Summary of Everything we have Seen

We've covered a huge amount! In this section I want to give a very quick summary/way to think of all the CPU material we have seen, as something of a reminder. With that framework in places, the changes Apple have made since the M1 should perhaps make more sense.

Let's start by asking what makes a modern OoO slow. The traditional answers to that are to split the CPU into a front end and a back end, and to ask about what slows each of these. I'm going to do that, but I'm also going to introduce a new section, the "middle end" and discuss that.

The front end of a machine is ultimately concerned with providing a *stream of instructions* in *execution order* to the rest of the CPU. Recall that we have a branch about every six instructions, and a taken branch about every ten instructions. This means that if we want the machine to execute ten wide, we need to generate a "*fetch group*" of about ten instructions *every cycle*. This starts with a Fetch Predictor that, every cycle predicts the next run of instructions until a jump (so a pair of a PC and a length).
We couple this to a set of various branch predictors (high confidence "easy" branches that can be handled by a simple counter and resolved at low energy in a cycle, complex correlated branches that take multiple cycles to resolve via TAGE, indirect branch prediction via ITAGE) that try, in the next few cycles, to correct any possibly inaccuracy in the Fetch Predictor. If we can correct a misprediction before instructions begin executing, we can simply resteer the front of the machine, not the back-end, and we lose only a few cycles to the misprediction.

This scheme of "cascaded" branch predictors where later (slower but longer latency) predictors can occasionally override an earlier predictor is described in (2022) https://patents.google.com/patent/US12236244B1 *Multi-degree branch predictor*

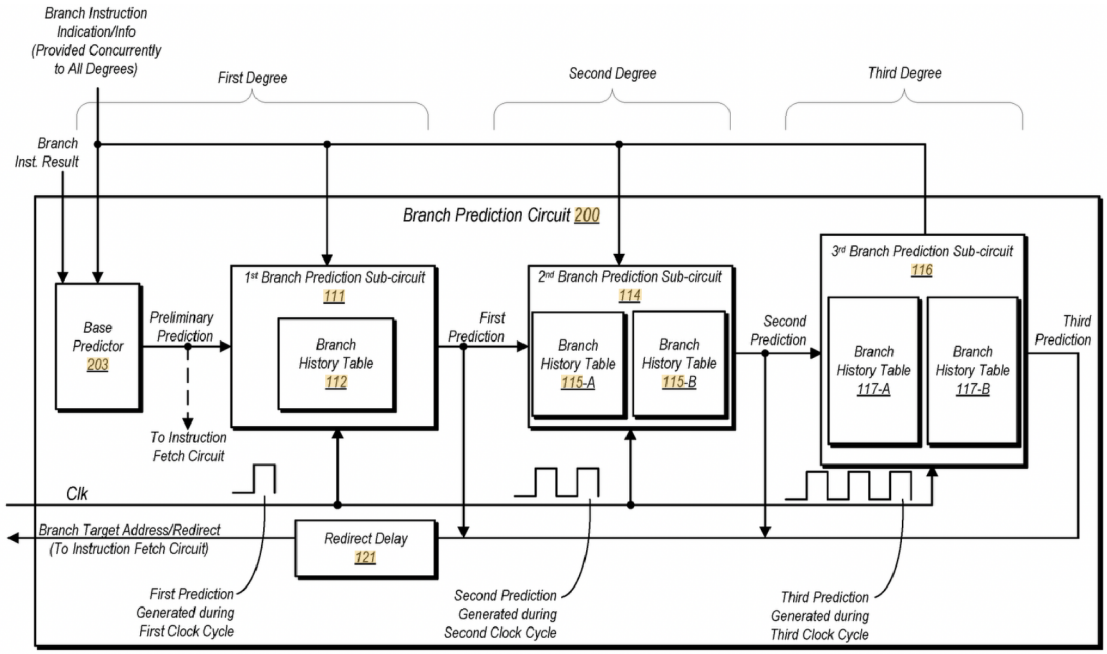


Fig. 2

Note that once we have a very wide machine we are concerned with two different problems.
One is to to be able to load long runs of instructions between *taken* branches because, at least for now, we can only load one contiguous run of instructions per cycle.
The second is to ensure that this contiguous run is correct.

Roadblocks in this scheme (many now fixed, as described below when we get to patents) include
- when we encounter new instructions that have not previously been processed, those will not be

present in the Fetch Predictor. This is fixed by the Scan on Fill predictor which pre-populates a secondary Fetch Predictor as prefetched I-cache lines enter the L1I.
- by default, the Fetch Predictor has to refer to the Return Stack when executing a return, and this costs a cycle, so we load a cycle of Fetch every time we return.
- the Fetch Predictor is of limited size and can't grow larger than can be accessed in a single cycle every cycle. So as of M2 we add a secondary Fetch Predictor that is twice as large, but which costs an additional cycle to access.
- it's not uncommon to have a small forward branch corresponding to something like `if(test){one or two instructions}`. It's a shame to have to break up the contiguous Fetch just because we predict that `test` will be false. Better is maintain a large contiguous Fetch (point one of the two points above) and then simply mask out the `one or two instructions` that we predict will not be executed.
All these techniques together allow us (hopefully) to maintain an average loading of ~10 instructions per cycle (sometimes we can load perhaps as many as 16 contiguous instructions, spanning two I-cache lines, to compensate for occasional bubble cycles). There will be occasional re-steers when a fancy predictor overrides the Fetch Predictor, but hopefully not too many, and hopefully not too costly.

Ultimately what has been described above is the generation of a sequence of pairs (PC address, length). For these to become instructions without glitching, we need the PC targets to be in the I-cache, raising the issue of I-prefetching.

Let's first discuss how the academic world (and almost everyone but Apple) view this. Their solution to this is FDIP, *Fetch Directed Prefetch*. The idea is to generate this stream of PC addresses into a queue, with the other end of the queue reading from the I-cache, and with the intermediate elements of the queue being used to prefetch those addresses from L2. Providing this address queue decoupled I-cache access from address generation, and allows each side to keep going when the other side might be blocked (address generation continues even if we miss in the L1I, likewise I-cache fetching continues even if we miss in the main Fetch Predictor and have to rely on the secondary Fetch Predictor or otherwise lose a cycle or two in the generation of our PC+trace length pairs).

This sounds goods, and I suspect at some point Apple will adopt it; but it's not quite as good as it sounds. There are two types of jumps in code, "near" jumps (things like if/then/else) and "far" jumps to other functions. Near jumps can be handled with fairly trivial prefetchers, for example always bring in the next line along with any loaded I-line. Far jumps are the problem, not easy to predict in large footprint code. While FDIP does generate far jump targets in the address queue, it doesn't do so very far in advance of execution. So you may reduce a load from L2 from say 15 cycles to 10 cycles, which, sure is not nothing, but it's also not ideal. And in really large code footprints, you may even be missing in L2 fairly frequently.

Apple's solution to this is to have a dedicated I-prefetcher which predicts (fairly far in advance) future function calls based on the most recent pattern of branches and function calls. This gives us a lot more

time to I-prefetch, and we can use simple machinery like a next-line prefetcher to handle bringing the rest of the function in time. This is great (and appears to work well) but doesn't conflict with the machinery of FDIP, and as I say at some point Apple will probably add FDIP, though for them it's been much less important than for everyone else (who don't seem to have decent I-prefetchers).

So this is the front-end. For x86 this whole area is a disaster, which explains why x86 reviews and analyses obsess over it. It's much less of a problem with ARM generically, and seems to be almost solved for Apple. (But... one issue which is coming up...)

Next we have the backend, ie the actual execution of instructions. The important thing that slows us down here is loads that miss in the cache. The whole point of OoO (and in particular a massive ROB) is to be able to keep going in our execution while we wait for a load miss to return. For a miss to L2 this is (for a modern core) basically trivial. Even for a miss to SLC we probably have enough capacity (~60 cycles at ten-wide instruction) to hide the miss. But a miss to DRAM is a problem!

At this point let's detour to recommend a paper: 2002 https://jes.ece.wisc.edu/papers/wmpi02.tejas.pdf *A Day in the Life of a Data Cache Miss*. This is an old paper and the exact numbers it gives are surely wrong, almost irrelevant to today. But the framework it gives is very interesting.
Once a load misses to DRAM, the machine continues to execute for as long as possible, then we wait till the load returns. What determines how much useful work can be done? Well, what determines "execute for as long as possible"?
- We might execute until the ROB is full. More realistically we execute until some other resource is no longer available. The tough resources are physical registers and store queue slots. The paper provides simple simulations that this case is common, and so it's worth doing two things
+ make your ROB larger than appears necessary because a large ROB is cheap. You should block because *something expensive* runs out, not because ROB slots run out.
+ consider every trick available to make your expensive resources (like physical registers and store queue slots) go further. This brings up ideas like virtual registers [reduce the lifetime during which a physical register is being used] and the same for store queue slots; and consider Out of Order Retirement (ie allow *non-speculative* instructions in the ROB sitting behind a blocked load to release their resources).

One way to think of Out of Order Retirement is that traditional retirement is (yet another of the principles we keep seeing) a conflation of two different tasks:
- maintain correct execution. That is, ensure that, whatever happens, we can correctly recover after a misprediction or exception.
- release resources after instruction execution.
By splitting apart these two jobs, we can optimize each one separately. One way to do this is is described in (2009) https://www.disca.upv.es/spetit/publications/ieeetoc.pdf *A Complexity-Effective Out-of-Order Retirement Microarchitecture*.

Out of Order Retirement can only work if there are a reasonable number of instructions between the head of the ROB (ie a load that missed to DRAM) and the first instruction that's speculative (and not resolved a few cycles later, eg if it depends on the load) or fault-generating. If this is not the case, another option might be to provide a small SRAM into which one might write the machines state (eg physical registers) should restoration be required, after which those physical registers can be reused. This works because SRAM is so much denser than a register file, and because if we're blocked anyway waiting for a load from DRAM, there's no harm in spending a few cycles moving these registers. This is a special case of what's generically called a Checkpoint mechanism, but I've never seen a discussion of its value in this context. (And to be fair, the value would be very dependent on the quality of your branch predictors...)

- The above case tacitly assumes that few instructions actually depend on the result of the load, and so we can OoO around those instructions as we keep executing. Is that true? We might instead execute until the various scheduling queues are full of instructions dependent on the load, and then we block, far before our ROB is full, because the relevant scheduling queue(s) are full. Perhaps the single most interesting and unexpected finding of the paper is that this is very rare. In the worst case you get maybe ~100 instructions dependent on the load, but usually far less, so you very rarely block because the issue queues fill up, rather you block because the ROB fills up. And this holds even if you have a very large (like ~4000 capacity) ROB.

- The fly in the ointment, the big problem, is *control dependencies*. I've mentioned this before in a few different situations. The above two pieces of good news (essentially we can keep executing past a load until we use up all our physical registers) doesn't help if a quarter of the way through execution we hit a branch dependent on the load, mispredict, and three quarters of the work in our ROB will be flushed once the load returns. This is what we need to keep in mind as we keep growing our machine!

So how do we use the above insights?
Obviously we want to prefetch whatever we can. Every time we avoid a miss to DRAM, we avoid the problems above. Along the same lines, increase the capacity (and "effective capacity") of caches by whatever means are feasible – zero content caches, better cache address hashing, better schemes for cache placement and replacement algorithms, cache compression, etc.
And obviously we want ever better branch predictors.
On the other hand, a data dependent branch is harder to predict than many other branches.
So it may be worth putting what looks like a lot of effort into tracking just a few (16 or 32 or so) load PCs that are known to feed into low confidence branches, and prefetch the targets of those loads, in a new type of dedicated prefetcher.

There is a second, less obvious, strategy also available.
Consider a simple `if(test){some instructions}` branch. There is a reconvergence of instruction flow after the `some instructions`, and it seems a terrible waste to have to flush all the instructions after that reconvergence point when the only thing that's wrong is that `some instructions`

either were or were not executed. The problem is how can you use this insight? `some instructions` modified a few registers, and those registers might have influenced a few later registers and so on and so on. There have been proposals to try to keep track of the "poisoned" registers and instructions, and flush then re-execute only those, not everything after the mispredicted branch, but no-one likes them very much – as you might imagine it turns into just a vast amount of state you have to track, along with complicated machinery to selectively flush just some instructions and register values.

But assume we wrote the above code differently, not as a branch, but as a few `some instructions` each predicated by `test`. Now we have converted a *control flow dependency* (with implications for Fetch, and requiring a flush on mispredict) to a *data dependency* (which just means some instructions that sit around in the Scheduling Queue(s) unable to execute until the load returns! And we have already seen that, contrary to what you might expect, the Scheduling Queue(s) are usually not a critical resource that frequently fill up after a load miss.

Obviously not all instruction sequences are amenable to predicate conversion.
But it seems that there is scope here for
- compilers that are more aggressive in terms of using predicates rather than branching when it's fairly clear that the payload of the if() statement is just a few instructions, and the if() depends on a recent load.
- the CPU detecting this configuration of events (low confidence short forward branch that depends on a load that frequently misses to DRAM) and converts the branch at decode time to a stream of one or more predicated instructions (as we described is sometimes done by recent POWER models).

So we can say that
- Apple seems to be doing OK with the front-end. FDIP would be a nice but minor boost, while predicate conversion of specific short forward branches might be surprisingly helpful.
- On the backend a load→branch prefetcher would probably be helpful, to help with the loads that feed branches which cannot be predicate converted.
- Ongoing tweaks to make the effective ROB size ever larger (eg techniques like virtual registers) will help because there *are* instructions that can be executed way past the load miss, as long as resources like physical registers are available.
Techniques like Long Term Parking that move instructions dependent on load misses out of the scheduling queue may not be necessary if the number of saved scheduling slots is just not that high (especially if we also use virtual registers, so those instructions are also not locking up physical registers).

So if both the front end is solved, and the back end is solved, game over, right?
Well there is also the "middle end" that I mentioned!
We know that we can think of the execution of a program as a graph of dependencies, one instruction dependent on another. Two features of this graph are
- If we're willing to look within a window of a few hundred instructions of the current PC, we can usually find a fairly large number of independent instructions every cycle. This is what makes it worth having a 10-wide CPU, as long as it comes with a large enough scheduling window (total capacity of all the

scheduling queues).
- there is a critical path through this program execution graph that sets an upper limit on how fast we can execute the program.
I call the “middle end” everything between the front end and the back end that works with these two facts.

So what can we do with this concept?
- We want the scheduling window to be as large as appropriate, ideally large enough that every cycle we can find ten independent runnable instructions. How large is this? I’ve never seen anything that really addresses this question. But given that a Scheduling Queue is a hot structure (takes a lot of area and power, and limits frequency) it’s probably the case that we aren’t making them as large as ideal.

I’ve suggested that one way to improve this situation is to provide a parallel Scheduling “Queue” that holds instructions that are either known to be runnable at Rename time (all inputs are already available) or almost runnable (one input is not available) and that a surprisingly large (~30%) number of instructions can fit in this queue. (Of course the academic simulations were done on much less wide machines, maybe this is not as true for a 10-wide machine?)
Another possibility is to increase the size of the non-ordered buffer that feeds into each Scheduling Queue, since that’s a much cheaper structure.

