*.
This iteration of the predictor (as always, a long time ago, and probably replaced by something)like ITTAGE/COTTAGE) has the following form:
- the predictor consists of two tables. The tables are read in parallel. One table is indexed by history data, the other by PC. A hit in the history data table is preferred over a hit in the PC table if both match. (This is a rare case, lots of things would have to alias, and my guess is that the thinking is the history table is much larger, so aliasing [ie false match] is more likely in the smaller table)

- the history table has 1024 entries (indexed by the lowest 10 bits of path data)
- the PC table has 32 entries indexed by bits 5..9 of the PC (not the lowest bits, essentially the cache line index of the PC, for 32B caches lines)
- but the PC table is two way set associative. So each index provides a row with two entries in it. Both are tested, and the matching one (if any) used.
The diagram should make much of this clear.
(Ultimately, though you might not see it at first, this is an implementation of Driesen and Holzle (1998)

http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=FCAA5DE4595332D2186B8E5CDC017A2C?doi=10.1.1.125.5000&rep=rep1&type=pdf *The Cascaded Predictor:*
Economical and Adaptive Branch Target Prediction.



So each entry has the usual validity bit. And a target which is the point of the exercise, telling us where the indirect branch should jump.
More interesting is the tag bit. The tag, as you can see, is bits 10..8 of the PC for one set of entries, and bits 2..10 of the PC for the other set of entries. So the idea for lookup is
- construct the index (one from PC, one from branch history vector)
- lookup the entry
- test if there is a match of the other bits of the PC against the tag.If so we can be reasonably confident that this entry corresponds to a value that was placed here earlier associated with this particular branch (ie chance of aliasing is low).
- if the entry is valid, and tag matches, then we have a prediction.

The other fields are used for training the predictor. You can look at the patent for the details but the approximate idea is that all entries have a usefulness count that's some small number if bits (possibly as low as one, but let's assume its two bits). Usefulness of a new entry begins at 0, but each time the entry hits and predicts successfully its usefulness count goes up. For the PC table, usefulness is used to replace entries, with the less useful of the two entries being thrown out when a new entry is required. For the history-based table it's used to decide if we should replace an existing entry with a new possible entry that matches this index value.
You can puzzle your way through the patent if you like, but to my quick scan it looks like the hysteresis value allows entries in the history-based table to make one mistake while retaining the current target. So imagine an indirect branch that usually goes to A but occasionally goes to B. The predictor will be trained to always give A as the result, and if makes one mistake where the actual target was B, it won't be trained to switch to B right away; it will only switch to B if it makes two successive mistakes.

You can see that this is already fairly sophisticated, trying to capture a variety of common patterns by various mechanisms (the two tables, the fact that one is 2-way, the tags to prevent aliasing, and the hysteresis bit). It's interesting to note that, even at this stage, the size of this predictor storage is ~1000 entries, call them 8 bytes long (if we have a 64B architecture, and are using say 6B of the actual address,

with 2B for tag, usefulness and suchlike). So we're talking ~8kB for the indirect branch predictor, to give one a feel for the size.
At least one item that is, however, missing, is sophisticated use of a very long branch history as in IT-TAGE, but I would assume that by now with the M1 that has been corrected. Compare the 8kB above with the 64kB suggested in the IT-TAGE paper.

Now think of the generic issue. We have Fetch running asynchronously ahead the core, using the Fetch Predictor to generate a stream of Fetch addresses, with this stream being validated (and occasionally corrected) by the Branch Direction and Indirect Branch predictors. If the correction happens via these predictors, it just involves editing the instruction stream before it even hits Decode, so it doesn't cause much pain.
How can we improve this situation? Well, even with the best predictors known to man some branches (especially indirect branches) are simply impossible to predict. Something like an interpreter that dispatches via a procptr is generating an essentially unpredictable stream of of indirect targets. What can we do about this?
In this case the Indirect Branch Predictor can't give a useful branch target prediction, but it can say "this branch is unpredictable". What can we do with that information? What we can do is halt Fetch while the instruction stream proceeds through the core. At some point the indirect branch will be executed by the core and the actual target value will be known. It can be passed back to Fetch, which can resume fetching.

This probably saves some time (whatever we guessed for the branch target is likely wrong, so going ahead blindly would require the cost of a mispredict pipeline flush once the error was detected. But more importantly we save energy -- if execution for a few cycles is unlikely to achieve anything useful, better to wait out those few cycles till we can get back to useful work. As usual there's a patent (2010) https://patents.google.com/patent/US8555040B2 I*ndirect branch target predictor that prevents speculation if mispredict is expected*. The main thing you need is some sort of usefulness indicator for the branch; the patent described the simplest possible usefulness indicator (a non-prediction!) in the context of a rather simpler predictor than the 2013 predictor described above.

Obviously this same idea could be used for hard to predict directional branches (captured, eg, by their having a low confidence), but the payoff for a random direction guess (50% chance the subsequent code execution is useful) is probably worth the gamble.
However just a few hard to predict branches are a real barrier to further progress in ever deeper speculation, as described in a paper we have already seen, (2019) https://arxiv.org/pdf/1906.08170.pdf *Branch Prediction Is Not A Solved Problem*. In principle, with a well-functioning checkpoint and misprediction recovery system, a hard-to-predict branch is not a catastrophe. Mainly what will be lost is the work between incorrect Fetch past the branch, and the resolution of the branch, so ~10..20 cycles of work.

Where this becomes a catastrophe is the pattern of: a branch that depends on a load that misses to DRAM, because now the entire machine state, the entire ROB and a few hundred cycles worth of specula-

tive work will be flushed on mispredict. This fact suggests a few possibilities

- can we detect this particular circumstance (not just a hard to predict branch, but that its resolution will take a long time)? This seems feasible, and if so, should we perhaps halt the machine until the branch is resolved? In the past it was argued that code that proceeded down the wrong path for some time was not as much of a waste as might appear, because the code frequently touched I- and D-addresses that would be needed by the correct path, and so it did useful work by pulling those lines in advance. It's unclear to me, with the most modern I- and D-prefetchers the extent to which this is still valuable.
This option is best if our primary concern is to save energy.

- alternatively, if the pattern in real code is that these problematic branches (difficult to predict, and take a long time to resolve) are sparse in execution time (ie usually only one of them is active at a time), one could just detect these cases and split execution at that point to run down both paths! This is essentially a very simplified version of SMT. One needs to tag the two instruction streams, the registers they touch, the stores they queue up in the store queue, and other paraphernalia, with a one-bit indicator, and one needs a way, when the branch is resolved, to flush all the allocations associated with the other bit. Technically, probably, the trickiest piece would be something I discussed earlier, making sure that all the speculative state that could pollute branch and other predictors, is segregated appropriately and the incorrect half is flushed when the branch is resolved.
I'm unaware of a CPU (or even a paper) that does anything like this, but I have seen it occasionally suggested on the internet. Of course it relies on these problematic branches being sparse; the scheme I am describing does not scale well to multiple successive problematic branches! (The point is not that these are rare; it's that they must not cluster together in time.)
On the other hand, is it worth the effort? Suppose we can store 1000 instructions worth of speculative state. In the baseline case, half the time a problematic load gives us 1000 instructions of progress (until we halt waiting for the load to return from DRAM), half the time it gives us zero. In the SMT-like case, every time we get 500 instructions worth of progress. Same consequence on average.
Is that actually better (even from an energy, not just a performance, viewpoint)?

- what if we could somehow detect the load far in advance and prefetch it? If this were easy, the basic prefetchers would do the job, but maybe (given how rare these problematic branches are) it's possible to build a specialized predictor that's tailored to them? The *Not A Solved Problem* paper did not mention this, but to me it looks like a more promising approach than their suggestions.

- there is a concept called CPU Runahead, eg (2020) https://users.elis.ugent.be/~leeckhou/papers/hpca2020.pdf *Precise Runahead Execution*. The idea is that, under certain circumstances (varying depending on the exact design) the CPU switches to a mode where it tries to explore the future execution stream as rapidly as possible, specifically trying to prefetch as much data as possible. So it drops certain instructions (like FP) that probably won't affect future loads, and may guess at various values (like the values of loads that have missed to DRAM, and will affect future loads).

One could fuse this idea with the earlier SMT idea and imagine a design that, on encountering a problematic load, switches to runahead mode where there's no expectation that 50% of the time these instructions will be retained; rather the (tagged) instruction stream is mangled and partially executed in the expectation that it and all the state it touches, will all be thrown away; but will ideally prefetch a useful amount of material while it executes. This is the sort of choice that probably never makes sense for a battery design, but may well make sense for an AC design.

So we have three immediate options, neither of which is especially appealing! Well, it's a problem for the future, we still have tricks to implement today within the constraints of existing predictors.

Now that we understand the general outlines of Fetch and Branch Prediction, there are a few interesting details to consider. (There are also many less interesting technical details I'll omit!)