- We want to execute the critical path through the program as fast as possible, leaving everything off the critical path to execute in its own good time using whatever slack is available. This means we want to track instruction criticality and schedule based on it. (Right now we schedule based on instruction age, which is an OK heuristic, but we can do better.) If we have criticality info available for each instruction other things become easier, like the scheduling of a secondary queue of known-runnable-or-almost-runnable instructions or use of a larger non-ordered buffer.

If you have total knowledge of program execution, the critical path and criticality are perfectly defined. In the absence of such knowledge, you can use various heuristics to define something that, while not perfect, is good enough.
One obvious path is to build up the backward slice (the set of instructions that feed into) loads, and we’ve seen papers that describe how to do that.
A second path, less obvious, but worth considering for code that is not limited by memory, is to define criticality by building up the backward slice of *unpredictable* branches. The intuition here is that these branches are going to cause trouble regardless, but the faster we resolve them, the fewer cycles (and the less energy) we waste on going down the wrong path.

- The best way to make the critical path execute faster is to shave cycles off it.
Some of this can be done by moving execution from Execute time to Rename time. Apple is already pretty aggressive in this respect in terms of handling MOVs and some immediates, and the various zero cycle loads, but there is scope for more. If Intel Golden Cove can perform short additions of an immedi-

ate (apparently up to 8b, with claims of even 32b) at Rename, perhaps Apple can do the same, along with even more trivial logical instructions like ANDs against an immediate?

Another way to do this sort of thing is via fusion. We've mentioned various instruction fusions (and will see more below). Another type of fusion I've mentioned is when we have two "easy" back to back instructions that modify the same logical register, eg an ADD (especially a 32b ADD) along with a logical instruction. There should be sufficient timing margin to execute both of these in a single cycle.

Yet a third idea in this space is value speculation. This is leading edge stuff, but as we'll see below Apple is making some initial explorations in this space.

Another way to optimize the "middle end" is to treat float and vector differently from scalar.
It's generally agreed that float/vector work (at least for modern designs) is primarily limited by *throughput* not by *latency*. In other words you're better off executing *more* vector ops per second than making each vector op *faster*. Apple already use this in that they don't bother making any vector op lower latency than two cycles even the most trivial operations, like negating a float.
But how much further can/should you take this?
Some papers, like (2024) https://dl.acm.org/doi/pdf/10.1145/3665314.3670805 *ReOVE: Restricted Out-of-Order Execution for Superscalar Processors with Vector Extension* suggest simply not even bothering with out-of-order vector execution. If you do this completely, you can drop much of the (FP/vector) scheduling machinery, all the vector rename machinery (including many many physical vector registers) and some of the load/store queue machinery. You can then translate that saved area into additional vector execution hardware.

What is not clear to me from this paper is how much this hurts performance in a very high end CPU. (The paper claims only 2.7% performance hit, at a 21% area reduction, but the CPU they modeled was not especially high end; better than an A6, weaker than an A7; and in particular only had one vector unit.)
However we can ask if a simpler version of the idea is worth doing. For example, if we swapped out the sophisticated vector/FP instruction scheduler for something that simply executed instructions in order, how much would that hurt performance vs the area and power savings? Or alternatively, execute out of order, but just pick randomly from the vector instructions that are executable this cycle, rather than bothering to select the oldest instructions? It's possible Apple do this already. When we explored the Instruction Scheduling patents we saw that there seemed to be different styles of instruction queue trying to do different things, and it may be that the specific style used for FP/vector is in fact set up to provide a large pool of instructions, but just choose randomly whatever is runnable each cycle without the costs of even attempting to maintain temporal ordering?

Another interesting possibility is the ARM Cortex 925.
(2024) https://www.anandtech.com/show/21399/arm-unveils-2024-cpu-core-designs-cortex-x925-a725-and-a520-arm-v9-2-redefined-for-3nm-/2

This surprised many people by providing 6-wide FP/vector support which seems like a fair bit of extra area for something people have generally not prioritized. Possibly AI drive some of this, but it's also possible that they have aggressively implemented some of the ReOVE ideas, which saved so much OoO silicon that they could add 50% more FP execution throughput without taking much of an area hit.
It's interesting to see that their IPC does not seem to have jumped more than about 30%, even for the most obviously FP/vector-friendly workloads (the chip is not yet released as of when I write this, so we're limited to ARM info). This suggests, perhaps that they are taking a hit from less sophisticated FP/vector OoO support -- but overall the tradeoff is worth it?

Bottom line is that there remains a fairly decent set of additional optimizations available to Apple to improve all three of front, middle, and back end! No need yet to panic that we have hit any sort of performance ceiling!

New material since version 0.93

I'm going to record here various interesting things I've discovered since 0.90, but without expanding on them much since my time is limited.

New in A15:
Dougall has so far discovered in the A15/M2 the following differences: https://twitter.com/dougallj/status/1534002276091629569
- ~10% smaller ROB and some other structures, slightly larger load/store scheduling queue. Probably these were slightly rebalanced/optimized based on traces from the A14/M1
- better memory latency in some situations. Could this be the long sought *stride address prediction* I could never find on the M1?
- some special purpose registers (TPIDRRO_EL0, used by multiple threads within a single process to access thread-local data) are made faster
- ADR/ADRP, used to access global data, made faster (now handled at Rename, like MOV immediate)
- Blizzard (the small core) is now 5-wide rather than 4-wide, with a 33% larger ROB and probably other substantial expansions. "Small", but probably already more capable than the good old A7 Cyclone that really got Apple into the CPU game!

New in A16
When we get to A16 there seems even more slight tweaking of structure sizes.
For the basic ROB rack (unit of 7 slots, one of which can be a failable instruction) we have
- for P cores: about 330 for M1, 290 for A15, and 270 for A16.
- for E cores: about 60 for M1, 75 for A15, and 108 for A16.
These are substantial reductions, which suggests that perhaps the structure has changed. (For example a "rack" may now hold 8 rather than 7 instructions? Although dougall suggests this is not the case, that NOP measurement still gives a 1:7 ratio). Ultimately, remember, usually the real constraint is the

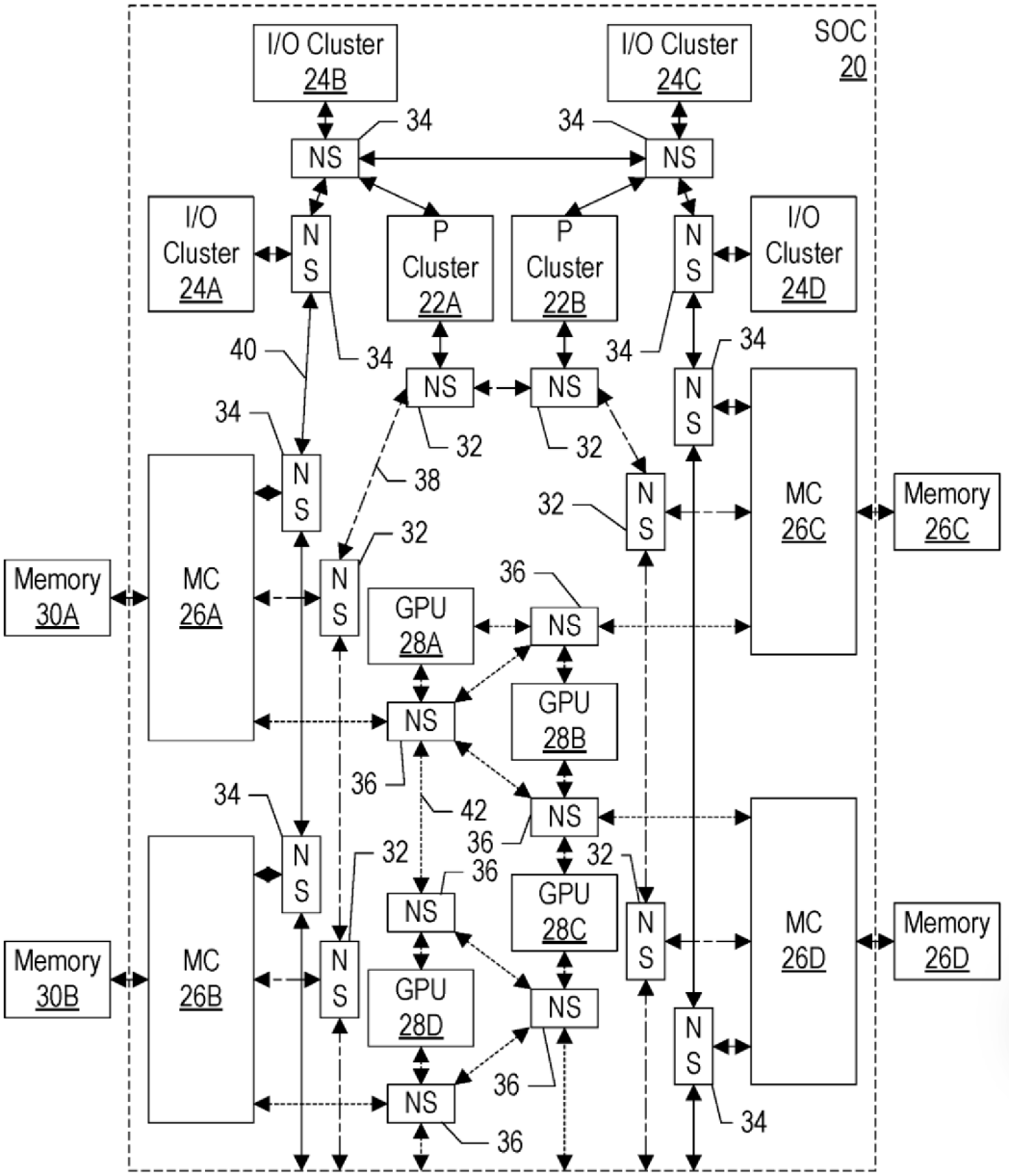
number of physical registers; growing that is hard while growing the number of ROB entries is easy. It could be argued that the M1 had more ROB entries than was justified by the number of physical registers, and so this rebalancing makes sense.
Remember also that Apple's constant trick is to split a structure that performs two tasks into separate structures each optimized for one task. Tying a single ROB rack to the task of both tracking one failable instruction (branch or load/store) and some non-failables goes against that philosophy. Maybe they split the ROB into two separate pieces for these two tasks, so that long runs of failable or non-failable instructions don't "use up" slots for the other type of instruction, allowing the overall number of slots to shrink even though the ratio remains 1:7?

NoC (multiple independent networks)

(2021) https://patents.google.com/patent/US20220334997A1 *Multiple Independent On-chip Interconnect*

Split the existing NoC (which is already multiple networks, for address vs data, with a separate network for IO/interrupts) into multiple "address+data" networks. Examples might be an IO network (ordering, coherency, high latency), a CPU network (ordering, coherency, low-latency) and a GPU/NPU/ISP network (relaxed ordering, non-coherent). These might also have different topologies.

We see an example here:


SOC
20
I/O Cluster
24B
I/O Cluster
24C
I/O
Cluster
24A
I/O
Cluster
24D
P
Cluster
22A
P
Cluster
22B
NS
MC
26A
MC
26B
MC
26C
MC
26D
Memory
30A
Memory
30B
Memory
26C
Memory
26D
GPU
28A
GPU
28B
GPU
28C
GPU
28D
34
40
32
38
36
42

The IO network is represented by the solid lines (marked as 40), the CPU network as the dashed lines (marked as 38) and the GPU network as the dotted lines (marked as 42).

The obvious win is in power (each network can be optimized for its task, so that eg the GPU network can devote substantial resources to bandwidth management, while this is probably less necessary for the IO and CPU networks); but of course there are simply more network resources available which presumably helps performance under extreme conditions (ie all of IO, CPU and GPU working hard).

A revised version of this patent, (2022) https://patents.google.com/patent/US20230239252A1 *Segment to Segment Network Interface*, contains much the same material, but lays greater stress on each of these independent networks being hierarchical, in a very particular sense of being somewhat like the Internet. The idea is that a local network knows essentially two things – how to perform local routing, and how to hand-off a packet with an unknown address to a "long-distance" router. This limits the knowledge the local network is required to possess as to the structure of the overall chip and overall network.

Somewhat related is (2021) https://patents.google.com/patent/US20220365896A1 *Transaction Generator for On-chip Interconnect Fabric*, which is mostly about testing, but which reveals a few more NoC details, like how the NoC is split into a local level (think IO or CPU cluster), often different, and each optimized for the characteristics of the cluster, all connected by one of the above three global networks.

One of these local networks is described in more detail in https://patents.google.com/patent/US20220237028A1 *Shared Control Bus for Graphics Processors*. Now the GPU network itself is a two level ring bus, the higher level communicating between GPU clusters (these could be entire chips like an Ultra, or possibly units of say 8..10 GPU cores), the lower level communicating between cores within a cluster.

Also in this vein is (2021) https://patents.google.com/patent/US20220365900A1 *Programmed Input/Output Message Control Circuit*. Mainly I like this patent because it clarifies so many things previously seemed correct but were unclear. It clarifies that we need a special way to address low-level hardware (for example to configure a particular switch in the NoC) in a way that cannot easily be handled purely by addresses based on memory locations, and that this addressing is called PIO. It also points out some less obvious points, namely that this mechanism (required at the least for startup/reboot, and for debugging) needs to work before fancier machinery like credits or perhaps even routing tables, have been set up.