One problem with the Fetch Predictor as described is that it can't handle alternating branches well. Consider code that alternately (depending on whether a counter is even or odd) goes down two different paths. This is an easy pattern for a sophisticated history/path based predictor to catch, but what can we do using a simple single-cycle predictor?
If the Fetch Predictor is always representing what we did last time, then it is consistently 100% wrong! We can at least improve this to only 50% wrong by implementing hysteresis, ie some sort of delay, eg we have to see a change from the current prediction twice before we'll insert a replacement. The details of one way of doing this are in (2011) https://patents.google.com/patent/US20130151823A1 *Next fetch predictor training with hysteresis*, but are less important than the idea.
(50% still sounds like not a great success rate!
Fortunately
- if this alternating case is within a loop, it will be caught by a loop predictor and handled in a better way, as we will see when we get to loop predictors, AND
- the more recent Fetch Predictors, at least as of 2016 or so, are tagged not only by PC but also by a few bits of recent branch history. This allows the 2-way set associative Predictor to hold two different Fetch Predictions for the same PC, but different branch history, so trivial alternating cases can also be captured.

However (there's always an however!) now that we have hysteresis, it will take longer to flip a Fetch Predictor element that really should be flipped. I *think* this is the problem our next patent is trying to solve, but this one is written at such an abstract level, with no helpful examples, that I can't be sure. I think the idea with (2017) https://patents.google.com/patent/US10613867B1 *Suppressing pipeline redirection indications* is: suppose we have a situation of a very tight loop containing in the loop body a conditional branch. And suppose that the large TAGE predictor detects that its (high quality) prediction doesn't match the Fetch prediction. What to do?
The hysteresis answer, essentially, was redirect immediately, and suppress the Fetch Predictor update for this iteration.

This 2017 patent's answer is, suppress the redirection and temporarily delay the Fetch Predictor update. The idea, I think, is for very short loops (ie every cycle is a new pass through the loop body) allow the temporary Redirection Circuit to make a decision about how best to train the Fetch predictor for this branch (is is alternating? is it a permanent change to the previous case?) By suppressing the redirection yes, we get Fetch looping once or twice pulling in bad data each time, but we can flush that in time. Meanwhile we manage to run the loop through TAGE and Training a little faster than if we had redirected the loop in response to the bad prediction, so we're faster in training the Fetch Predictor to match to the ongoing loop.
To justify this somewhat, consider that Apple suggest the time from start to branch predictor output value is around 5 cycles. This means five cycles are lost on each round of redirect and restart; as opposed to just one or two extra cycles being lost if we hold off on redirect for one or two cycles once we have noticed a mismatch, to figure out an optimal value setting for the new Fetch Predictor entry. But hey, if you can understand the patent and provide me with a better explanation, be my guest!

The Return Address Stack (RAS) seems sufficiently obvious that there's nothing much to patent in the main idea, but this changes when you start to think about it!

Here's are some issues that you ignore at first, then realize are important:

- Fetch prediction, as we have described it, won't work for returns. Using the mechanism described, we can do an adequate job, that will usually pull in the correct fetch stream, for conditional branches and for function calls, but not for returns. If returns are treated as just a "follow this run of instruction by going to the same address as you did last time" they will frequently be mispredicted.
This means we need three things
+ a marker in Fetch Predictor entries that says "the jump that ends this Fetch Group is a return so treat it appropriately"
+ a miniRAS for the Fetch Predictor that's not necessarily too large or too fancy, but which usually is able to push function return addresses when a Fetch Group begins with a function call, and pop on predicted return.
+ which in turn tells us we also need a marker in the fetch group saying indicating when a function return address needs to be pushed

This means in turn that we will have at least
- a MiniRAS being used by Fetch Prediction
- an "Execution" RAS being used by the proper Return Address Predictor. This one will be larger, and will try to keep things accurate in the face of various mistakes.
- the absolute truth RAS which is what is stored on the physical, in DRAM, stack at any given time, and which will be resorted to in the event that things go hopelessly wrong and the predictor makes no sense (for example when we revert to a thread after having context switched to a different thread).
The MiniRAS is not just smaller than the Execution RAS, at any given time it holds different contents.

For example the MiniRAS may push, then pop, a return address over two successive cycles, before the PC's associated with those two Fetch Groups have even been processed by the Execution RAS.

Next, the big obvious problem to be solved with a RAS is of the stack becoming corrupted under mis-speculation. Suppose, for example, that the speculation path we travel down at some point involves a function call (push an address on the RAS), but we discover the misspeculation and redirect Fetch to the correct spot, leaving that inappropriate return address on the RAS.
The MiniRAS is small enough and lighweight enough that I suspect it makes no attempt at correction. Suppose the Return Address Predictor notices a mismatch between Fetch has done and what the Execution RAS says. Best response is, at the same time that Fetch is being redirected to the correct address, to copy over the top few (presumably correct!) entries from the Execution RAS to the MiniRAS.

What about the Execution RAS? As usual there is a range of ever more complicated options.
To get started, think about what a stack means as HW implementation. Conceptually a stack is a (for now indefinitely long) array of storage, along with a number which we call ToS (Top of Stack) that tells you the top of the stack. What makes it a theoretical stack is that ToS can only be incremented or decremented by 1.
First level of correction is to ignore that rule. Maintain (somewhere...) the value of ToS. Then, in the scenario described what happens is
- we know the value of ToS pointing to the correct point in the array at the point before misspeculation
- the incorrect function call dumped an extra address on the stack, and incremented ToS but
- recovery means reverting ToS to the value before the mispredicted path that led to the function call.
This means that we need to have one piece of storage associated with the stack called ToS, along with the recovery ToS recorded at every point we might need to revert to. At least approximately (we'll figure out details later) let's assume something like
-ToS can change (correctly or incorrectly) only at the execution of calls and returns
- so in principle we could store just at each call and return what the ToS value was at that point
- then on mispredict recovery we'd run through the ROB looking for the first call/return older than the recovery point to read its ROB
- that works, but seems suboptimal. Presumably we also need to to store the appropriate value of ToS in Checkpoints, and maybe we want some sort of structure associated with the ROB that is dedicated to this job of holding ToS, so that we can find the correct recovery version faster than via a sequential search backwards.

OK, so we have figured out, by logic, that we need a ToS value for the stack, a recovery ToS value associated with each call/return, and somewhere to store these recovery values that's, hopefully, fairly easy to search.

Next complication is that the recovery scheme I described above only fixes half the problem. What I described was the case that
- we mispredict a call,

- the call pushes an inappropriate return address on the stack , increments ToS) (and saves recovery-ToS)
- but it's fine because we recover by finding recoveryToS and setting ToS to that correct value

Consider the reverse case
- we mispredict a return
- the return pops a value off the stack
- we continue down this bad path and mispredict a call
- the call will *overwrite* the storage slot that was being used by the return address

Our games with recoveryToS will not help us now. We can revert ToS to pointing to the correct storage slot, but the value in that slot is invalid and the correct value is gone!

With this understanding, you can now look at Skadron (1998) https://mrmgroup.cs.princeton.edu/papers/micro31_retstack.pdf *Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms,* which describes what I said in more detail.

So we have the problem that we also (in some fashion) want to save the return address, recoveryAddress, at the top of the stack.
Skadron's solution to this is to just save that return value in the same place as wherever we are storing the recoveryTOS.
In principle you don't have to store the recoveryTOS and recoveryAddress for every branch, because it only changes at call/return, so you can use some indirection to store these values in one structure (associated with recent call/return) and have a short index into that structure associated with every recent branch.
Intel implemented, for the Penryn generation, a complicated scheme(based on (1997) http://esca.korea.ac.kr/teaching/com609_TESII/RAS/Recovery-Branch-Misprediction-1997.pdf *Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution*, that implemented a stack via a linked list and allowed pushing new values on the stack to "bypass" rather than overwrite older values, so that those older return values were still available on the stack if required.

I haven't seen anything that explicitly covers how Apple handle this, but their RAS-related patents suggest that the structure is extremely reliable, so it must be utilizing some recovery scheme. The main BIT patent (to be covered in a few pages) refers to some fields that are suerly related to RAS maintenance and recovery, but which are not explained.

We can deal with this by converting the stack to a "stack-like" structure that never overwrites data. This is fairly complicated, so make sure yo understand each step before moving to the next step.

- Consider a standard doubly-linked-list. Call the pointer to the head of the list ToS. It should be clear that you can implement a stack by obvious easy operations on this doubly-linked list. (ie think what it

means to either pop an entry from this stack/list, or push an entry onto this stack/list).
The nodes of this list look something like (`return value; previous; next`) where previous and next are pointers.

- Now, consider an array of these nodes. Now we can make previous and next indices into this array, not generic pointers; and ToS is likewise an index into this array.
Once again, work out in your head what it looks like when you push or pop this stack.