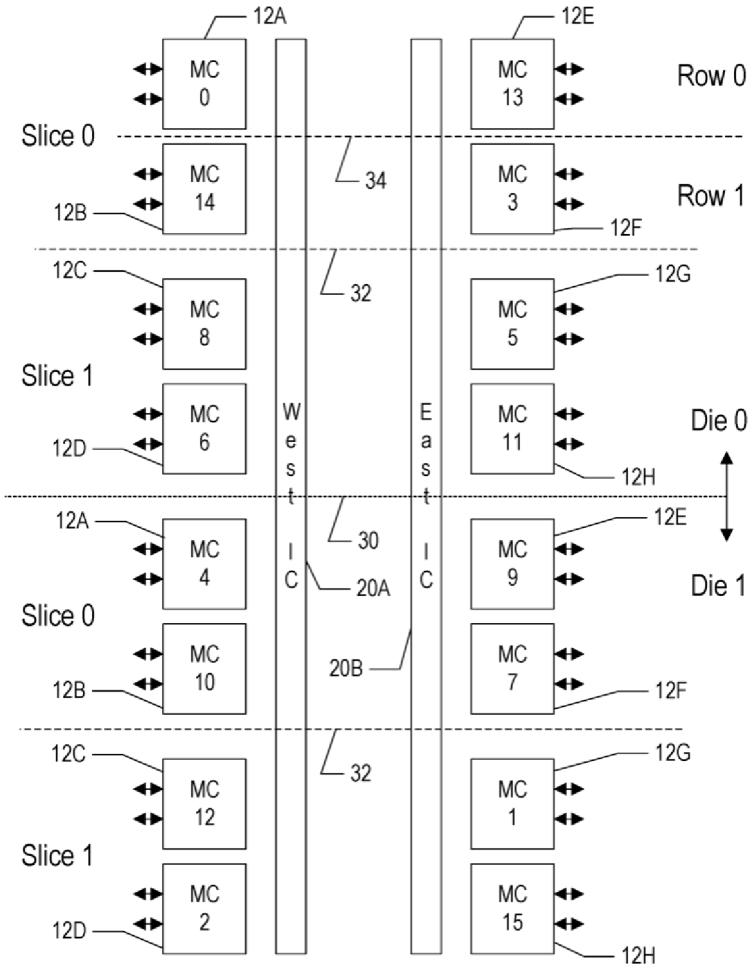
One way to do this is via a separate network which doesn't have to be very sophisticated, but which will take some area. An alternative is to add some lightweight additional storage and logic to every "memory controller" (by which I think they mean "memory-address-based buffer/router") that treats PIO as a special case. This side logic buffers PIO transactions and sends the transactions to the correct place via slow, but simple and reliable, mechanisms, for example using broadcasts and simple sequencing/flow control (not allowing a new transaction till an existing transaction is complete). The

details are uninteresting past the main point, that we replace the area of a separate PIO network with what I assume is the smaller area of a distributed “PIO mode” attached to each node in the main network. This seems analogous to the way interrupts were also moved from more of a separate dedicated network to more of a special type of packet on the main network.

(2021) https://patents.google.com/patent/US20220342588A1 *Address Hashing in a Multiple Memory Controller System*

In the first place this confirms an obvious point – that the physical memory address space is hashed then spread over multiple memory controllers; but it gives some details as to how the spreading is done so that the most common patterns (ie sequential-like streams) are maximally spread so as to make best use of the full NoC bandwidth at every level. More interesting are two other points:

- after each routing decision is made (to a particular chip, then a particular slice, then row, then side; and also for subsequent routing internal to the memory controller to different sub-structures within the DRAM chip) the address bit(s) that forced that routing decision are dropped, because they’re no longer needed and moving/storing them takes energy. Maybe not a massive energy saving, but a cute idea!


12A
12E
MC 0
MC 13
Row 0
Slice 0
MC 14
34
MC 3
Row 1
12B
12F
12C
32
12G
MC 8
MC 5
Slice 1
MC 6
West IC
East IC
MC 11
Die 0
12D
12H
12A
30
12E
MC 4
20A
MC 9
Die 1
Slice 0
MC 10
20B
MC 7
12B
12F
12C
32
12G
MC 12
MC 1
Slice 1
MC 2
MC 15
12D
12H

On-chip machinery is tracking how busy each memory controller is. Every so often this business is reported up to the OS, which also knows the total physical memory footprint in use. Based on these, certain memory controllers can be shut down (to save energy, of course) in low usage situations. If there are a few pages that are still being used, those pages will be migrated/consolidated (by changing the VM mapping) onto a busier memory controller. Depending on details either the memory can be completely shut down (not being used at all, all modified pages migrated to another controller) or the memory can be placed in self-refresh mode until access is required. Presumably there's also some input from the IO system as to how much memory being used as page cache is really valuable, but that's not described.

There are a few patents in this vein, handling various technical aspects of a multi-die system like testing and debug. The most interesting is probably (2021) https://patents.google.com/patent/US20220365579A1 *Die-to-die Dynamic Clock and Power Gating* which talks about how, before any die is shut down (or similar changes) all the other dies are queried to see if there are pending transactions targeting the relevant die, and if so the transition is delayed till those are cleared.

(2022) https://patents.google.com/patent/US20220318136A1 *I/O Agent*

We've seen multiple patents that try to mate the relaxed semantics of Apple Fabric with the specific rules for different IO buses. This patent seems to be the final answer to solving the issue once and for all. The idea is that each bus with particular rules is placed behind and IO agent and talks only to that agent; the agent mediates between the IO bus rules and Apple Fabric rules. In particular the IO agent is able to act in a cache-like fashion, in that it has a small local buffer ("mini-cache") and understands the SoC coherency rules. Rather than sequences of Io transactions being forced to route all the way to/from SLC and to complete one at a time at some small width (anything from 8b to 64b) the IO agent is able to request the relevant line from DRAM/SLC, modify it locally, and hold onto it in the expectation that multiple successive requests will probably change successive parts of this line.

There are related patents (2022) https://patents.google.com/patent/US20230064526A1 *Ensuring Transactional Ordering in I/O Agent* and (2022) https://patents.google.com/patent/US20230063676A1 *Counters For Ensuring Transactional Ordering in I/O Agent* which discuss, within the above framework, ways to reorder transactions for optimized performance. An example is something like if a snoop comes in that hits the IO Agent cache, the snoop reply may be delayed for a few cycles if that allows the last elements of an IO transaction using that cache line to complete, before the snoop is acknowledged.

Related is (2021) https://patents.google.com/patent/US20220357784A1 *Telemetry Push Aggregation*. Now these "wrapper" IO agents don't just translate protocols and handle cache, they also gather a constant stream of statistics which are reported to the Power Manager which in turn sends messages to the IO agent to sleep or change the performance characteristics (eg frequency or bus width) of various elements behind the IO agent.

Similarly we have (2021) https://patents.google.com/patent/US11550745B1 *Remapping tech-*

niques for message signaled interrupts. Now the issue is the ways foreign devices can handle interrupts (eg PCIe in-band Message Signaled Interrupts). Once again the wrapper IO agent intercepts the outgoing interrupt and remaps it to the uniform scheme used by all Apple Fabric interrupts. In this case the win is not only simplicity but some security improvement in the that "raw" PCIe interrupts have an undesirable degree of access to raw DRAM.

Power

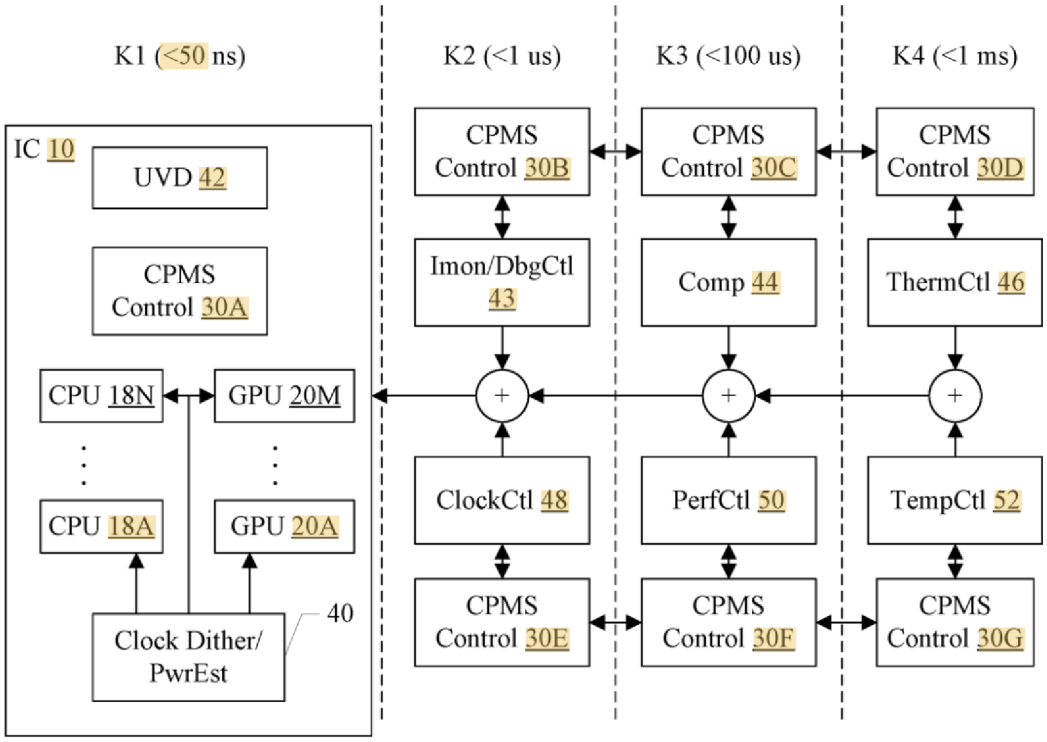
(co-ordinated behavior across multiple very different timescales)

(2021) https://patents.google.com/patent/US20220137692A1 *Systems and Methods for Coherent Power Management*

So far power management has been about

- performing "fast" actions as close as possible to the IP block being controlled but
- with centralized overall control (one place to make a request that changes power states for the SoC or a sub-IP block)

This gives us what we might call "spatial" coherence (each sub-system knows what another subsystem is doing more or less simultaneously) but does not give us "temporal" coherence. This patent changes that.



The above image gives us an idea of the issues. We see that there are four levels of control, operating at four different time scales.

On the fastest time scale (which is still of the order of ~150 cycles) we see that the primary goal is to respond to an undervolt (UVD) event. This is done by clock dithering (preventing the clock for emitting a pulse for a cycle) which will prevent most work being done for a cycle; if this were done for, say, every second cycle on average you'd reduce instantaneous power in that block by about a half. This goes hand in hand with trying to prevent the UVD from even occurring via power estimation over the next few cycles (based on the DPE and the upcoming instruction flow).

The next time scale up is based on varying the clock frequency within a certain range, and is controlled by tracking the current flow.
Next up is controlling the voltage of each voltage island (which in turn sets the range of possible frequency adjustments).
Finally at the slowest time scales we monitor temperatures.

The point of the patent is that in the past each of these mechanisms operated somewhat independently on their separate time scales, but the patent describes ways by which each level can communicate to the central power controller what it plans to do next. Based on that, lower levels may modify their plans; for example if the stage one level up is about to increase its power delivery, eg by increasing voltage, the system one level down may alter its plans which were to reduce frequency or whatever; those plans may not be appropriate once more power is delivered.

This sets up the background; the specifics of the patent are concerned with ultra-low power. In particular, even though (as we have seen) Apple uses sophisticated power supplies and IVRs, there is a limit to how low a power level these can efficiently supply. What to do when the system still requires power, but at lower levels than this minimum? Given all these levels of control, if we know this case is about to occur, the answer is to "coast" – we switch off the power supply and let the low power IP block(s) operate off capacitance for as long as possible; then we restore power until it has recharged the capacitors, and repeat.

This is a patent that keeps being revised, and accompanied by the companion patent (2020) https://-patents.google.com/patent/US11513576B2 *Coherent power management system for managing clients of varying power usage adaptability* which is much the same ideas, but operating at even slower time scales.
Whereas the previous patent talks of timescales up to 1ms, this patent talks of timescales of 10s and 1s, and is about informing the OS of thermal/power pressure, so that the OS can take appropriate OS-level response (DVFS, changing scheduling, eg to pause background/less user visible apps, and sending thermal/performance notifications to apps so that they can, if possible, dial back their activity).

For example (2021) https://patents.google.com/patent/US20220147132A1 *Quality of Service Tier Thermal Control* is essentially the same point (OS response to excessive power) but handled by adding an additional limit to the maximum control effort, namely the maximum extent to which we will allow the various temperature sensors to rise is set to different values depending on whether we are executing priority code vs background code.
We could think of this, for example, as even if we are willing to pay a certain energy price for running some background services, if the mac or iPhone is currently hotter than usual, let's run those background services at a slower rate, or even run them later, so as to allow the machine to cool down.

A constant theme we have seen with Apple is placing "adaptors" between different hardware blocks translating communication native to some block (interrupts, power reporting, clock control, etc) into a

standardized form for a centralized controller. You might think we're at the end of this, but not yet! See (2021) https://patents.google.com/patent/US20230076507A1 *Controller for Multiple Sensor Types in an SoC*.
The background seems to be that in the past a single sensor type (think of, eg thermal sensors) was used across the entire SoC, but a single thermal sensor type is not optimal for all use cases. Or perhaps two or three different types were used, each type reporting their results differently?
To deal with this, the patent describes an adaptor that sits between each sensor and "the NoC" and transforms the results of the sensor into a uniform format for the rest of the chip. The adaptor handles both protocol conversion and value conversion (eg converting from raw sensor voltages or whatever, via lookup tables and mathematic transforms, into something standardized like Kelvin).

(If possible pre-calculate the next few frames to be displayed)

(2022) https://patents.google.com/patent/US20220413589A1 *Electronic display power management systems and methods*

More power saving. The idea is that there are some situations where the next few frames to be displayed can be predicted fairly easily in advance. (Media decode is one possibility, but another is something like scrolling, or an Apple Watch face.) Under these circumstances, the display system switches to 'Flipbook mode". The relevant IP blocks generate a few frames in advance stored to memory, and enqueue (time stamp, frame address) pairs in the display controller. The IP blocks can then sleep for a few frames while only the display controller and memory controller need to stay awake to handle the subsequent frame displays.

SoC

A theme we've seen repeatedly on the SoC is rationalization of various elements (how power is handled, how QoS is handled, how blocks are powered on and off). How far can this idea be taken? Well consider (2024) https://patents.google.com/patent/US20240160479A1 *Hardware accelerators using shared interface registers*, which applies the idea to "accelerators", which can broadly be considered IP blocks, ie even things like controllers that are handling the precise voltage levels of pixels in a display. The patent suggests that there is value in, as far as possible, providing each IP block with a common register API so that common functionality is monitored and controlled in the same way, and lists a large number of configuration registers (some not obvious, for example registers handling boundary conditions) that could usefully be unified under a common architecture.

Another way in which this idea can be pushed is co-ordination across different chips.
Something like an iPad has not just the SoC but a variety of subsidiary chips like display hardware, USB hardware, radios, etc. Optimal power handling requires these all be in sync, for example they can't all increase their power draw simultaneously. (2022, based on an earlier 2020 patent) https://patents.google.com/patent/US20220382701A1 *Billboard for Context Information Sharing* describes some of how this is handled.

Sitting on a common bus with all these chips is a *Coexistence Hub Device* which tracks the state of all the chips. As you might imagine this does many things, but one of the things it does is maintain a "billboard", a central repository of state for all the other chips. When these chips need to perform some

change that requires co-ordination, they can first check with this billboard to see how this will affect the rest of the system. This scheme deals with the problem you otherwise face that aggressively powering down chips when they are not needed means that they miss out on general announcements of how the rest of the system is changing. If system design interests you, the whole thing is kinda fascinating in terms of (yet again) how much is offloaded from the CPU and OS, once again trying to ensure that every piece of silicon can sleep as soundly as possible for as long as possible.

Memory Controller

When we last left the memory controller, it was at around its third generation in 2018. By 2021 we see evolution in the direction of a 4th generation. The third generation controller sorted requests via successive queues to try to achieve both high bandwidth and meeting various QoS constraints. The final two stages of this sorting chose

- for each bank, the virtual channel (essentially the client agent) that was going to be granted access to that bank then,
- of all the possible banks, which bank to access next

These two stages are decoupled and mostly independent of each other. The one linkage is that bank selection is driven (if relevant) by whichever virtual channel has pressing latency requirements.

This scheme could mean that, if one agent is spreading requests across multiple banks, while a second agent is limiting its requests, for some reason, to one bank, then the second agent gets a noticeably reduced bandwidth. This is to some extent unavoidable, but can be improved by hitting the second bank as aggressively as possible, ie by ensuring that the last stage, which decides on which bank to access next, hits the second bank every time as soon as it becomes available. In other words we are now coupling, to some extent, the bank selection stage to the virtual channel selection stage based not only on an agent's latency QoS but also on an agent's bandwidth QoS. The details are, of course, in exactly how to implement this!

(2021) https://patents.google.com/patent/US20220357879A1 *Memory Bank Hotspotting*.

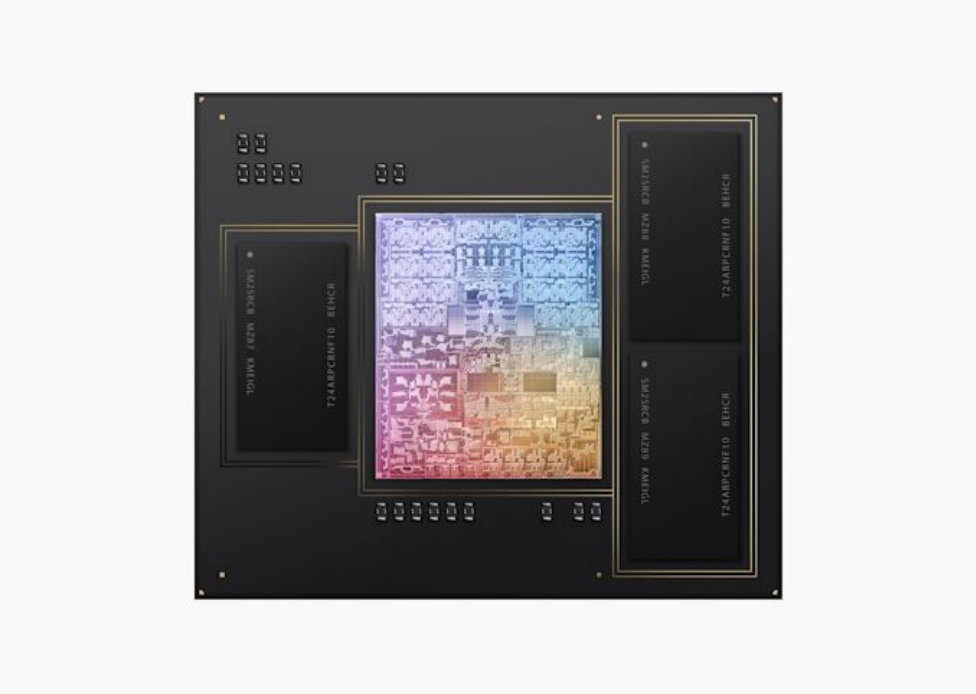
Once we have Ultra-type designs, and perhaps even earlier with Max systems, the question of memory energy arises. These systems can provide a lot of DRAM, but may also be somewhat bursty in how that memory is used; most of the time only a fraction of the system's DRAM may actually be useful, but of course at times large tasks are fired up and it is all used. Is there some way we can dynamically alter the amount of DRAM in use?

(2021) https://patents.google.com/patent/US20220342806A1 *Hashing with Soft Memory Folding* suggests a few ideas.

The starting point is a programmable system for hashing physical addresses. Obviously you always want to hash physical addresses so that (as far as possible) requests are spread evenly across all SLC's and memory controllers, and then within the ranks, banks, and pages of the DRAM itself. But if you can program the hash, and if this programming has some flexibility, a few other options open up.

One possibility is that we can balance the hash to send twice as much traffic to one SLC as to another. This seems strange and pointless, but the M3 Pro seems to require this. Recall that the M3 Pro has

~150GB/s of DRAM bandwidth, corresponding to three “memory channels”, and die shots of the M3 Pro indeed show two DRAM elements on one side of the chip and one DRAM element on the other side:



It’s unclear how Apple handles this but in some sense the easiest solution is probably something like balancing the hashing algorithm to try to to route $\frac{2}{3}$ of the traffic to one SLC+memory controller and $\frac{1}{3}$ to the other?

Another possibility is that we dynamically vary the hash. In other words as “appropriate” memory capacity rises and falls, we change the hash to route traffic from say four SLCs (full Max complement) to three, two, or even one.
One element this requires is, when we want to shut down an SLC/Memory controller/associated DRAM, we need to move any still useful data in that DRAM to some other DRAM. That’s not too hard.
The more difficult part is dealing with users of that DRAM. There might be multiple clients using a

particular virtual→physical mapping to that same physical page, and if we want to modify the physical address we would need to modify those mappings. Then there are clients that operate in the physical address space, like some hardware and DMA agents. The OS itself may use physical addresses internally. It seems like a huge mess, to dynamically take DRAM offline! Even so that's what the patent seems to be suggesting.
One possible to think of this is that the hash (possibly augmented by some tables) acts as an additional level of address mapping between the SLC and the DRAM, mapping what we might call "SoC" addresses (which almost every SW and HW client and the OS think of as "physical" addresses) to "DRAM" addresses. So by changing the hash, I change the (SLC, rank, bank) in which a particular physical page resides. With some *very careful* sequencing I might be able to "pause the machine" (eg just like before it goes to sleep) move the data page by page from its current physical page to the page where it will be after rehashing, then flip to the new address hashing and power down the now no-longer-required SLC/memory controller/DRAM?
Reading the patent it's unclear quite how dynamic Apple intend to be, and maybe what I have described is aspirational, a goal for one day, but not yet attempted?

The third element the patent describes is somewhat cute. As the data (and more precisely its target address) moves through the NoC arriving at the SLC then at a memory controller within the SLC, then one of the queues targeted at a particular DRAM bank, at each stage one or two address bits are no longer relevant since from that point on the subset of possible targets is set. So we can remove that redundant address bit from storage and routing and save a small amount of area and power.

The earlier stages of the memory controller try to sort requests by various QoS properties. The latest version of this is (2021) https://patents.google.com/patent/US20230060508A1 *Memory Controller with Separate Transaction Table for Real Time Transactions*. I don't think the "Separate Transaction Table for Real Time" is really the central feature; rather it's how credits are handled. At every stage of transactions across the SoC credits are used, sometimes as flow-control, sometimes to limit bandwidth. The previous memory controller used credits on entry "into" the memory controller, to limit how many requests each different agent could enqueue but once that was done, I believe the only QoS control was in the ordering of requests as they flowed through the different memory controller queues. The separate real time queue comes with separate DRAM bandwidth credits, so in a way you can think of this as, even after real time requests have made it into the memory controller queues, they now also get a dedicated fraction of the DRAM bandwidth that they have first dibs on. (My guess is that with the increased performance of the GPUs in the M1 Pro or Max, stress tests showed that in certain conditions the GPU could hog almost all the DRAM bandwidth even in spite of the special treatment given to realtime requests, always moving them to the front of various queues.)

We also have (2022) https://patents.google.com/patent/US20230063772A1 *Memory Device Bandwidth Optimization* which describes how best to optimize for bandwidth given the new constraints on LPDDR5 (as opposed to LPDDR4 and earlier), specifically how to schedule and interleave the different maintenance operations required by the DRAM; https://patents.google.com/patent/US20230068494A1

Multi-Activation Techniques for Partial Write Operations, which (as I understand it) is to be about speeding up partial writes, again making use of LPDDR5 particulars; and https://patents.google.com/patent/US20230064187A1 *Communication Channels with both Shared and Independent Resources* which merges, in the memory controller some final stages that in the prior controller were independent hardware in a way that gives the same QoS results, but using less area and power.

(optimize NoC not just for bandwidth but also for energy; ie variable QoS policies)

We also see the usual on-going improvements to the SoC/NoC.
When we last left this, the essential ideas were (to simplify):
- transactions were marked with a QoS which established a priority at each router/arbitration point. High QoS items were processed first from the queue of transactions. In other words QoS mostly reduces latency.
- orthogonal to this, agents were given credits at a rate proportional to their allocated bandwidth, which limits them from exceeding a target bandwidth

The obvious problem with the credit scheme as I described it is consider eg a (very simplified) case where we give agent A 1000 credits/sec (so that it can use a maximum of say 1MB/sec) and agent B 2000 credits/sec, with the maximum throughput of the A+B link being 3MB/s. What happens when agent B is not active? Agent A is still throttled to 1MB/s, which is clearly sub-optimal.
So the next step is we not only supply credits at a certain rate, we also track the rate at which credits are being used up. The difference (more or less) gives us excess bandwidth, and so we can allocate excess credits to a “shared credit” pool. We now have another lever of policy, namely how to allocate the shared credits.
The simplest scheme that’s not to totally stupid is probably to allocate excess credits proportionally, based on each active agent’s baseline bandwidth (ie baseline number of credits). But if you think about it, we can do much better! Any agent that is not already using up its full allocation of credits probably doesn’t need more, so instead let’s allocate still proportionally, but now only diving up the pie between all agents that have been using up all the credits they are allocated.
Something like this is optimal for maximizing bandwidth, but at this point you can ask a secondary question: is *bandwidth* what I actually want to optimize? Maybe instead I should optimize for energy? For example suppose I have two active agents, A and B, but while B is non-idle it uses a lot more energy than A; in that case perhaps I should allocate essentially all the available excess bandwidth to B to get it to idle ASAP?
(Of course you may want to vary this dynamically – always energy for an iPhone, but optimize for bandwidth for a Mac that is plugged into the wall.)
(2020) https://patents.google.com/patent/US11436049B2 *Systems and methods to control bandwidth through shared transaction limits* is the latest version of this sort of NoC-wide co-ordination, now tweaked to allow the optimization of targets other than just bandwidth, by allowing the programming of a variety of different policies into the hardware that arbitrates the handing out of shared credits.

(better handle the large DRAM requests of some IO agents)

Here’s another interesting NoC-level idea. Some agents (think for example camera, or media decoder,

perhaps even GPU) naturally access long streams of data, and it's more efficient (both bandwidth and energy) if they can perform those large accesses, ideally as large as an entire DRAM page. But the most obvious and naive way to build a NoC is around cache-line sized transfers.

To improve things, you'd like to
- be able to indicate to each agent how large a maximum sized transfer it can use
- extend the "syntax" of the NoC so that an agent can indicate a single large-sized transfer (rather than multiple separate cache-line-sized transfers), and have this transfer preserved as a single transfer all the way to the memory controller. For writes to DRAM, or read data returned from DRAM, obviously you need a sequence of cache-line-sized data elements, but, again ideally, the collection of these elements should be treated as a single unit through routing and arbitration all the way to the memory controller.

The patent (2021) https://patents.google.com/patent/US20220334984A1 *Memory Fetch Granule* doesn't quite achieve these goals, but it's a first step in this direction. Rather than improving the NoC syntax as describes, we only update the individual agents (or more precisely, their abstraction units, the "control" blocks that sit between each agent and the NoC). These abstraction blocks accumulate successive cache-line-sized requests until the ideal memory fetch granule (MFG) for this agent is reached, and then they are sent as successive cache-block-sized requests over the NoC (but with some indication that they are tied together as a single MFG). Hopefully by dumping these rapidly into the NoC, they will stay together enough that the memory controller can do the right thing and ultimately handle them as a sequence of back to back transactions to a single DRAM page.

The idea seems to be to put most of the change into the abstraction units, with only minimal changes required to everything else (the individual agents, the NoC, and the memory controllers). Presumably as time goes by more of these changes will move into these other areas, reducing some of the bandwidth and energy overhead of multiple transactions down to a single transaction.

The immediate target for this seems to be the ANE which mostly deals with blocks of data much larger than a single cache line (either 64B or 128B) but you can imagine its value for many other clients – media, ISP, GPu, even occasional CPU use cases.
(Interestingly, this is something like an abstraction and generalization to all agents, of (2014) https://-patents.google.com/patent/US20150310900A1 *Request aggregation with opportunism* which attempts the same sort of aggregation of a stream of successive requests, but only for the Display Controller.)

The other, wilder, aspect of the patent is that it suggests each agent records not just a single MFG size, but multiple MFG sizes appropriate for different memory controllers. This doesn't make sense for multiple memory controllers all handling the same sort of DRAM (ie the Pro/Max/Ultra case); so it's yet another slight hint that Apple is thinking about designs with heterogeneous RAM of some sort (possibly Optane/Z-Flash/MRAM type storage, persistent, slower, but denser; possibly HBM though it's unclear what advantages that would provide over the current DRAM strategy; possibly CXL type DRAM connected over a PCIe type connection).

(*Provision for Ranks)*
(2023) https://patents.google.com/patent/US20240095194A1 *Read Arbiter Circuit with Dual Memory*

Rank Support and (2023) https://patents.google.com/patent/US20240094917A1 *Write Arbiter Circuit with Per-Rank Allocation Override Mode* form a remarkable pair.
Superficially this is yet another version of the various earlier memory controller patents: we have a trade-off between bandwidth and latency, and we choose which to prioritize at any given point based on various signals from clients and visible in the request stream (eg are we maintaining bandwidth guarantees? if not, then deprioritize latency).
The interest element is the reference to *rank*s, which is something I have seen in none of the earlier patents.

Recall that when we transition from reading to writing a DRAM there is a period of dead-time, so much of improving memory controller performance is trying to get as much reading or writing as possible done before forcing a turn, along with seeing if the turnaround dead-time can be overlapped with something else that has to be done anyway.
Now recall what ranks are. A rank is basically a set of DRAM chips that share the same wiring. The traditional example is the two sides of a DIMM module that share all the wires except for one or two that indicate which side of the module is being targeted. The think about ranks is that they give you more memory capacity (a channel with two ranks can now handle twice as many DRAM chips) in what may not be very much more space (especially if you are able to use the second side of a surface, like a DIMM) but there is a price to be paid for reusing wiring which is that, like the read/write transition, there is some dead time when switching addressing from one rank to another.
So the bulk of both patents is modifying the memory controller slightly to take this into account, in the same way as reads/write – gather requests by the targeted rank, if you are in high bandwidth mode then stay with the same rank as long as possible, if you are in low latency mode, then accept you may need to switch ranks at some point to service the other-rank requests.

All this might seem technical and uninteresting, but these are the first Apple Memory Controller patents I have seen that reference rank! The obvious implication is that Apple wants to create some products with a larger DRAM capacity (without the more extreme solutions of continually adding more memory controllers and channels, and thus bandwidth). Maybe we will see this as an option in a new Studio and Mac Pro? Maybe it's simply a Plan B, to be added to the SoC, but not turned into a product until really necessary? Maybe it's for internal use for Apple chips in Apple Data Warehouses?
The scheme we have already seen, using memory spines to extend the physical area available for DRAMs could use ranks; a simpler scheme which might be feasible might be to copy DIMMs and place a second rank on the reverse side of the SoC substrate, behind the current DRAM chips.