- Now we will make this a “one-time-write” stack. Along with ToS, we add a second index (associated with the array), called Alloc. Alloc, like ToS, starts at 0, but Alloc has the property that it can only ever increase, never decrease.
So we push the first value onto the stack. This value is placed at 0, and both ToS and Alloc increament to 1. new entry looks like (val0; null; 1) at 0
We push a second value onto the stack. This value is placed at 1, and both ToS and Alloc increment to 2.
new entry looks like (val1; 0; 2) at 1
We pop the stack. ToS reverts to 1, but Alloc stays at 2.
We push a third value on the stack. We will need to write, so Alloc increments and ToS is set to Alloc=3.
new entry looks like (val2; 0; 3) at 2
And so it goes. Note that
- this still behaves like a stack. We know how to push, we know how to pop, just follow the linked list links.
- but older values in the stack are never overwritten, they are just snipped out of the linked list.
This has two consequences:
+ The previous scheme we described of storing a recoveryToS with each call/return will now still work! The value at the recoveryToS is still valid, and the links from it backward down the stack are still valid. (Forward links are a random mess, but who cares; they represent state from the invalid path).
+ I’ve described this in terms of unlimited storage for the stack or the linked list. As far as stacks go, the “frequently accessed” depth of most code is, I don’t know, probably covered just fine by 64 entries. Some code will exceed this (leading to mispredicts and generating some slowdown) but the real issue is with a budget of 64 entries, how long will you go between either under or overflow and encountering mispredict?
(I have described this via a doubly-linked list because I think that’s easier to visualize and imagine in your head. But once you understand how it works, you will see that there are no conditions where you actually need the next point, only the previous pointer; so you can remove the next pointer and its updating.)

But with this linked-list scheme I described, we use up one element of our storage every time we call a function, not just every time we increase the depth of our function calls. This is very different scaling! So we will use up any practical array for our linked list fairly rapidly.
There are multiple ways one can imagine for solving this, but here’s one that somewhat matches what Intel does, simplified and ignoring unimportant details:
- maintain both the stack RAS and the linked-list RAS.

- entries in the linked list each have a color which tracks wraparound. So entries start off red, after we wraparound the end of the list, entries we write are green, then with a second wraparound back to red.
- each time your store a ToS, store both the stack value and the linked-list value, and the color of that value.
- this scheme (or something equivalent) allows you to track when you have overwritten a value in the linked list ToS
- when you need to recover from a mispredict, first look at the linked list recoveryToS. If the color matches the recovery color (ie that value has not been overwritten) we can trust the linked list RAS, so we use that stack; otherwise we use the value from the stack RAS.

The Intel scheme is described in (2008) https://citeseerx.ist.psu.edu/viewdoc/download?-doi=10.1.1.217.155&rep=rep1&type=pdf *Improvements in the Intels Coret2 Penryn Processor Family Architecture and Microarchitecture*. If you want to know the full details, look at (2001) https://patents.-google.com/patent/US20030120906A1 *Return address stack*.

Before we discuss what Apple does, let's return to an issue we skimmed over. We need somewhere to store the the recoveryToS for each call and return.
I described doing so in the RoB slot, but that matches a pattern we have seen frequently before – don't make a general structure larger, rather use indirection to store data in a task appropriate structure.
The issue we have seen here is one that is actually more general: every branch, until it retires, needs to hold onto some context information. We have seen this context information for calls and returns as being recoveryToS values. But for other branches, we will want to store a bunch of data related to the state of the machine at the point of the branch, things like the history/path vector, and the target of the branch. These may seem no longer necessary
- we predicted the branch long ago at Fetch. Correct or not, the prediction is done
- we compared the prediction to the appropriate value at branch Execution. Again, right or wrong, that's over. All the ROB needs to know is if we mispredicted, and if so where to redirect Fetch.
No! You are forgetting that we also need to train/maintain the Predictors. And what have I kept saying? We don't want to train them on invalid data!
So we want an additional structure (almost like the equivalent of the History File or Register File) for branches, that holds data associated with each branch through approximately the period from when the branch is Decoded to when the branch is Retired. This data structure is called the Branch Information Table. Think of it like an extension of the ROB, but just for branches; in actual implementation I assume branches in the ROB have an index that points into the Branch Information Table (BIT).
We will get into details but, approximately, there are two large subtables, the BIT and the TBIT (Taken Branch Information Table) each with subsections for different types of branch (eg conditional vs call/return). Entries in the BIT/TBIT are the sort of stuff described –what you need to maintain the prediction machinery. So the call/return slots will, among other things, store ToS values; and the conditional branches will, among other things, store their history/path vectors and targets, whatever is required to train the Predictors. On Retirement these fields will be copied out of the BIT/TBIT and sent to to predictors to be handled appropriately.

Apple's patent is (2011) https://patents.google.com/patent/US9354886B2 *Maintaining the integrity of an execution return address stack*. This patent is extremely confusing until you already understand what it is trying to say. You have to read it with the Intel patent in mind, and even so still read between the lines. This is my best attempt.
As terminology what I am calling the MiniRAS, they call the Speculative RAS. What they call the Execution RAS is also not a great name. It is associated with Execution, yes, but it is still essentially a speculative structure :-(They mention as an aside that they only use these two structures, which may seem obvious except that this is in reaction to Intel, who use about five of these RAS structures! So Apple have one for Fetch, and another that's modified at Execution. In particular the Execution RAS (implemented in the linked list form) has to handle overflow differently from Intel, it can't, as Intel does, either drop back to the stack version occasionally.

Apple tell us one way they prevent their Execution RAS from being polluted, beyond everything we have already covered. The problem (I think) that they are trying to deal with is Replay, though the patent explanation is terrible. We have described how instructions can be scheduled speculatively, on the assumption that the load they depend on will succeed, but if fails they will be re-executed. This means that all execution has to be *idempotent*, it must not ultimately matter if a given instruction is executed one or twice or three times; all that matters is that the final execution does the right thing and everything done by the prior executions is overwritten.
We have followed that in great detail in terms of standard integer (or FP) instructions, and how they can be re-executed multiple times because their values are always written to the same physical register, so the final execution after multiple replays will overwrite whatever was incorrectly written earlier to that register.
Now you might not think that there could be any dependencies between a branch and a load, but there is, in the form of either `RET` or `BR Xn`, the latter being an indirect call to a function pointed to by Xn, the former being a return which is an indirect branch through register X30. In both cases register Xn or X30 could be filled in by a load, or a calculation dependent on a load. Thus if the load replays, the RET or indirect branch will replay. Will that execution be idempotent? No, not by itself, because it will result in a second push or pop onto the Execution RAS. Hold that thought, while we rephrase what we said in different words (to make the point), and read below while you look at the diagrams in the patent.

+ Each entry in the BIT is a separate branch execution. It may be the same branch as far as its address in the cache is concerned (think of the branch at the end of a loop that is called repeatedly to test loop exit; every execution *instance* of that branch will have an entry in the BIT).
+ The Index in the BIT is some way to differentiate and select a particular instance in the BIT. The best way to think of it is that it's the same number as the ROB slot that the branch occupies, so we have a connection between a given ROB slot and a given BIT slot.
+ The entry in the BIT also has one more bit, the "first time" bit. What's that? Didn't I just spend ten sentences saying that each BIT entry corresponded to a unique execution of a branch? Yes but language is difficult! This is the bit that handles Replay.

The basic flow is
-- at Decode time, when we allocate ROB slots, we also create an entry in the BIT and mark it as “first time”.
-- Things continue well, we execute the branch, we clear the “first time” flag.
-- Ideally that’s the end of the story. But maybe there’s a Replay of the load, which feeds into Replay of dependent instructions.
-- Do the branch is executed again. It recalculates the address correctly (ie looks at register X30/xn which, presumably, now has the correct value from the load) *but* it does the correct thing with the RAS.
For a push (ie an indirect function call) it *replaces* what it put on the stack last time (a junk address corresponding to whatever random junk was in xn since the load failed).
For a pop (ie a return) it does nothing because the the pop executed the first time and ToS has been moved down to its correct location.

So basically we ensure, under very technical circumstances (Replay of a load that’s feeding a return or indirect call) the Execution RAS maintains integrity.

The second question of interest is how does Apple deal with the constant growth of the Execution RAS? Remember that when we left Intel, we were at the point that
- we understood the reason why you want a one-time-write stack (implemented as a linked list)
- but that implementation has the problem that it uses up new entries in the storage for every call (as it must, that’s the whole point!)
- so we have this somewhat intricate business of wrapping around the storage buffer, and flipping color, to try to reuse storage with as little damage as possible.
Here’s my guess as to what Apple does.
The Intel version is creating a stack via a linked list, but takes the stack aspect more seriously than the linked list aspect. Forget that, treat the structure as a pure linked list implemented with a finite array of node-sized entries (say 90 or so). Each entry has a “valid” bit; also forget the Alloc index.
Now every time an entry needs to be allocated we can ahead down the buffer (with wraparound at the end of the buffer) looking for an invalid entry, which we use. This means we will soon have entries all over place looking like a random linked list; but the stack structure will be there when we follow the pointers.
The above constantly allocates new entries. When do we free entries? Well, the reason we don’t want to overwrite an entry is that it may correspond to the return address of a `RET` that seemed not to matter while on a speculative path, but ultimately did matter. In other words, when a `RET` Retires, we can be absolutely certain that the slot in the RAS holding its return address is no longer relevant to anything; so when a `RET` Retires we can mark its slots invalid. This will now give us a process that is freeing slots as rapidly as we are filling in new slots and overflow is no longer an issue.
If this sounds familiar, it should remind you of how we find free registers in the register file, as discussed earlier...
(There is still the issue of what if you go really deep down a set of nested calls? At that point you have to just cull the oldest value in the stack, and accept the cost when you finally pop the stack; just like with

the very traditional RAS that began this discussion. I have hypotheses as to how this might be done – eg how to track what counts as the oldest value – but we're already on such a limb as to this proposed implementation that further discussion makes no sense. Hopefully future patents will come to light explaining how Apple deals with the various types of RAS over/underflow).

One final side issue in the patent is they point out the obvious fact that if the front-end RAS (what I am calling the MiniRAS, what they call SRAS) is corrupted, you can recover adequately by copying values from the Execution RAS to the SRAS. Intel say the same thing in their patent, but they seem to suggest doing it on demand, one entry at a time; Apple do it wholesale whole stack at once. I guess that's different enough for a patent. Don't take this part too seriously because we move on to

(2013) https://patents.google.com/patent/US9405544B2 *Next fetch predictor return address stack*, which covers the miniRAS in more detail.
This patents (two years after the previous RAS patent) clarifies and improves much of the machinery.

The first nice improvement is that the main RAS (what was previously called the Execution RAS) is now maintained at Decode. Logically this makes much more sense. Decode is where BIT entries are allocated and, if you think about it, all you need to maintain the RAS is knowing that an instruction is a call or a return, which you know at Decode time. So Decode can either pop the RAS (return) or allocate a RAS slot, and we no longer have the worry about idempotent executions and ensuring that Replay of call or return pops or pushes the stack too often.

Secondly we have confirmed that some level of predecode is being done. Suppose that Fetch accesses lines in the I-cache that it has not seen recently, and so are not part of the Fetch Predictor. From the predecode bits, it will know that the Fetch Group it is sending downstream includes a call or a return, and it can behave appropriately – a return means generate the next Fetch address from the MiniRAS, a call means, as we saw, pause Fetch'ing until upstream can inform us as to the predicted target of the call. (Note that Decode can accurately fill in the return value of a call, even if it does not know where the call will be directed, eg an indirect call!)

The final improvement is that the MiniRAS is no longer really a full separate stack running in parallel with the primary RAS. The MiniRAS is only required if there is a return following a very recent call, so that that the call has not yet propagated to the primary RAS, so it can be very much smaller than the primary RAS. What's needed is to track how many calls and returns have been encountered as part of a Fetch Group, incrementing these at Fetch and decrementing at Decode, so that you know whether you should pop a return value from the MiniRAS, or use the value at the top of the main RAS.

I'm not much interested in security, but if you are, you might want to look at (2015) https://patents.google.com/patent/US10867031B2 *Marking valid return targets*. The idea is to foil ROP by tagging calls and returns in such a way that a faked return (by placing an address on the physical DRAM stack and forcing a return to execute to that address) will not have a matching tag at the point where it returns.

Mostly this is its own thing, the one interesting part as far as we are concerned is that it uses the RAS prediction mechanism to know when to check tags. Under normal control flow, the control flow happens via the RAS and MiniRAS which have not been attacked; the first indication of an attack is when the executed return does not match the predicted return address. So it's only under those conditions that the CPU toggles to paranoid mode, where it checks for a mismatch between the call site and the return site.

I'm not sure if the above idea was ever implemented in HW. It is followed by (2016) https://patents.google.com/patent/US10409600B1 *Return-oriented programming (ROP)/jump oriented programming (JOP) attack protection*, which should look familiar; this is the PAC (Pointer Authentication) stuff that encodes the 2015 tag in a few high bits of the return address; so now it's pretty much always there, and always checked as part of logic that reads and writes the RAS.

The third patent in this space is (2018) https://patents.google.com/patent/US20200192673A1 *Indirect branch predictor security protection*. This builds on ideas we have seen before. We've seen that the branch predictor (in whatever form) tags each entry in its prediction table so that, even after the index hash that looked up this table is probably unique, we still compare the tag to a different hash of the PC and/or the path to get a stronger indication that we're dealing with the correct entry, not an aliased entry. The idea of the security patent is to augment that tag with a security tag which holds, among other things, the protection level, the process ID, the VM ID, and some of the high bits of the PC. This security tag is compared with the state of the machine, and if the two don't match, the prediction is not used.
This means that an attacker cannot seed the predictor so as to get the attacked code to speculatively go down certain paths for a few cycles before the speculation is discovered (ie the content of the SPECTRE exploit). Each entry in the tag prevents certain types of attacks, for example the process identifier prevents one app doing this against another app.
To my eye, the most interesting one is using some high bits of the PC. Normally one ignores the high bits of the PC in any sort of indexing or tagging because they carry so little information (ie code tends to hang around a limited region of the address space). But suppose we have JIT'd (ie untrusted) code that wants to attack the host process. As long as we place the JIT'd code somewhere that has different high PC bits from the host code (eg reserve the highest 1/8th of user process address space for JIT'd code), then storing a few of the high PC bits in the security tag is enough to prevent the JIT'd code from being able to create branch predictor entries that will affect the host code.
(Note that this patent also answers the question we raised about reusing branch prediction data across context switches. While such reuse is probably theoretically the optimal choice, security forces us down to the intermediate extreme of tagging branch data by processID.)

Now consider branches, direct calls, and even the `TBZ` and `CBZ` branches that test a register. All of these have the property that the target consists of an offset added to the PC. Suppose that when the

branch is predecoded (ie when it is loaded into the I1 caches from I2) we calculate the target address there and then and store that in the I cache. This would save the latency of the addition, and the energy cost of many future address target additions and seems like a good idea. It is a good idea except there is one tricky detail. Consider the following set of issues:
- we have a line from a shared library, which includes a branch located at virtual address vA and physical address P.
- we predecode that line, and replace the the branch target address with an address tA which equals vA plus some offset.
- now we context switch to a second app which uses the same shared library, but aliases its placement so that the branch is located at virtual address vb (but still at the same physical address P). If this doesn't sound familiar, you need to go read about Position Independent Code, and Address Space Layout Randomization.
- this means that app B will see the cache line already in the I1 cache (because that cache is physically address), but it will see the target of the branch as tA (ie vA+offset) whereas it should see the target as (vB+offset).
Oh dear!

But the idea can be saved, somewhat. Suppose that rather than calculating the entire target address we calculate only the lowest fourteen bits. This will give us the page offset of the branch which is always the same. Then we actually use the branch we only have to sum the upper bits (a shorter sum, so faster and lower power). It actually gets even better because there are some paths for which we may want to look up the branch target in a predictor or cache, and if that predictor/cache is indexed by just the lowest 14 bits of an address (or that mixed in some way with other data) then we can perform the first stage of the lookup immediately, in parallel with the sum to find the page number, then compare the page number with the tag of the indexed lookup. (This should sound just like an L1D cache lookup, because it's essentially the exact same trick!)
This is the content of (2014) https://patents.google.com/patent/US9940262B2 *Immediate branch recode that handles aliasing*.

Back to Fetch Predictor

Now that we've seen two interesting things predecode can do (mark branches of various types, compute branch target low bits), how much more can it do?
We've already seen (2014) https://patents.google.com/patent/US20160011875A1 *Undefined instruction recoding* suggesting a not very interesting option – detect all the possible *undefined instruction* encodings in an instruction and replace them with a single "undefined instruction" value.
But once you see the idea, many possibilities open up! Suppose we are willing to widen the "in-cache" instructions to 40 or even 64 bits wide. (Since all instructions have the same size, this does not complicate branching and addressing; with Thumb it might have been more painful.) While AArch64 has a pretty regular instruction encoding, the necessity to fit into 32 bits means some irregularity in the

encoding. One could imagine various small cleanups like ensuring registers are always in exactly the same locations, all immediates have the same value, etc. Of course what's worth doing depends on details we don't know about the pain points in Apple's A64 decoder.

Another obvious case is mapping various types of NOPs down to a single canonical NOP.

One thing that is likely is that xzr is removed from many instructions. For example it's cute that the `MUL` instruction is simply multiply-add with xzr being added, but for anything beyond the simplest CPU, implementing `MUL` literally in this way is throwing away performance. I could imagine pre-decode also mapping many of these "use xzr to provide a simplified ISA" techniques to what are literally simpler (lower energy, one cycle less) opcodes. We've seen that xzr behaves differently for some situations, like MOV, and maybe this is a casualty of such pre-decode?

Another interesting direction for this I could imagine is to move the pre-decode unit up to the L2. Most cycles it's unused, and it seems having a single pre-decoder shared across all cores in a cluster is a better use of resources, and justifies expanding the pre-decoder to handle more cases.

Now we get to the more specialist patents.
(2013) https://patents.google.com/patent/US10901484B2 *Fetch predition* [sic] *circuit for reducing power consumption in a processor*. Many of the precise details in this make no sense to me (they may be obsolete relative to the M1 details), but the general idea is clear enough: since the Fetch Predictor contains information about whether the next Fetch Group contains (perhaps untaken) branches, indirect calls, and returns, each of the second level structures used to validate these predictions can be powered by (probably via clock-gating) for the next cycle or so if it will not be required immediately.
In terms of details, I think what they are trying to say is that
- the Fetch Predictor array is split into two pieces. They are both indexed by the current PC, but one piece holds Fetch Groups that terminate with a target (ie a taken branch of some sort), the second piece holds Fetch Groups that run on to the next Fetch Group.
- this split obviously means some saved storage space (the second piece doesn’t require a Target address); but it has an additional implication: the sequential Fetch Group piece can indicate not only what (presumably all non-taken conditional!) branches are present in the Fetch Group, but what branches will be present in the Fetch Group that it flows into. This means that we can know not just what predictors to power down for one cycle, but even for two or three cycles, and that may allow for some additional energy saving.
(We’ve stated repeatedly branches are dense in code, and that’s true. But it’s also true that some loops, especially in FP, are unrolled to become a single long loop say 100 instructions or so long without a branch. I think Apple is trying to take advantage of this sort of thing whenever it’s encountered, both in the space savings in the Fetch Predictor, and in the energy savings of allowing the branch predictors to sleep for multiple cycles.)