(Addressing Based on Multiples of 3)

Here's one that might initially stump you: (2023) https://patents.google.com/patent/US20240104017A1 *Routing Circuit for Computer Resource Topology*.
But it's easy to understand when you remember that the M3 Pro now comes with three, not four, SLC's and memory controllers. So a physical address is going to take the form of 3×(some power of 2 bytes). How do we rapidly extract what the multiple of three is and so decide whether to route the request to

SLC/memory controller 0, 1, or 2? If we place the factor of three at the top we could just mask out the top two bits and know that they only encode 0, 1, 2 not 3. But that requires allocating physical addresses at a very coarse granularity. We'd probably prefer to stripe the addresses (by page, or even by 128B) across the three SLC/memory controllers, which means the multiple of three is now somewhere in the middle bits. Now the problem is not so easy, is it!
We have something of the same problem again when we deal with some of the modern DRAM sizes like 12GB rather than 8 or 16GB.

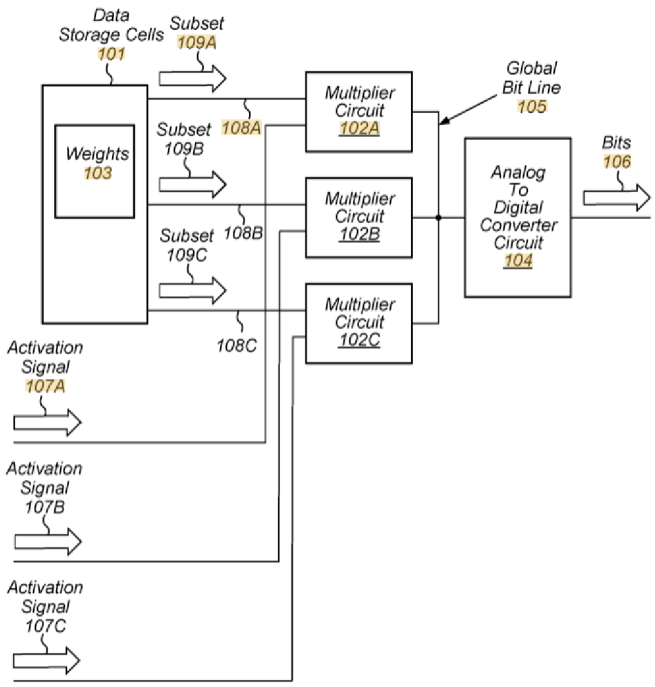
The patent describes a general solution, giving some examples for how to handle the case of 3 (obviously relevant to both the examples above) and the case of 15. Why 15? They don't say. But if you've learned anything from these volumes, it's that I'm a great fan of hashing to solve most problems! Certainly if you can hash cache addresses into say 15 sets [or 15×(power of 2) sets] rather than just a power of 2, you can remove the most common cache addressing/bank conflicts, which should help a fair amount of code. So if this were used at the L2 or SLC level (or even eg for the SRAM banks of the GPU) that would probably be advantageous.

PiM

(more sightings of the elusive, and possibly never-to-be-implemented PiM)

In the packaging chapter we discussed how the packaging of DRAM for the A15 and then A16 were both somewhat unexpected, and speculated that unexpected items we saw were related to PiM. This now seems unlikely (in that Apple has mentioned nothing about this and no-one has discovered anything relevant) so the extra items may just have been stiffening silicon to give the package strength. However Apple does seem to be interested in PiM as evidenced by the following patents.

We start with (2020) https://patents.google.com/patent/US20220156045A1 *Performing Multiple Bit Computation and Convolution in Memory* which is mostly about circuits, but the big idea is that, right next to memory storage we have "multipliers" which multiply data loaded from the memory with "activation signals" to generate an *analog* result on a single line. These analog results add together (as analog voltages) to be converted by an ADC to a digital output.
The "multiplication" is done bit by bit and so is less a multiplier than a choice of either zero voltage or a pass-through of the weight (converted by a DAC equivalent and multiplied by some power of two).



When I hear PiM (processing in memory) I tend to think of logic attached to DRAM, but I think that's the wrong model here. Logistically it might be difficult for Apple to get a DRAM vendor (which is so optimized for a particular process) to modify that process to add some side logic. So I think the idea is more that this is something like an SRAM on the SoC with some side computation attached. (A separate SRAM, or the SLC?)

The above patent is followed by (2021) https://patents.google.com/patent/US20220101914A1 *Memory Bit Cell for In-Memory Computation* which assumes the same basic idea, but suggests the "data storage" be a specific bit storage cell based on capacitors, which allows more analog circuitry in terms of transferring the (digital) stored weight into an (analog) voltage on the common line that is adding the products/-voltages together.

Finally we have a later patent (2021) https://patents.google.com/patent/US11694733B2 *Acceleration of in-memory-compute arrays*, which discusses the routing part of the above design, ie how to move activations relative to weights to achieve various common products or convolutions at minimal energy cost.

Perhaps this will replace the existing ANE? I don't know what sort of dynamic range or accuracy this tech can handle; but maybe it's good enough for all the most common use cases of ANE, with the GPU as backup for more demanding situations?

An interesting development is that the Mediatek Dimensity 9500 (announced Sept 2025) claims to have what Mediatek called CIM (Compute in Memory), apparently in the form of a large'ish SRAM array that seems to have some compute logic attached to SRAM cells. The numbers associated with this particular NPU, if they are being correctly interpreted, are very impressive for both performance and power. Which suggests that PIM may be about to move from being an academic idea to a genuine product.

Realtime CPU...

Here's one that's very strange!
We've send endless examples of ways to handle QoS: arbiters that respect QoS, QoS tagging for cache transactions like snoops, ways to avoid priority inversion, dynamically modified QoS for when CPUs are more sensitive to latency or bandwidth, etc etc. And yet it's still not enough!

The closest analog I can think of this in what we have seen elsewhere (check Volume 3 and Volume 6) is the DSIDs used by the GPU. The GPU DSID mechanism is a way to tag the use of certain resources so that cache lines associated with those resources can be treated a certain way (subject to quota in cache, preferentially retained or preferentially released etc). However making use of the mechanism relies quite a bit on how structured GPU use is – how resources tend to be large, are explicitly marked in Metal/MSL API uses, and how shaders tend to be small and used for just one task.
Is there a away we can bring some of that to the CPU?

That's partially what (2022) https://patents.google.com/patent/US11886340B1 *Real-time processing in computer systems* is about.
Suppose that we designate a specific range of physical address space as "real-time" address space. Then, via the right APIs and permissions, we can get data (and maybe at some point also instructions?) mapped from an arbitrary virtual address range into the physical pages within that real-time range. We can also (because all transactions beyond the CPU core operate in physical address space) mark some lines in the various caches (L1, L2, SLC) as reserved for realtime lines, and we can give NoC transactions to these addresses maximum priority.

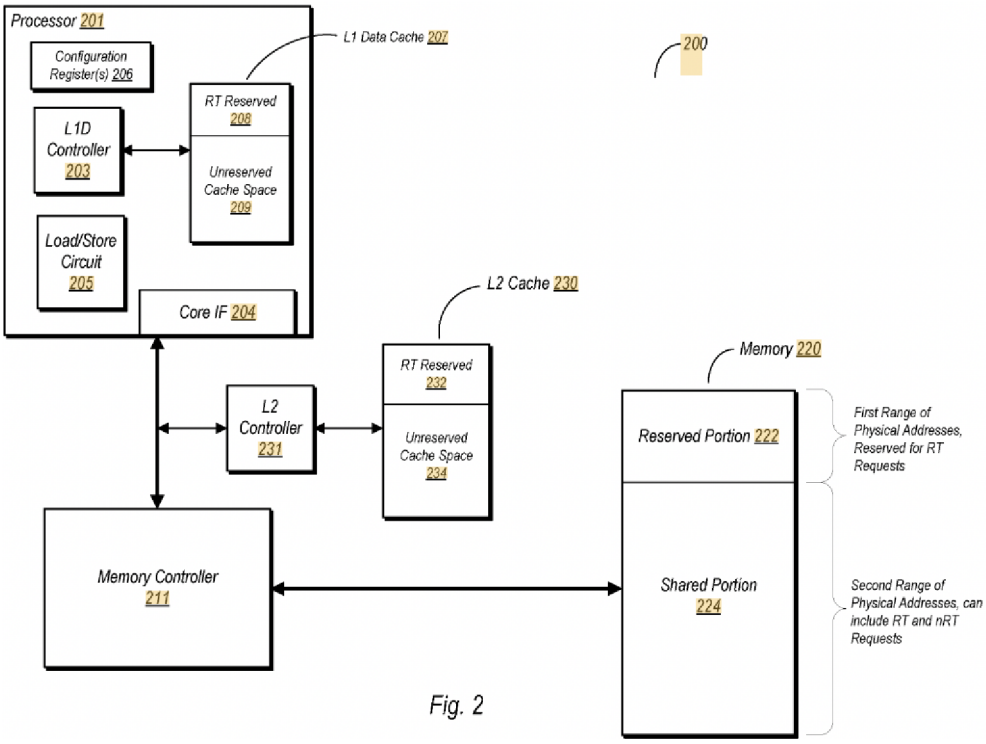


Fig. 2

What's all this for? Hmm.

Presumably it is for genuine CPU transactions; presumably the existing QoS/realtime machinery is good enough for media, graphics, camera, etc.

The obvious assumption is that this is somehow relevant to Vision Pro. The timing kinda works.

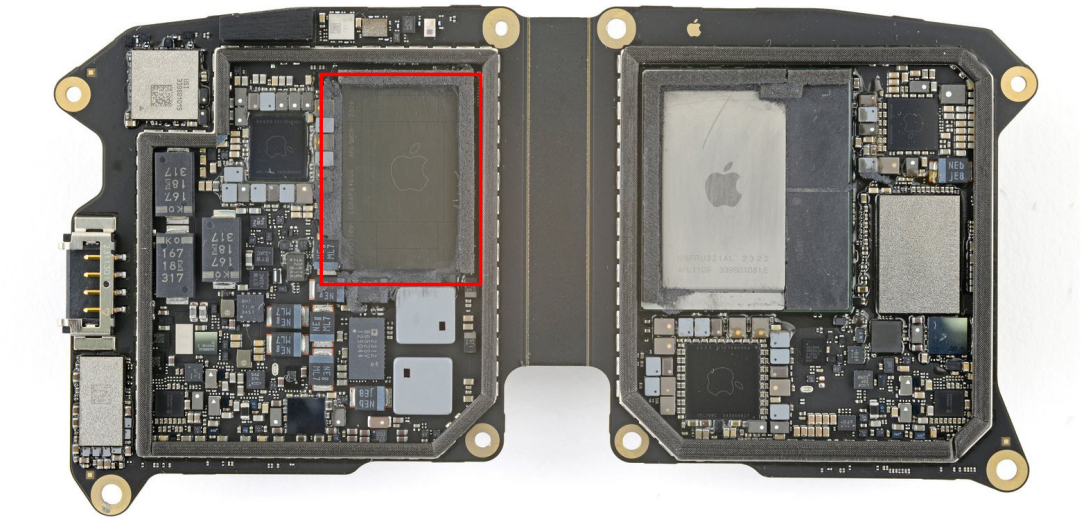
The patent seems to be describing something like an M2, not something like the R1 (insofar as we know what the R1 does). But truth is we have no real idea what the R1 looks like below the surface. The R1 package looks different from the M2 – but is it really?

If you look at the R1 surface shot, it seems to be made of chiplets, and presumably some of these correspond to DRAM. But the main R1 SoC, what is it?

Could the R1 be essentially a modified M2, not a completely different chip? Maybe start with the M2

design, *including this real-time support*, then remove whatever is not appropriate to the mission (media, ANE, ISP, Secure Enclave, some, maybe all, of the GPU, etc)? That would certainly fit with the way Apple is very disciplined in reusing IP rather than reinventing the wheel for every new product.

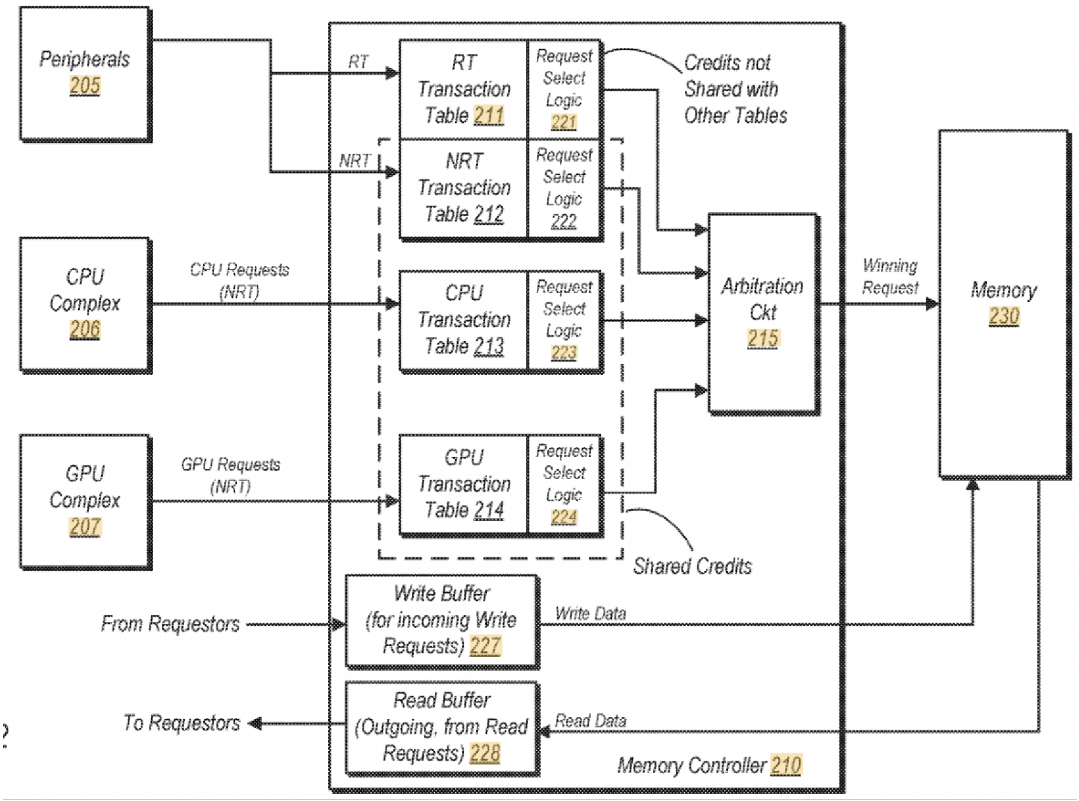
https://pbs.twimg.com/media/GFwQhB8bsAARJ3R?format=jpg&name=large



In the same sort of area is (2021) https://patents.google.com/patent/US11900146B2 *Memory controller with separate transaction table for real time transactions*, which does exactly what it says – in the

memory segregates real time transactions into a separate queue, and subjects them to appropriate arbitration (not described, but probably boils down to something like “use the previously described techniques to maximize memory bandwidth, interleaving RT and NRT transactions as opportunity allows, until the age of an RT transaction grows larger than some threshold at which point just switch to servicing the growing-old RT transaction”?

This patent (older than the previous patent for RT support in the CPU) talks only about RT peripherals, but presumably now RT CPU requests also get in on the action.



Cache

(zero content cache?)

https://patents.google.com/patent/US11442855B2 *Data pattern based cache management*

This seems like an extension of the earlier patent https://patents.google.com/patent/US11303478B2 that masked out zero bytes on NoC transfers. The idea is that the caches track common "data patterns" and when such a pattern needs to be transferred, the pattern index can be transferred rather than the data.

This would seem one more step on the way to a zero-content cache, giving us the energy saving side, but still not giving us the additional capacity side of such a cache.

HOWEVER look at the explanation associated with Fig 5. This seems to suggest that the system plays games with the address space so that certain addresses map into "pattern space" rather than physical memory space. The only way I can see this making sense is essentially having zero-content pages mapped in this way so that TLB lookups to such pages (at least until overwriting starts) don't require actual allocation (in physical memory or in caches). Which is nice, one of the zero-content tricks I suggested, but only increases capacity at the page level, not at the zero-line level :-(

Maybe this is all required as a workaround to someone else's zero-content cache patents, and will change once those expire?

(There may also be interesting debugging functionality available here which Apple may or may not expose. If "zero'd" pages are actually routed by the OS to a page of random noise or all 1s ("zero'd" pages should never be relied upon by user code to be zero) various, even exploitable, initialization bugs may be exposed?)

(more efficient remote atomics?)

(2021) https://patents.google.com/patent/US20220365881A1 *Memory Cache with Partial Cache Line Valid States*

Consider the SLC as a cache. The first thing that's unusual about it is that cache lines only requires states like Invalid, Modified, and Valid. States like Exclusive or Shared are about some "agent" *within* the cache (eg a CPU) modifying the data, but the SLC doesn't engage in that sort of thing. [At least not yet…]

Second point is that SLC cache lines are probably long. My guess is that they perhaps 512B long (this seems like a reasonable size for users like the GPU). You want to operate a cache like that as a sectored cache, with one tag (corresponding to address information) covering the entire line, but per-sector flags marking each sector as valid or not. For example if we split this into line into four 128B sectors, then when a CPU L2 read or wrote a single 128B line, we only need to modify that sector of the 512B line.

This means that a line lands up with, let's say, one invalid sector, one modified sector, and two valid sectors. Suppose now the GPU requests that line . The easy way to handle this is to write the modified sector back to DRAM, then load the entire 512B from DRAM. But obviously that's not ideal! Much better is to load from DRAM the 128B that's invalid, merge it with the 384B of the line that are valid, and send that to the GPU. It's optional (but generally makes sense) to write back into the SLC the 128B that was loaded from the DRAM.

OK, this should all make sense and seem reasonable. But with this set of ideas, can we do anything more interesting? The patent suggests that we track data validity at a finer granularity. Let's assume

(for the sake of argument) that we want to track data at the granularity of 8B, so that a 512B line holds 64=2×32 such 8B units. Then in principle things like strange IO transactions (perhaps even some of the weirder CPU atomic transactions?) could modify data at this granularity. If we were tracking data at this fine granularity we could still know when values have been modified or are invalid, and thus pull data in from DRAM as necessary, but hopefully some of the time we could reduce the amount of data we need to move from DRAM. Likewise these unusual transactions that have written unusual values may want to read back these unusual values, and we can serve just those values from the cache line (whereas if we wrote 8B within a 128B granule without fine tracking, then this request would force a read from DRAM to get the full 128B, so that we can merge that 8B into the rest of the 128B).

This should all make sense in theory, but it seems to requires a lot of extra flags in the SLC tags, not just bits cover each of the 128B sectors, but say 2 bits covering every 8B bytes, so an additional 64×2b per tag, which seems kinda excessive for what's probably a fairly rare use case. And that's where the true genius of the patent comes in, even though the patent itself downplays this part! In fact what we do is

- split the cache line into two halves, upper and lower. Each half gets three bits of tag, so eight states are available. Two of these are technical states to keep track of things when waiting for data to arrive. Some describe states where the entire half-line is invalid or invalid. But the interesting states say that a half-line is "partial" meaning that at least one granule (some small fragment, like 8B) within the line is invalid. How do we know which granule(s) it is? Well, we know that at least 8B of the line is invalid. So let's slide the line left by 8B, squeezing out the first invalid 8B (whose value we don't care about). This gives us 64b of free bits at the left of the line, which we can set to, eg 32bits of valid/invalid and 32 bits of modified/unmodified. From this mask we can figure out, when necessary, how to reconstitute the line (by shifting appropriately) and what bytes, if necessary, we need to read from DRAM to fill in the invalid slots. Cute, huh!

I'm somewhat guessing the exact numbers used by Apple, but they give the flavor of the patent. Is it worth doing? It's not quite clear to me. My *guess* is that the real win here is for remote atomics. This scheme allows remote atomics to be executed in the SLC in a way that (more or less) isolates each atomic, limited to its 8B, without causing what might look like nasty 512B- or 128B-wide modifications involves moving lots of data back and forth every time a CPU updates a shared counter or whatever. There's a similar patent a month earlier, (2021) https://patents.google.com/patent/US20220321490A1, *Data Encoding and Packet Sharing in a Parallel Communication Interface* which likewise allows you to drop some bytes from a wide NoC transaction using a mask (encoded into the transaction in much the same way, though the details differ because the concern is less about saving storage bits and more about transmitting as few changing bits as possible); and that patent likewise only seems relevant to the rare narrow (smaller than a cache-line) transactions, like either some IO control, or remote atomics.

(lower power for L1I way prediction)

The issue of way prediction has been somewhat confused throughout this exploration! Experimentally there seems to be no way prediction for the L1D (or whatever they are doing seems to be a perfect predictor even when one tries to generate a random stream of accesses!).

As far as the patents go, there have been attempts to save power in various ways on the I-side by things like not bothering with way prediction when the next loads from an I-line remain in the same I-line as the previous cycle, and by moving the way prediction into the Next Fetch predictor as a single lookup. To complement these we now have (2021) https://patents.google.com/patent/US11487667B1 *Prediction confirmation for cache subsystem*.

Consider a standard way predictor. From the address of interest we generate a hash giving the set of interest, and we feed that setID into a predictor which gives us the predicted way. We then look at the way of interest, compare the tag to the address of interest, and either match (the way is correct) or not. Now suppose we access that same address a second time. The setID will be the same, as will the predicted way, and we will again match tags which will succeed. Can we avoid this second (and subsequent) tag matches?

The insight is that
- until something goes wrong, the way predictor will generate the same result each time for a given setID
- it's cheaper to read the tag associated with a way from the way prediction storage (along with reading the way prediction itself) than to read it from the cache SRAM
- we can record that a given wayID matches a correct prediction in the cache line as one more "valid" bit.

Now consider various cases
- the normal case is we lookup again in the same line. The tag from the address matches the tag in the way prediction SRAM so we know the prediction is (probably...) correct and read the line directly without testing the line's tag. We have saved some energy (and some time, though probably not enough to matter.)
- the abnormal case is that something about the line has changed (for some reason it was invalidated or evicted, a TLB mapping changed, whatever). In that case we need a way to know that the way prediction is invalid because the line is no longer appropriate.

How can we handle this second case? The patent's answer is to have an extra bit associated with every line that can be flipped in the case that the line is no longer "appropriate to the way predictor", so this bit will be flipped under conditions like line eviction or TLB changes. So basically the flow now is
- get the predicted way and see that the (way predictors version of the tag matches the address)
- read the line (including the "way prediction is allowed" bit)
- if the bit is set, accept the data; if not, treat it as you would a way misprediction and check all the lines.
- so we've basically replaced transporting and comparing a tag (multiple bits, perhaps 40 or so) with a single bit.

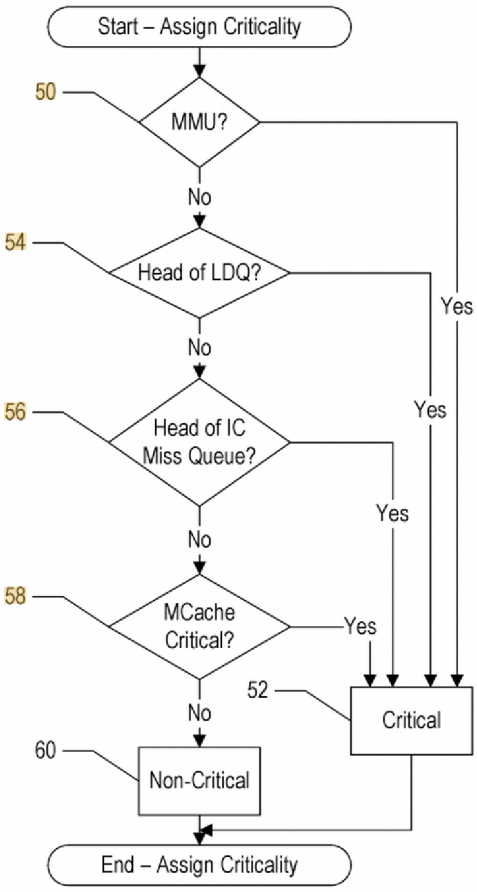
Whether it's worth doing depends on how often we have this situation where the same way of a set is

hit repeatedly. This seems plausible for I-caches (not exactly for loops, in that small loops should be captured by the loop buffer machinery, but things like repeatedly calling the same function from different places); the stats would have to show whether it's worth doing for other caches like L1D or L2; perhaps not. What about more unusual caches, like texture caches or vertex caches? I've no idea what the access patterns for those look like! but maybe?

(special handling of "critical" lines)

I've mentioned in a few places that one aspect of the academic state of the art for caches is to handle different types of lines differently, so that more critical lines (ie lines that will cause more of a slow-down if they are not present in cache) are handled appropriately; for example making it more difficult to replace I-lines than D-lines in the L2. With (2022) https://patents.google.com/patent/US20230060225A1 *Mitigating Retention of Previously-Critical Cache Lines* Apple implements a more sophisticated version of this idea.

What defines a critical line?



A line that defines page table leaf entries is considered critical, as are the data and instruction cache heads as described (I'm guessing that being head of the queue means that you weren't easily

prefetched, and also can't hide your load time in the shadow of an earlier load the system is already waiting for, so should be "locked' more aggressively in the L2). The last case is, I think, page table non-leaf entries, which will be stored in the memory (table walker) cache rather than in the L2. Since they go to a separate cache, I'm not sure why they're treated specially.

The elements include:
- lines in the L2 have an additional field that describes how "critical" they are. This field moves around with the line as it travels down to L1 or out to SLC, only being lost when the line reverts to DRAM. In the SLC the existing DSID (data set ID field, mainly used by the GPU) is overloaded for this purpose.

- it's unclear whether the criticality line is used while in L1; perhaps not, or perhaps only when the line is inserted into the cache.

- the criticality field is used in various ways at different cache levels, but always with the idea that the cache tries harder to retain more critical lines.
You can think of multiple ways to handle this, which usually turn into an issue of which cache line to replace.
Consider the L1 cache. If a new line comes in, which of the 8 lines in a set should it replace? Options include
+ random. Easy to implement but seems unlikely to be optimal.
+ random-MRU. This only requires us to use a single bit to mark the MRU line, then make sure our randomness does not choose that line.
+ LRU sounds good in theory, but tracking the LRU line is not obvious. Schemes like recording the exact order in which lines have been used, then updating all the values on each new line access, require a lot of energy. You can approximate this through schemes like https://en.wikipedia.org/wiki/Pseudo-LRU
I'm not aware of anything more sophisticated than pseudo-LRU variants. Apple is definitely doing something at least pseudo-LRU-like, possibly fully LRU tracking.

Now consider a new line comes into the L1 cache. Should it be marked as MRU? Obviously it *is* the most recently used line, but it's also the case that we frequently use data in a streaming fashion, so that this line might be used once and never again. One possibility is we use whatever hints we can (perhaps the line was loaded using a non-Temporal load? Or came from the stride predictor?). Perhaps if we have high confidence that a line is streaming, we should mark it (however we can using our pseudo-LRU technology) as the least recently used, or second least recently used line? Give it one chance to be reused soon, otherwise it's the optimal candidate for replacement.
Then for "average" lines we can mark those as maybe not the most recently used by the 3rd or 4th most recently used; while critical lines are marked as most recently used on placement in the cache. This would be one way to try to "lock" critical lines longer into the L1. Another way to do it might be every time we would downgrade a critical line away from MRU (because we have accessed some other line) instead of definitely performing the demotion, however that is done (again depends on the details of our pseudo-LRU technology) we make this probabilistic. 50% chance the critical line slides down one

slot, 50% chance it stays in the same place.

We can play similar games with the L2. For the L2 often something like random-MRU is a good enough solution. On the one hand there's less time pressure, so we can use a more sophisticated algorithm; on the other hand there are many more ways to track and we don't have perfect info as to what lines should be retained. Even pseudo-LRU may be more effort than it's worth? It may even be more useful, instead of working on a better L2 victim algorithm, to devote that area and logic to a "dead-line predictor"; if we can often mark lines as "probably dead", then we have a good pool of first choice lines to evict...
2014 https://www.lume.ufrgs.br/bitstream/handle/10183/96062/000918761.pdf?sequence=1 *Increasing energy efficiency of processor caches via line usage predictors* has an overview of various ideas in this space.
However if we *do* have to evict a good line from our L2 (or SLC), again we can use randomness to try to prioritize critical lines. For example when the random eviction generator chooses a line, if that line is critical we run the random eviction generator a second time. Once again we're not striving for perfection, just trying to introduce some friction that makes it a little harder to evict critical lines.

Given all these ideas and possibilities, the patent seems to suggest
- the L2 uses a strict LRU scheme, but then uses the ideas suggested above to place new lines in an "appropriate" place in the LRU ordering, to bias the LRU ordering to retain some lines (eg critical) and not hold strongly onto other lines (non-temporal, prefetched but not yet referenced), and to choose a line to evict
- the L1 maintains the criticality setting of a line, but there's no description of how that criticality is used by the L1, or any hint of how the L1 makes these line-by-line decisions.

- the academic literature generally suggest a static assignment of criticality, something like I-lines and page cache lines being marked critical. (I-lines because the CPU soon grinds to a halt if waiting on instructions; page cache lines because a single TLB miss can affect a large number of subsequent loads.)
The Apple scheme is more general. When a request is sent from L1 to L2, the request is accompanied by various data which is used by the L2 to assign a criticality to the line. These data include whether the request will service a TLB miss, or whether it represents a request at the head of the queue of I-cache or the queue of load requests. (In each case the point is that the line will probably land up benefiting many instructions, not just one waiting load.)