(2016) https://patents.google.com/patent/US10203959B1 *Subroutine power optimiztion* [sic] builds on

this above idea by noting that a second common and very stable pattern is of a Fetch Group that terminates in an unconditional call. In that case, just like the previous sequential prediction case, we can make useful predictions about what branch predictors will be utilized over the next two (and perhaps a little further) cycles.
A second small tweak is: suppose a new cache line is pulled into the I-cache. This cache line will be pre-decoded before it hits I1 and the rest of the core, and that pre-decode will note the presence of branches. This information can be conveyed to Fetch which can, once again, use to it determine whether branch predictors need to be powered up or not. (An obvious case is if the line includes no branches, or no indirect branches; a less obvious case is if the first branch is unconditional so no prediction is needed, we know it will be taken, and this will be handled when the branch hits Decode.)
A third way to exploit these new-to-the-cache lines combines the above two; suppose that the first instruction in the line is a call, followed later by an unconditional branch of some sort. Then, even though the Fetch Predictor may not know this line (it's new to the cache) we can know, and store away, that when the call returns to this line, the number of instructions to load from I-cache should not be the rest of the line, but just the instructions up to the unconditional branch. (This may seem like a rare situation, but remember return, or calling a second subroutine, are both unconditional branches...)

Getting back to the 2014 patent, given that you are packing the Fetch Predictor with various energy-saving data, you can go further! For example you can pack both way information and ITLB-translation information into the Fetch Predictor, meaning that now, as long as we are successfully running out of the Fetch Predictor, both the ITLB and tags comparison of the I1 cache can be slept. Very nice!
(This facet of the patent builds on (2010) https://patents.google.com/patent/US8914580B2 *Reducing cache power consumption for sequential accesses*, which is a much less ambitious version of the idea; essentially suppressing the ITLB and tags lookup in the case where sequential Fetch reads instructions from the same I-cache line as was accessed in the previous cycle; also upon (2013) https://patents.google.com/patent/US9311098B2 *Mechanism for reducing cache power consumption using cache way prediction*, which applies the same sort of idea to sequence the way activation in the correct order when following straight line code or known branches.)

I'm going to out on a limb here and suggest that the Fetch Predictor is defined by "slices". The slice concept seems to be a common feature across many parts of an Apple CPU. The easiest example, given what we have seen so far, is to consider the ROB. We described the ROB as consisting of multiple "rows" where each row can hold up to six "simple" instructions, and one "failable" instruction. If you think of the ROB as a two dimensional table, it consists of multiple rows, and a row consists of six entries of type A, and a seventh entry of type B. The columns, in this structure, are, I think, what Apple calls slices. So for the ROB there one slice (or perhaps six slices?) that hold "simple" instructions, and an additional slice for the "failable" instructions.
The advantage of this setup is that you get to share some of the indexing and control across all slices of the structure; the disadvantage is that the number of items in one slice is tightly linked to (a multiple of) the items in another slice.

So for the Fetch Predictor, I think we have this sort of slice structure. Each row in the Fetch Predictor has
- a first half that holds “Fetch Groups that will end in a branch”, so that entry includes a field for “target address” and another field for “number of instructions to Fetch”;
- the second half of the row holds a field for “something about the branch structure in the next fetch group or two or three”, but doesn’t need either the target address field or the fetch width field.
Remember both halves have a separate tag, so even though they appear in the same row, they are unrelated; they just happen to both have the same lower bits in the PC; there is no implication that once a Fetch has used the upper (or lower) half of a particular row in the predictor table, it switches to using the other half of that row.

We will see yet another version of this slice structure when we look at the Branch Information Tables in detail.

To try to summarize some of the above.
We described the idea of a Next Fetch Predictor, and showed
- how it could be bootstrapped from nothing by being continually retrained as its predictions (starting with “always go forward”) were corrected one after the other.
- how the sooner these corrections were provided (so, if possible, in Decode rather than at Execute or, even worse, Retire) the better

Even so, it takes up time and a constant stream of mistakes to populate the Next Fetch Predictor.
Can we improve this? That’s what (2016) https://patents.google.com/patent/US10747539B1 *Scan-on-fill next fetch target prediction* does.
Suppose that the Next Fetch Predictor jumps to some piece of code that it has never encountered before. This cache line, once it is pulled into the Fetch unit, will be scanned to construct a row in the “Scan on Fill” predictor. Essentially this predictor holds, for a few cache lines, some details that are relevant to Fetch, like where the branches are, the type of branch, and (if possible) the branch target.
Now next time a Next Fetch Address is looked up, the lookup will occur in both the Next Fetch Predictor and in this Scan on Fill Predictor. If there’s a hit in the Scan on Fill Predictor but not in the standard Next Fetch Predictor, then the Scan on Fill Predictor data will be used. This data will make its way to the standard Next Fetch Predictor and eventually this line of the Scan on Fill Predictor can be removed.

Even this, as I have described it, is slightly reactive. So the Scan on Fill Predictor takes one extra step. Suppose a line is moved into the I-cache but not by Fetch (ie the prefetcher has pushed it into the I-cache). Then when the I-cache has a cycle free, the Scan on Fill Predictor pulls in that line and builds an entry for it. Assuming prefetch is working optimally, this means that by the time Fetch will be ready to generate an access to the line, not only will the line be in cache, but it will also be sliced up by the Scan on Fill Predictor into Fetch Group units.
This mechanism can obviously handle, as embedded in a newly encountered cache line

- direct calls
- returns
- indirect calls (by the “Halt Fetch when it encounters and unknown indirect call, until the target is resolved” method we’ve already seen).
- what about conditional branches? The patent does not tell us. It suggests these are always either considered taken or not taken. Another alternative might be to probe the Branch Predictor to see if it has a suggestion. A full probe might takes too long to be feasible, but what about a simple probe of the base Branch Direction Prediction table, the one that’s just a local 2-bit counter?

The most recent version of the NFP is described in (2017) https://patents.google.com/patent/US10445102B1 *Next fetch prediction return table*.
The following strands are unified:

- entries are defined by a few different “strength” bits.
One of these is a hysteresis bit which tells the system to hold onto the entry even in the face of an early misprediction.
A second such is confidence bits that indicate the target address of the entry (ie where to jump next at the end of this Fetch Group) is trustworthy. If this goes down, we may retain the entry but change the target (eg a virtual call that ends the Fetch Group has changed its target).
Another way to think of this (different from the way Apple describes it, but I think more helpful) is to consider that we have say a 2-bit confidence field, but the field is initialized not at 0 but some higher value. Depending on the initial value we set (1, 2, 3) the entry gets up to three chances at early misprediction before it’s considered hopeless and is a prime candidate for replacement.

- now that we are tagging entries not just by the PC but also by a few history bits (as we said, now allowing for a few variant paths to this Fetch Group) two-way set associative is not always enough. Rather than accepting this, we define an “overflow table”. If both entries for a particular index in the primary Next Fetch Predictor table are valid and have high strength, then we allocate this additional entry into this overflow table (so essentially a victim cache, though for reasons I cannot see, the patent calls this the NFP Fast Table).

- additionally we realize that any NFP entry that terminates in a Return is wasting the storage used for the Target (the PC to go to after this Fetch Group is loaded). Hence those entries are moved to a separate table called the NFP Return table, relieving some pressure on the primary NFP Table.
(This table can also omit the Fetch Width value because that can’t be exploited – returns to different locations will result in the next Fetch Group having different widths, so best one can do is pull in everything from the cache line and splice it out as appropriate once it reaches the Instruction Queue. This seems like another area amenable to a small tweak, like calculating the post-return width at call time, and storing it on the RAS. This is expensive to do dynamically, but could be done at the time that a Fetch Group that terminates in a call is inserted into the NFP. A second reason to have a distinct table or two for NFP entries that terminate in calls, as I suggest below – such tables could have an additional “post-return fetch width” field...)
This separate table for Return-terminated Fetch Groups can be appropriately sized (and probably created as direct-mapped rather than 2-way associative).

- So in any given cycle, what we will see is the PC presented to
+ the primary NFP (two-way set associative, also matching recent history)
+ the Return NFP (probably direct-mapped, no history tagging)
+ the victim cache Fast NFP (FIFO of recent overflow entries)
+ the Scan on Fill Predictor (some sort of cache)
Each of these will try to provide a best guess as to the address of the next Fetch Group.
One could imagine further subdivision over time. For example, perhaps direct calls usually don't need the history tag and could be moved to their own, direct-mapped, table with a smaller tag? Likewise for indirect calls, only their table is 2- or 4-way mapped, with longer history tags. Not trying to compete with ITTAGE, but to at least pick up the lowest lying fruit of virtual calls with an easy prediction pattern...