An obvious (but not mentioned) additional way to use the fact of criticality is to give the request maximum priority QoS when it leaves the L2 for servicing by some other cache, the SLC, or DRAM...
There are then various rules for how the criticality is updated over time, and what to do when cache capacity is reduced (eg a portion of L2 is put to sleep, or the SLC has to give up some capacity to increased GPU activity).

One thing you may not have thought of is that threads may move to a different cluster. To deal with this, the cluster monitors various indicators that things have changed (included reduced hits rates for the critical lines, or increased snoops from another cluster hitting in this cluster) and if it's con-

cluded that a thread has moved, then the system enters what I assume is a temporary period (some number of cycles) during which criticality of lines is ignored, so that lines can age out of the cache as normal rather than holding onto them.

It's not clear to me that the above scheme is fully optimal. Snoop monitoring will tell us that a thread has moved to a different cache, and presumably the line will get allocated criticality in the remote L2 by the demand requests of the remote core. On the other hand, reduced hit rates might suggest that the thread has been killed, and that it makes sense not just to ignore the criticality bit but to clear it? There seems scope for some simulations to suggest slightly different behaviors in these two cases?

There is also one glaring case that is missing from the above classification!
Consider a line that comes into the cache and that contains some number of data pointers in its contents. Such a line may well hold one or more nodes of some sort of list of pointer-based data structure. We know that Apple is already detecting such lines from the Content-Directed Prefetch patent, https://-patents.google.com/patent/US9886385B1. It seems worth making an effort to retain such lines in the L2 given that these types of data structures are notoriously difficult to prefetch. Maybe the Content-Directed Prefetcher does a good enough job to make this moot, but I'd hope Apple has run a simulation testing this hypothesis!

This all represents implementation of one large strain of academic thought on how to get more value out of a cache.
Not yet addressed is dead block prediction (so that lines unlikely to be reused soon are the first to be dropped from the cache; kinda the flip side of criticality prediction). An alternative way to think of dead block prediction, which fits somewhat into this topic and Apple's framework, is the handling of "write-only" lines. Improving cache performance is all about detecting some sort of general regularity in cache usage which is fairly easily tracked and exploited. One such regularity is that
- we care about read performance much more than write performance AND
- it turns out that there are many cache lines that are written to but then never read from until either much later, or read in some non-CPU-relevant way (like you write the data then at some point the file system moves it to storage)
It would be ideal if you could exploit this in some way, along the lines of transferring write-only lines, once they are case out of L1, directly to DRAM, without bothering to store them in L2 or even SLC. This sort of idea (done correctly) results in slightly higher *write* memory traffic (because sometimes a line you write directly to DRAM is then overwritten and, if it had been present in L2 or SLC, we would not have needed an overwrite to DRAM); but this higher write traffic is balanced by even fewer reads from DRAM (because the L2 and SLC are able to hold a lot more read cache lines). So it's something very rare – both a performance win and an energy win!
(2014) https://people.inf.ethz.ch/omutlu/pub/read-write-disparity-in-caches_hpca14.pdf *Improving Cache Performance by Exploiting Read-Write Disparity* discusses various ways to exploit this observation, though I suspect one may be able to do even better than their observations and suggestions.

Also noteworthy is that some of these ideas (in particular the special treatment of lines that come from reading the page tables) might also be relevant (though with differences in the details) for the GPU. In fact we will see this to be the case in our GPU investigation, though the details differ.

(tracking cache thrashing)

(2022) https://patents.google.com/patent/US11886354B1 *Cache thrash detection* is maddeningly light on details, but suggests a lot.
The previous patent suggested special handling of certain lines of particular importance, and was mainly geared towards the L2. This patent is more about handling default lines, and is more geared towards L1.

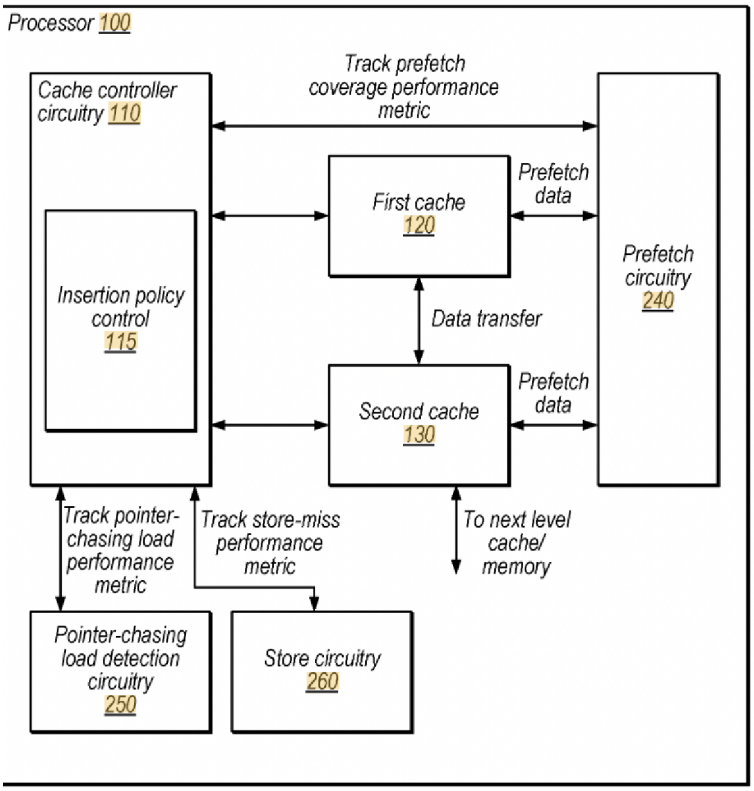
The paper (2007) https://www.cs.cmu.edu/afs/cs/academic/class/15740-f18/www/papers/isca07-qureshi-dip.pdf *Adaptive Insertion Policies for High Performance Caching* forms the background to the patent. Suppose, for example, that we are working on a dataset that is 1MB in size. We will find ourselves continually loading into our 128kB L1 cache a line, which removes a previous line, and which is probably not touched again for some time. So we continually pull in the entire 1MB. Can we do better?
One option is to effectively lock as large a fraction as possible of the data set in L1, so at least that gets reused, and then just accept the rest have to keep being reloaded. We have seen that a policy like this is used by the SLC in some circumstances, and then you try to optimize by spreading out the "locked" data in as even a fashion as possible. But CPU datasets are often accessed in a less predictable, less even way that GPU or Display data sets.
Another way to look at this is to say that if we are going to effectively lock data in the cache, we should lock the most useful (ie most reused) data, and this leads to the following thinking:
- When we mark a newly loaded line as MRU, that means it effectively has seven "chances" to be re-accessed (and again moved to MRU) by the stream of loads directed to this cache set. If it repeatedly fails to be useful, then eventually it falls to LRU, and gets demoted.
- Why are we being so generous in giving this line repeated chances? If conditions are such that we are cache thrashing (ie most of our loads are missing in the cache) then we are being too generous, assuming the average line loaded will be reused when it's clear that in fact the average line is not being reused.
- So under thrashing conditions, why not place default lines as LRU? This means we access them once to get whatever the load was that brought them into the cache, but our default assumption now is that the line will not be reused, so if we load a new line into this set, it will be this current line that is replaced. A policy like this effectively sets aside one way of the eight ways of each cache to hold streaming data, and ideally $\frac{7}{8}$ of the L1 will hold reused useful data, while the large ephemeral data set will stream through the designated "streaming way" not affecting the other $\frac{7}{8}$.

So that's the background. Before we discuss the Apple patent, let's think about the above. The paper is an easy read, and it suggests various options for when to flip between MRU and LRU. But it suggests all this in the context of L2, not L1. Why? It's obvious if you think a little. Suppose I am handling my streaming data in the most efficient way possible, just walking along a large array reading 16 bytes (one NEON

vector) at a time. This will generate four successive loads against my line before I am done with it... Even if I start with my line placed in LRU, it will be reaccessed to move it to MRU. I can try to use details like doing a paired NEON load or even the fancy weirder 3x or 4x NEON loads with rearrangement, but those are all cracked, and will look to the cache like multiple loads. The L2 does in fact see a streaming load as a one-time load to L1; but the L1 does not see a streaming load as a one-time load, more as a short cluster of loads then nothing. So you can't get what you want in the L1 by playing games with LRU ordering.

What might work in the L1 (but I have never seen anyone suggest this) is tracking per-line access counts. Something like increment a counter every time a line is accessed, every so often (100,000 cycles or whatever) shift all the counts by two, and replace lines based not on LRU but on something like "lowest access count of the way that is not the MRU line".

Anyway, so the ideas above are academia: reasonable ideas for L2, no good ideas for L1. What does Apple propose?



We have the sort of total control we have come to expect from Apple, a single controller that's tracking both L1 and L2 and collecting various metrics.

The particular metrics of interest include

- how often are prefetches useful? Each prefetch is marked with a bit, so we know if a particular load hits a prefetch line, and we can calculate which fraction of cache hits are served by prefetches. If this fraction falls too low and our overall cache hit ratios fall too low, then clearly prefetches are just wasting energy and making things worse. So if those metrics fire, we limit prefetching in some way. You could imagine a more sophisticated version of this where lines are tagged by whether they came from

stride prefetcher, region prefetcher, indirect data prefetcher, etc, and each of these was independently dialed up or down.
- the most critical data lines are probably lines accessed by pointer-chasing, so we have special machinery tracking those types of loads. If these types of loads seem to be hitting less than usual, prefetching is dialed up. This *may* be a specific version of what I suggested above, tracking the different types of prefetch differently so that the useful ones are encouraged, the less useful ones discouraged?
- the usual case with modern caches is to allocate L1 lines on write, so that even if only a few bytes are written to a line, the line is allocated in the L1. Under normal circumstances this is useful because the L1 is big enough, and we may often write again to the line later, or even overwrite. But if we are thrashing, so that we're constantly waiting on read data, maybe it makes more sense to just send the writes to L2 (we will try to accumulate them in a few buffers associated with the L1, on the off chance that the code does in fact write full lines of changes at a time).

So we have these various metrics being detected by the Cache Controller, and various details (how aggressive is prefetch?, do we treat L1 as write-through or write allocate?) There is one other case of interest which is when we are frequently missing in L1, but rarely missing in L2, so in other words our working set fits in L2 but not in L1.
This is the case I described above, and the patent basically says "yeah, we know this case exists, and we do something to handle it". But they don't tell us what!
They simply state that in that case you should switch to a Modified Insertion Policy, but the kinds of policies they list as possibilities are precisely the ones that (IMHO) aren't really appropriate for L1 usage.
Well, maybe in a year or two we'll get a patent on exactly what that modified L1 insertion policy looks like...

(variable latency cache)

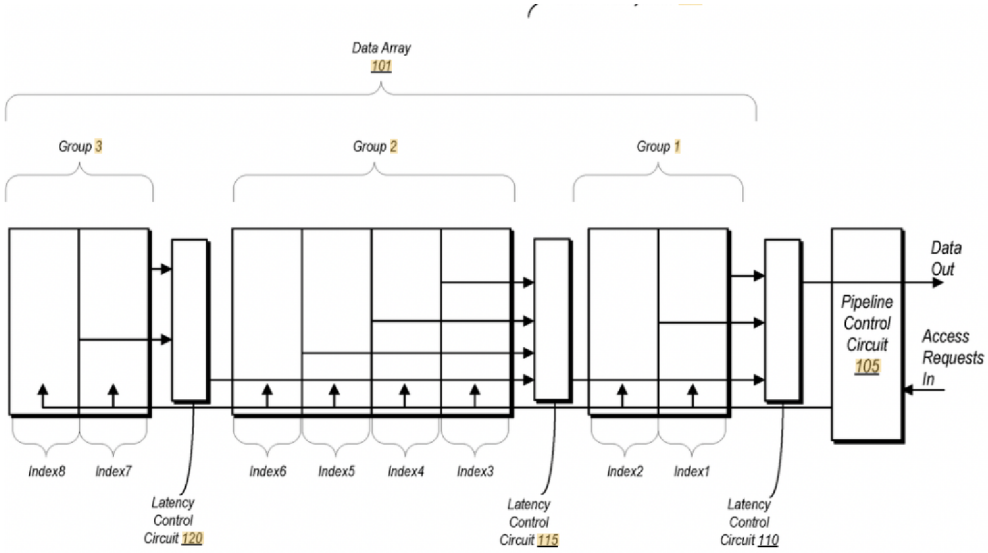
You might think that the M1 already has variable latency caches. Doesn't it seem like the latency to L2 or SLC is somewhat variable in almost any test?
But we have to be careful here. As far as a CPU is concerned, latency can become variable as soon as anything can cause a variation in timing, most likely because the bus/NoC is active for a cycle or two and we have to wait to gain access.
That's not the concern here, the concern is what happens once the request hits a cache. Even in that case there can be variability if, before the line is handed over to the Bus/NoC to be returned, various elements of cache coherence have to modify the line state, send out coherence message or whatever. Again, that's not our concern.
Our concern is the specific issue of time of travel. Once the cache gets physically large enough, it may take more than one cycle to send a request to the furthest parts of the cache. What should we do in response?

One possibility is to break up the cache into multiple separate pieces that each act independently. This is more or less the Intel solution, with each L3 nominally associated with a core (so that the amount of

L3 scales up with the number of cores) but each L3 can hold data from other cores (by hashing addresses and sending different hash bins to different L3's), so each L3 is never too large.
Apple could probably do that, if required, for the SLC, but it could be messy given how tightly each SLC block is integrated with its memory controller.
Another alternative is to simply run the SLC at the latency required for that longest lookup distance. Not great, but better than nothing (and probably better than shrinking the size of the SLC).

A third alternative is to run the cache at variable latency, so that accessing near blocks takes T1 cycles, mid blocks take T2 cycles, and far blocks take T3 cycles.
This sounds obvious and simple, but it's trickier than it first seems. The specific problem is that you can now land up with situations where data is being returned from say both a far block and a near block in the same cycle, so that the machinery that accepts a line from the SLC and moves it to the NoC has to be able to handle and buffer the worst case version of that situation with multiple lines arriving from multiple directions and all piling up at once, possibly repeated over multiple cycles. So the problem now becomes something like a NoC with routers and buffering required to move data between stages, and of course the opportunity for our old friend QoS to play a role.

If you're going to make this effort, you might as well go all in and fully optimize the system. If you can rely on the equivalent of a NoC and buffering to handle any sort of collisions and overflow, you can now run the cache more aggressively for maximum bandwidth; instead of occasionally having to limit sending out a new request because you don't want to overload the simple bus between segments of the cache, you can now run a pipelined series of requests, a new request sent out every cycle, and just rely on the NoC to handle and balance out whatever collisions may arise.
In terms of QoS you could imagine multiple ways to look at the issue, but Apple's viewpoint is they want to bound the cache latency to never larger than a certain possible maximum, while keeping the system as simple as possible. The effective consequence of this is that whenever there is a collision between say a line from a far block and a line from a near block, the far block gets priority. That way the far block stays within the latency bounds, while the near block may be delayed one cycle, but it will also be within latency bounds.
In terms of numbers, they suggest that reasonable values for current caches are the nearest blocks are one cycle away, the farthest blocks four cycles away.
(Note, if you look carefully at the diagram below, while it gives the essential idea, it is missing one arrow, from the Index3 block of cells to Latency Controller Circuit 115...)