Now let's talk Loop Buffers.
It should be clear from what we've discussed, that there's no real *performance* need for a Loop Buffer on the M1. A Loop Buffer (and related concepts) is a way to avoid paying the cost of the backward branch of a loop; but M1 already has that cost down to zero. Even so M1 incorporates a variety of loop buffers, mainly, but not exclusively to save energy. The reason the different variants exist is because the most aggressive energy-saving techniques only work with the simplest loops, but one doesn't have to give up on more complex loops, one just has to accept a lower energy reduction.

We start with the basic idea, to use a loop buffer for "easy" loops. The characteristic of an easy loop is that there is no variant control flow within the loop body – we enter at the top, we loop back at the bottom. In the strictest version we could insist on no branches in the body.
Such a loop (if it's short enough, which it usually is) can be captured within a buffer in the Fetch Unit right next to Decode. We can run the loop out of the buffer and avoid the energy costs of branch prediction and the I-cache.
This is covered by (2012) https://patents.google.com/patent/US9557999B2 *Loop buffer learning*, and the main trickiness is detecting such a loop (and recording that other potential such loops should be ignored). However we also see a few additional optimizations:
- The loop buffer lives behind Decode. This means the buffer holds uops rather than ops, and the cost of Decode can also be avoided!
- The loop buffer does allow for a limited number of forward taken conditional branches (ie simple `if(condition){}` clauses within a loop), but no indirect branches. The patent is somewhat ambiguous as to whether simple call+return would be allowed.
The loop body has to be *invariant*, ie the same from iteration to iteration, so conditional loops are allowed – but only if they don't change their taken/not taken status.
The number of branches allowed is surprisingly generous, 8 are suggested in the patent. In other words, even at this simple stage we're capturing loops a lot more sophisticated than a basic blitter or strcmp.

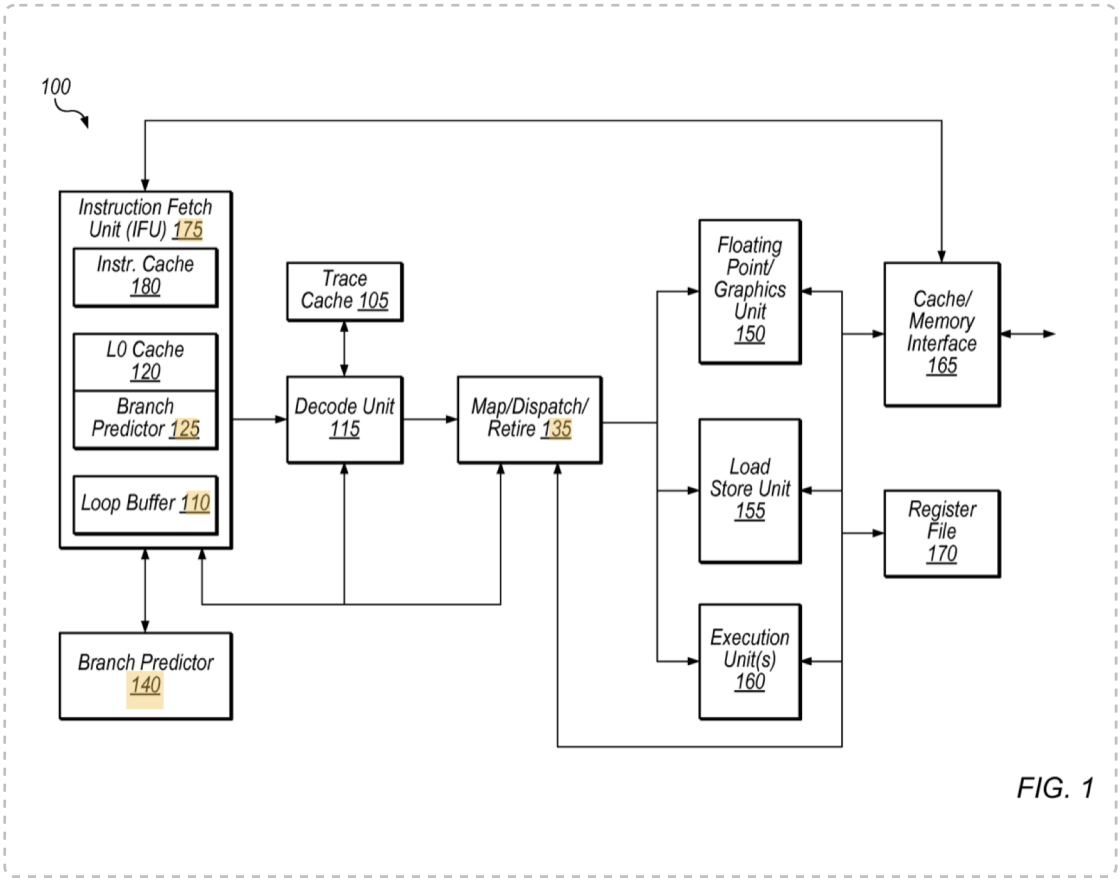
Once we have a loop buffer, what more can we do with it? Well, one minor flaw in the Fetch scheme we have described so far is that only one Fetch Group (ie run of instructions before a branch) is acquired per cycle. This is not ideal if someone writes a really tight loop with the loop body as, say, 3 instructions – now the maximum speed at which we can run is 3 instructions per cycle/regardless of any other details. But suppose we could unroll that loop in the loop buffer... Unroll it three times and we at least have the possibility of running 8-wide per cycle; unroll it more than three times and we may be able to engage in even better optimizations like running multiple dependency chains in parallel.
This patent also provides insight into the size of the loop buffer suggesting (at least as of 2012) that it was arranged as 16 rows of 6 slots (6 slots because decode for that generation of A7-class CPUs had 6-wide decode). To compare, the (somewhat equivalent) structure in Skylake, the IDQ, can hold 64 µOps. (The IDQ is statically partitioned in 2x64 halves, but I'm only interested in single-thread behavior. The point is also somewhat moot because a bug in Skylake means streaming loops out of the LSQ was disabled via microcode update in 2017.)
This is the content of (2012) https://patents.google.com/patent/US9753733B2 *Methods, apparatus, and processors for packing multiple iterations of loop in a loop buffer.* This is of particular interest to Apple because they recommend all code (unless there's a good reason otherwise) be compiled as -Os which, among other things, will not unroll loops that clearly look like they would substantially benefit from unrolling.

But the basic loop buffer is limited to easy loops. Next up in complexity, suppose we have a loop that involves control flow, but nothing hard that needs to be predicted. The most obvious case is a loop that calls a simple function. We do need to predict the return of the function, but that's not hard. This sort of thing can be captured by a
. Depending on the exact design, we may be able to capture the control flow as a Trace in the Loop Buffer, or we may farm it out to a very simple Trace Cache. Either way, we still avoid most of the costs of Fetch.

Finally we have loops that involves some degree of branching. Neither a Loop Buffer or a Trace Cache is a good match for such code, but we can move the loop into an L0 cache that uses lower energy, and we may be able to clock down some structures (eg RAS or Indirect Branch Prediction) for the duration of the entire loop, once we see what is and is not in the loop body. We could even assume (or perhaps test, and validate) that the branches in the loop can be handled by a simple (few entries, just a local 2-bit counter) L0 branch predictor tied to the 0 cache.
(This brings up another issue which is interesting, but I've rarely seen discussed – how to treat loops when training branch predictors. Simply mechanically advancing the history vectors on every iteration of the branch back at the bottom of a loop, or even on a simple branch within a loop, is mainly a good way to remove all useful data from your history vectors... On the other hand one probably wants at least one iteration of the loop to be captured by the history vector, to provide some context of the occurrence of that loop into the history.
So all these different energy saving tricks, even if they were not designed with better prediction in mind, probably help towards that goal! If two or three iterations of a loop go through the normal

branch training procedure, and then a specialized loop mechanism takes over which, among other things, powers down branch training and continually updating the branch history vector, that's probably the ideal outcome even if it wasn't explicitly designed that way.)

(2014) https://patents.google.com/patent/US20150205725A1 *Cache for patterns of instructions*, discusses how and why Apple uses an L0 cache (with associated L0 local branch predictor). The patent also refers to a Trace Cache but describes it in only the most minimal detail, so make of that what you will.
The patent also describes how more complex loops (including even nested loops, as long as they fit in the loop buffer/trace cache/L0) are tracked and handled. If that interests you, *Instruction loop buffer with tiered power savings* (to be discussed below in a different context) gives a few more different examples of nested loop patterns that can be detected.



An issue with the loop buffer is how rapidly do we switch from (higher energy) traditional decoding to a lower-energy loop buffer mode. Something I don't understand is that none of these patents suggest storing what I'd consider obvious, namely a table of "known loops" storing things like the loop start+-size, "number of iterations" vs "unpredictable exit", "types of branches encountered in the loop", etc. With a table like that, you could, much of the time, capture loops into a lower-energy mode every time after the first encounter.
The concern (at the time of the patents, anyway) appears to be that prior iterations of the loop will

affect the branch prediction in later loops, and so one wants to iterate the loop enough times to "saturate" the predictors. I've already mentioned that this seems like a terrible idea once you have very long (hundreds of branches) TAGE-like predictors, and I also have to wonder, for most loops, how much useful prediction information is created by repeated iterations of the loop after the first two or three. My suspicion is that some of these details have substantially changed by now.

Regardless, given a loop buffer, another dimension for improvement is reducing energy.
(2014) https://patents.google.com/patent/US9471322B2 *Early loop buffer mode entry upon number of mispredictions of exit condition exceeding threshold* provides a slight tweak to the 2012 design. One of the items that is tracked by the table tracking candidate loops is whether the loop exit is unpredictable (think for example of something like strcmp). An unpredictable loop exit isn't great, but whatever you do, there is going to be a mispredict at loop exit. Given this, the logic is you might as well at least still run the loop at lower energy out of the loop buffer for as long as possible, and give up on trying to train the branch predictor on details of the loop.

Meanwhile (2014) https://patents.google.com/patent/US9524011B2 *Instruction loop buffer with tiered power savings*, attacks the problem from a different direction, rapidly shutting down some elements of the machinery that feed a loop, then gradually shutting down more elements as confidence in the loop's persistence grows. The first shut-down is of the Fetch front-end; the neat conceptual breakthrough here is that the loop is already present in the queue between Fetch and Decode (I described above, when talking about Asynchronous Fetch, why you want such a queue and why you want it to be fairly large). If the loop is present there, then we can at least stop bringing it in from the I-cache every iteration, and so, even though we continue to power-on the high quality branch prediction, we can sleep the Fetch Predictor and all access to the I-cache. This is done as soon as two or three loop iterations are detected. Then after more iterations, we may feel confident enough to shut down branch prediction (and presumably also copy the loop to the Loop Buffer holding uOps and placed after Decode).

At this point we should discuss the Branch Information Table. We mentioned this when we discussed the RAS.
The patent discussing this is (2016) https://patents.google.com/patent/US10175982B1 *Storing taken branch information.*
The problem to be solved is that we want to store all branch information in a tentative state until the branch retires, at which point the state can be used to train predictors. As an additional design criterion, insofar as possible, we want predicted non-taken branches to be as close to a NOP as possible; ideally they do not take up space in the predictor tables, so that when no prediction is found we can just predict "branch not taken". So if a predicted taken branch is not taken, we need to inform the predictor to update. But if a not-taken branch is encountered, I think we don't bother training on it, we're happy to leave it out of the tables forever if it's always non-taken.

This means
- we want a structure that's like the ROB, a large circular queue into which every branch is inserted at one end (the write pointer)
- there's also a retire pointer; all branches between write and retire are pending in the machine
- there's also a training pointer; all branches between training and retire have retired but not yet been fully incorporated into training
- so the layout looks like (write ..... retire .... training)
- to reduce area, there are two tables, a Branch Information Table and a Taken Branch Information Table
- BIT holds information relevant to all branches, TBIT holds additional information relevant to taken branches
- a BIT slot is allocated at Decode, as is a TBIT slot if the branch will be taken
- but what if the branch is predicted not taken, but that's incorrect? Then at the point of branch execution (where the incorrect prediction is discovered) a TBIT slot will be allocated
- but, something very weird, in the example given in the patent, they suggest the BIT holds 60 entries, while the TBIT holds 96 entries. Why make the TBIT (which is supposed to be a subset of the BIT, isn't it) larger? I think what is going on is that while the BIT operates like a simple circular queue as described, entries in the TBIT are opportunistically allocated and deallocated. In particular I think that at or shortly after Retire "ownership" of a TBIT entry is handed over to Branch Predictor Training which holds onto it for a few cycles doing whatever training needs to do. Thus at any given time, up to 60 pending execution branches may be present in the BIT, along with up to (but probably few less than) 60 associated pending taken branches; along with an additional up to 36 records of taken branches that have retired but their branch information has not yet been integrated into the Predictors.
(Or the patent simply got these numbers backwards?)

- with all this now in mind, most of the fields in the BIT and TBIT make sense. For some of the less obvious ones:
+ a few of the low order bits of every branch address are recorded in every branch, taken or non-taken, conditional or not. I think these are used to update the branch history vectors at the point the branch is Retired
+ for branches that will be predicted (and thus will be looked up in a predictor) we also need some of the higher address bits because they will form the tag to check that the value looked up in the predictor actually is the branch we want (ie check that aliasing has not occurred)
+ for all taken branches we record the branch history vectors at the point the branch executed. We want these to restore branch history state if we need to recover from a mispredict at this branch.
+ the remaining fields are related to the precise details of how either the Indirect Branch Predictor or the Call Stack are updated. Some look familiar'ish, some not. They suggest that the Indirect Branch Predictor uses elements of TAGE (like multiple tables and U bits), but also unfamiliar elements like "BTP hit table". Likewise the RAS pop pointer field is familiar (recover the RAS when additional elements have been pushed on top) but the "RAS branch" stuff only suggests to me that Apple are doing something different from Intel in how they prevent return addresses from being overwritten by specula-

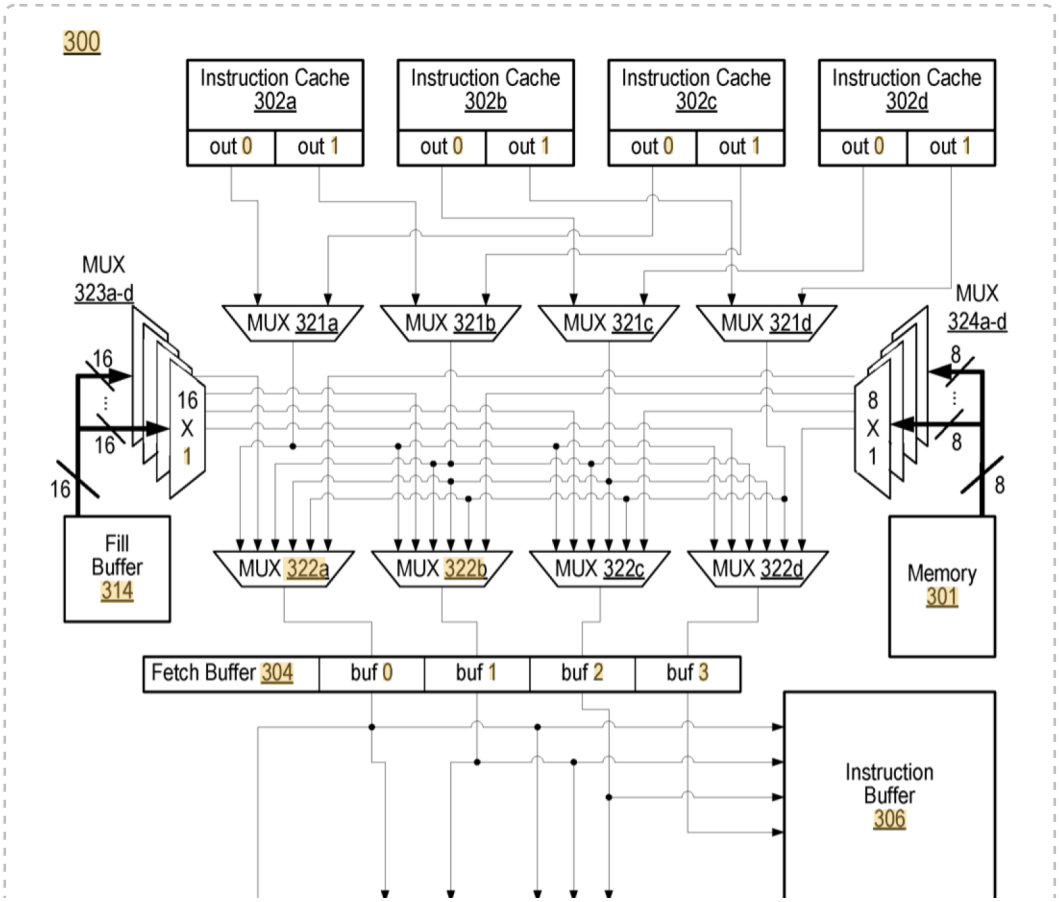
tive code.

- as mentioned, I think the slice stuff means eg one branch per predicted branch (alternate unconditional and conditional; also call and ret as “unconditional”
and maybe 1 target, 2 “all branches”, 1 call/return in the BTIC?

need to run lots of tests here, especially with pairs and triplets of different types of types of branches, to figure out the exact details

We have talked glibly about Fetch feeding an Instruction Queue which then feeds Decode.
Naturally the details of this become ever more complicated the closer you look. For example in any given cycle, Fetch has to pull in some number of instructions from up to two cache lines, and those cache lines can in fact be in any of an actual cache line, or a prefetch buffer, or delivered on demand from L2 or higher. Beyond that, if the Instruction Queue is empty, then the Fetch’ed instructions should be delivered directly to Decode without buffering and, in a worst case, some of the instruction to be delivered to Decode will come from the Instruction Queue, some will come from Fetch, while the remainder from Fetch will go into the Instruction Queue.
This sounds like nightmare. Fortunately we have a patent (from 2016, but it’s clearly describing the A6, so it’s much simpler than any modern implementation). The patent shows us both the naive solution, and a somewhat neater solution.

In[231]:=



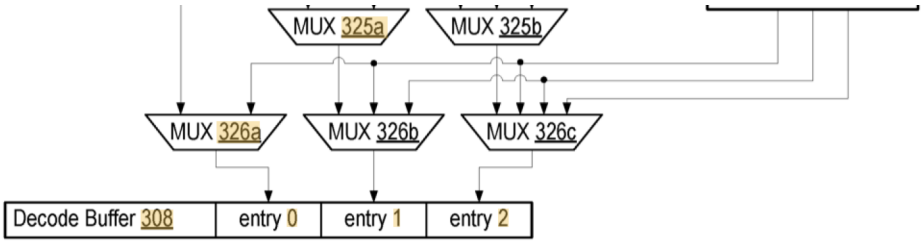

MUX 325a
MUX 325b
MUX 326a
MUX 326b
MUX 326c
Decode Buffer 308
entry 0
entry 1
entry 2

FIG. 3

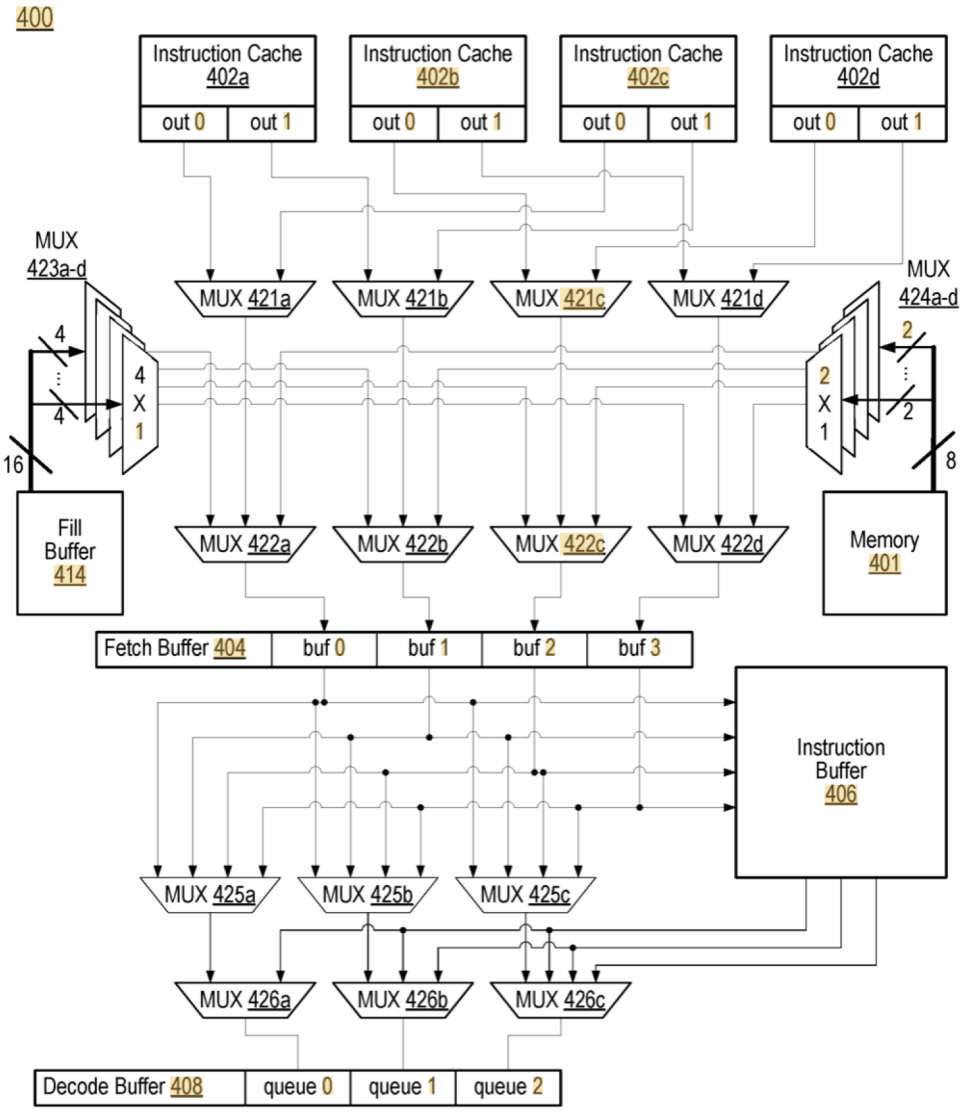
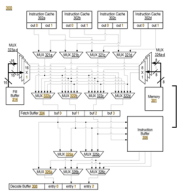
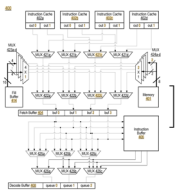

400
Instruction Cache 402a
Instruction Cache 402b
Instruction Cache 402c
Instruction Cache 402d
out 0
out 1
MUX 423a-d
4
4
4 X 1
16
MUX 421a
MUX 421b
MUX 421c
MUX 421d
MUX 424a-d
2
2
2 X 1
8
Fill Buffer 414
Memory 401
MUX 422a
MUX 422b
MUX 422c
MUX 422d
Fetch Buffer 404
buf 0
buf 1
buf 2
buf 3
Instruction Buffer 406
MUX 425a
MUX 425b
MUX 425c
MUX 426a
MUX 426b
MUX 426c
Decode Buffer 408
queue 0
queue 1
queue 2

FIG. 4

Out[231]= Canvas[] × Canvas[]

You probably don't want to puzzle through these in full detail, unless you;re really dedicated, but Figure 3 is the naive solution.
The Fill Buffer is the Prefetch Buffer, so we have that eg the 16 instructions are replicated to 4 16-wide muxes. Each of these buffers chooses up to one instruction which gets fed to the 322a..d muxes. Likewise each sequential pair of instructions delivered by the four instruction cache subarrays, and likewise for memory. The point of these complicated nested muxes is not just to choose the next four instructions we want to Fetch, possibly from more than one of these sources; it is *also* to store the instructions in the correct order in the Fetch Buffer.
Note also how the Fetch Buffer then feeds into the Instruction Buffer and the Decode Buffer so that, once again with a correct selection of controls to the 325 and 326 muxes we can pull some number of (appropriately ordered) instructions from both Fetch and Instruction Buffers, while also dumping other instructions into the Instruction Buffer.

The insight of the patent is that no-one actually cares how the data is placed in the Fetch Buffer. So what Fig 4 does is remove some of the logic from Figure 3. The instructions are still placed sequentially in the Fetch Buffer, but treating it as a circular buffer, ie the sequence of instructions can be placed at any location.
This allows a substantial reduction in the number and complexity of the muxes above the Fetch Buffer, while only requiring a fairly simple rewiring and slight increase in the complexity of the muxes 425a..c (and a similar slight modification to the queueing logic in Instruction Buffer 406).

Of course even before Fetch, we want an I-prefetcher to be ensuring that, as much as possible, every I-line that Fetch touches is already in the I-cache.
I-prefetch is, in a sense, even more important than data prefetch because the machine can do less work while it is waiting for an I-cache miss. It can work through the instructions in the Instruction Queue but that might, best case, cover fifteen cycles or so; enough to get to L2 but no further.
The first helpful technique, not even really a prefetch, is to slightly prioritize I-lines in the L2 cache over D-lines. One can imagine a few ways to do this, and I expect Apple is doing so, but this is well known and probably nothing patentable.
Next up is simply ensuring that, like the branch predictors, the I-prefetcher is not tainted with incorrect data. Obviously this means training it with a sequence of PC's generated at Retire, not the (admittedly more easily available) sequence of PC's available at Fetch. But it also means filtering out PC's generated by interrupts and exceptions...

At this point we're at the prefetcher itself.

Out[232]= and branch cf compare exhaustion experiments
for integrate numbers^2 patent ROB s the^2 with Dougall′

Out[233]= TLB

Out[234]= Prefetch

Prefetch
https://patents.google.com/patent/US9009445B2 Memory management unit speculative hardware table walk scheme
(sorta I TLB prefetch)

As I said there's scope here for improvement! So far our list of minor buglets that should be fixed includes
- treat xzr properly rather than via slow path (this includes using xzr to zero fp registers)
- handle EXTR via hidden register, not allocated register
- use a wider RegisterID (of order ~22 bits) to encode all possible immediates, which would allow for a variety of magic, like setting Immediates in Rename 8-wide, for all possible Immediate values, including the fp Immediates.
- catch b+4 and treat as NOP. Likewise for b.cc+4
- treat b+8 as a predicated version of the next instruction rather than as a branch.

- test the most basic of these
+ cross cache line
+ 1 load crosses 2 stores, 4 stores, 8 stores
+ 1 store covers 2, 4, 8 loads
(test for forwarding, replay, flush)

zero cycle loads -
previous store value (register pattern?) try eg store [x2] ld [x3] where x2, x3 equal
then x3 = x2 + 8, store [x2, #8] ld x3
then indexed versions

stack versions of above

2 load 2 store to climbing addresses
3 load 1 store to climbing addresses
to 2, 3, 4 dift cache lines
to 2, 3, 4 dift TLB pages

TLB tests