The details of handling all this are covered by (2022) https://patents.google.com/patent/US11893241B1 *Variable hit latency cache.*

SLC cache quotas

When you see a patent titled (2022) https://patents.google.com/patent/US11914521B1 *Cache quota control* your response might be “so what’s new? SLC has been doing that for years”. And that’s true. What’s new is how the quotas are now implemented.

The obvious way of handling quotas is that an IP block, say the camera, requests a quota, is granted that quota, and uses it while the camera is active. But then what happens after the camera is no longer in use? In a way this is like the critical lines patent: in both cases you do in fact have a stage where you want the lines/quota to be locked down, but it’s also sub-optimal for those lines still to be in place (or still to have the perks of criticality, like being marked MRU) once circumstances change.

So how can the SLC do better? Conceptually what we want is something like not just allocation of quotas, but also some sort of monitoring of how aggressively each quota allocation is being used, so that entities not making aggressive use of their quota have it gradually shrunk. For example you might consider how much traffic there is by each entity; if an entity is infrequently accessing its quota for

whatever reason (maybe its local cache works well, maybe the entity is just not doing much aggressive memory work), then perhaps some of its quota should be granted to an entity engaged in more traffic? Once you have this idea, you can then start to consider various elaborations. You might track if the traffic to the SLC frequently misses in the SLC – if so, maybe the working set is just larger than can fit in SLC, and attempting to cache it is futile? Or if the traffic hits, does it mostly hit in the same repeated lines, or does it cover the whole quota?

This set of ideas informs the patents. Approximately what we do is track these various statistics over some epoch, and at the end of each epoch adjust quotas to reflect the patterns just seen in the previous epoch.

The patent also suggests these ideas are applicable to L1 or L2 caches. I don't see how this makes sense for L1, but maybe for L2 you might start with a quota allocation that equal per active core, and then adjust things based on these usage patterns?

CPU instruction Scheduling

(reduced power for pipeline flushing)

(2018) https://patents.google.com/patent/US11422821B1 *Age tracking for independent pipelines*

Consider what happens when a pipeline needs to be flushed (eg branch misprediction). Somehow we need to go through each instruction in the OoO portion of the CPU and either kill it (if it is younger than the mispredicting instruction) or allow it to proceed (if it is older). Perhaps the most obvious way to to do this is something like an N-bit timestamp attached to each instruction. This will work but has at least the following disadvantages

- we have to handle N-bit wraparound
- copying (and when necessary comparing) this timestamp uses energy

The patent describes an alternative that is complicated but uses less energy.

First there's some fiddling within the scheduling queue to ensure that a short offset relative to a base timestamp is used for the timestamp of an instruction within the scheduling queue; this saves area and energy while the instruction sits in the scheduling queue, but the full timestamp can be reconstructed as necessary.

Now imagine the following

- at the point where a set of instructions are to be simultaneously issued to different pipelines,
- consider the set of instructions that could cause either a mispredict or an exception
- for each such instruction create a bitmap of what's in the other pipelines based on either "older than me" or "younger than me".
- this bitmap is then attached to the instruction and referenced if flushing is required. From it we can work across all the pipelines to figure out what needs to be flushed.

This reduces the data to be propagated down an execution pipeline.

In fact, what's done is even more sparse than this. As far as I can tell, enough about the instruction and its timing is preserved in the scheduling queue through the initial stages of execution (at least long enough to determine the presence of a misspeculation or exception) that we only perform the above steps on demand! If flushing is required, we construct these bitvectors then use them to flush; but

under normal circumstances none of this work needs to be done and fewer bits have to be moved around :-)

(support for wider execution without wider register writeback)

This one is really surprising because, in a sense, it's something I've been hoping for but didn't imagine would be implemented, at least so soon:
(2021) https://patents.google.com/patent/US20230011446A1 *Writeback Hazard Elimination*.
Suppose that in a given cycle all six integer execution units generate a result, along with three load pair instructions. In theory this means we need to write back 6 + 2∗3=12 results in one cycle. Other execution patterns could be even worse. Thus it seems like we have to scale our register file to 12 (or more?) write ports, even though most cycles far fewer than this number are required. This is depressing in terms of area and power. Can we avoid it?

One option is to detect this situation and throttle (insert a wait cycle) into as many execution pipelines as required. This works, but we lose a cycle, and this will probably upset any sort of speculative scheduling forcing instruction replays and more lost cycles.

The patent suggests an alternative of a pool of temporary *local* storage such that the excess result writes happen to that temporary storage, from which they will later be drained to the main register file. The patent discusses nothing of the obvious complications in such a design (like routing data correctly from this temporary storage to a pending instruction, if required)!

The reason I like this patent is that one of the barriers to increasing performance is the variance in IPC from cycle to cycle; you can have cycles where only one or two instructions are able to execute followed by cycles where twelve or more instructions all become executable. If you have to design to some sort of mean IPC, then you have to throttle how many instructions you can execute during these peaks; but designs like this write-back pool allow the peaks to be a lot larger than the mean without having to clip them.

The patent also describes, as a side issue something that I don't think I mentioned before, that the Apple cores implement fill-forwarding.
Suppose we have a load that missed in L1. The load is sitting in the Load Queue waiting for its data. The natural thing to do is, more or less, wait until the data is stored in the L1 and then have the load replayed to access it. In fact the design is more aggressive. As soon as the cache interface unit knows that data is coming in, the LSU is informed of this, and the relevant load is speculatively scheduled. Then, as the data flows into the L1 cache, the appropriate bytes are also forwarded to the LSU. This presumably knocks two or three cycles off the latency.
Versions of this have been suggested since at least the 90s, so it's no surprise that Apple does it, but nice to see it confirmed.

Fusion

Consider a stream of dependent FADDs, something `FADD V2.4S, V2.4S, V4.4S`.
On M1 the latency of FADD is 3 cycles and frequency is 3GHz, so we expect, and see, 1B instructions per second of this stream.

For M3 the performance is 1.6B instructions per second, which is rather higher than you'd expect from the 4GHz frequency. Where does the extra boost come from?

IBM POWER10 has something called "back-to-back fusion" for FP/SIMD which is described in detail nowhere (that I could find!) but which appears to be the following. Suppose you detect that an instruction generating register R is followed by an instruction that uses R as input. You could "tie" these two instructions together as a kind of light-weight fusion, and send them together to an execution unit. Then only one initial cycle is required to gather the input registers; the second instruction can execute based on registers already present, and we can eliminate one cycle of register marshalling.
My guess is that this is what's being done by Apple, in the same way and for the same reason. It allows us to reduce the latency for dependent pairs of FADDs from 6 to 5 cycles, and if you do the math, that, together with the 4:3 frequency ratio, is just right to get us 1.6x higher throughput. Presumably this extends to many other pairs of dependent instructions of this form, though likely with some limitations.
Intel has something similar in Golden Cove and later generations.

This somewhat matches the pointer-chasing speedup already present in M1, where a load whose destination feeds into a subsequent load can execute in 3 cycles rather than 4.

(load-op fusion)

I've suggested elsewhere in these documents that instruction fusion is still not exploited as aggressively as it could be, that doing so optimally will require a decoupling queue between Decode (expanded out to say 11 or 12 instructions wide) and Allocate (still at say 10 instructions wide), and that one set of fusions that was missing from M1 was load-op and op-store fusions.
Well we don't have all of this yet, but we are on the way to load-op fusion! (2022) https://patents.google.com/patent/US12008369B1 *Load instruction fusion* discusses such a fusion, implemented by having a small ALU within the load/store unit which can execute the ALU op directly on the loaded value while it's still in the load/store unit.

There are some honestly weird elements associated with this patent.
One I applaud - rather than just the obvious fusions, like ADD or XOR, they also implement shift/rotate, and even some compares. Specifically, for compares, they don't fuse anything that touches flags, but they do fuse load with either CBZ (compare and branch if zero) or TBZ (test bit and branch if zero).
But being so ambitious raises a question: how do you pack all the required additional info (eg the target address of the CBZ or TBZ) into the fused instruction?
Rather than the obvious solution of just making instructions wider (which, who knows, maybe one day will come) for this implementation they do something unexpected! The patent points out that already a Load Pair instruction has some bits that define the second target register of the the pair, and those bits go unused for single load instructions. Those bits are repurposed to act as an index into a small table, the so-called *Fusion Table*, that holds various auxiliary info about the fused instruction, like this target address. Then when the load-branch instruction is executed, more or less simultaneously
- the load executes
- the load pipeline sends the index of the Fusion Table to a branch pipeline

- the branch pipeline calculates the target address (and all the other associated material that will later be used by the branch predictors)
- then a cycle or two later, on receiving the data, the load pipeline calculates the result of the CBZ or TBZ, sends it to the branch pipeline, and the branch pipeline proceeds as though it had performed the CBZ or TBZ itself.

This is a lot of extra work! But you realize that it's necessary once you remember that anything branch related ultimately also has to communicate with, to train/correct, the branch prediction hardware; and you don't want to have to replicate all that inside the load pipeline!

What about other cases, like load-add? Fortunately those are much simpler, more or less as you'd expect. The only real difference now is that the load-add has to also wait on the other source register of the add operation, and, when that is ready (and so the entire load-add is Issued) forwards the additional source register into the load pipeline.

In various ways this second set of load-ops is more ambitious than I expected.

First the load and op do not need to be adjacent! As long as they are in the same decode group (ie decoded in the same cycle) the fusion will happen. (What if they are separated by something like a branch? The patent does not discuss this case, but presumably in such cases the fusion won't happen?)

Secondly the obvious allowable fusion would be something like `load into rD` then `op rD, rD, rA`, ie we load into rD, then the op uses rD as one of its two inputs, and *overwrites* rD as the output. This means we don't have to allocate a second result register to the fused pair; the only result is the value that comes out of op. That simplifies register allocation and means the load pipeline only has to return one result, but it also means that we can't fuse if we want to use the result of the load into subsequent instructions beyond the immediate `op rD, rD, rA`. However, since the load pipeline already has the ability to return two results (again for the case of Load Pair) Apple decided not to require this limitation; the fused op can return its result in any register, and the load pipeline returns both the result of the load and the result of the op, in two different registers.

(Interestingly, at least when I checked in 2024, the most recent relevant LLVM commit, for M4, does not have any reference to a load-op fusion pattern. This may mean that the patent is not yet active in M4 and A17, or that public LLVM hasn't yet been tweaked to handle this case. Or maybe, given what we have discussed, Apple considers the machinery powerful enough [picks up nearby ops, they don't have to be adjacent; and they don't have to reuse a register] that compiler tweaking is unnecessary?)

There is another interesting implication to this patent.

Recall that M1 has an accelerator for pointer chasing, so that `LD Rd, Rs` followed by `LD Rd', Rd` takes only three cycles, not four, because after the first load pulls in Rd from the L1 cache, that `Rd` can immediately be used deep in the load-store unit as the address for the second load, without having to propagate outwards to the address generation unit at the "front" of the load-store. But this doesn't work if the pointers being chased are not aligned at the very start of the nodes being walked, in other words it fails if the second load looks like `LD Rd', Rd+8`, because to calculate the address in this case (even though it's a fairly simple calculation) requires moving `Rd` out of the core of the load-store

unit up to where an address calculation can be performed.
But this (potentially) changes if we have an ALU deep within the load-store unit, and set up to perform these sorts of calculations in less than a cycle. In that case it may be feasible to treat this pattern as something like `LD Rd, Rs;Add Rd, Rd, 8; LD Rd', Rd`, treat the address ad as fused to the earlier load and have it happen within the same cycle, so that a wider range of pointer chasing situations (with the pointer being chased not always at the 0 offset of the node data structure) might still be accelerated.

Instruction Prefetch

(first tentative adapting I-prefetch to a TAGE-like design)

We also see an update to instruction prefetching. (2021) https://patents.google.com/patent/US20230023860A1 *Multi-table Signature Prefetch*.
Recall that the constrains on I-prefetching are:
- we only care about long distance jumps (essentially function calls); local jumps like loops and goto's can be handled by a simple next line prefetcher
- we need to predict a function call quite a few cycles in advance
- we don't want to pollute the prefetch table with material based on speculative execution, so the table is only updated via retired instructions.

To do this we take an idea from TAGE and predict based not on local environment but on the path we took to get to where we are. Specifically Apple's first version of this calculates a "signature" of the path through the code (essentially a hash based on shifting+xor'ing the PC's of successive calls and returns). These signatures populate a table which is looked into every time the signature changes, and used to launch prefetches (details all in the volume 4 PDF).

The problem is that, like any hash, this is amenable to collisions. In other words two different paths, ending at two different functions, may have the same hashed path, so that any table trained on that path is constantly prefetching instructions for the wrong one of these two functions.
Another way to view this is that we have a tradeoff between a hash and table based on a short path, which can be trained rapidly, but has frequent collisions; or a hash and table based on a longer path, which is more accurate but takes a long time to train.

The solution suggested in this patent is to use a different aspect of TAGE, namely two (and perhaps later more than two?) tables tracking different history lengths. Suggestions for creating these different history lengths include using the same sort of XOR as before but shifting more bits out each time, or by omitting some of the call/return PCs in some way, eg every second one.)
[You might think of adding other control flow info, like the if/then tracking used by TAGE, but you probably don't want to do that. Most of that tracking is internal to a function, not relevant to the large-distance control flow that we want to track. We'll insert a lot of noise into the hash without adding much signal.]
Now we're in the same sort of situation as TAGE and operate in essentially the same way, training and using the short history table as long as it works well (the usual case) and shifting to training and using matches in the long history table for those cases that behave badly (ie continually alias) in the short history table.

The patent also suggests (but not give details; maybe this is one of these things that is still on the list to be done) that this same sort of technology could be used to inform the Fetch predictor under circumstances where Fetch prediction is difficult because we are jumping to a function that has not been visited in the recent past.

As far as I know all elements of this idea are original. The idea of using long-distance control flow to direct prefetching has been seen in papers like (2013) https://akolli.github.io/pubs/rdip-micro13.pdf *RDIP: Return-address-stack Directed Instruction Prefetching*; and the way the signature is used in a table of most recent signatures looks somewhat like (2020) https://webs.um.es/aros/papers/pdfs/aros-ipc20.pdf *The Entangling Instruction Prefetcher* (currently the state of the art in the academic world for "reasonable" storage budgets).
But the idea of using a path, and of then treating that path as variable length (so you can use TAGE-like ideas) seems to be wholly Apple. It's especially interesting that Seznec, who has been so aggressive in applying TAGE all over the place (at least also to indirect branches, to value prediction, and to load/store aliasing) never hit on this idea!

(relevance of Seznec's Omnipredictor to E-core?)
But Seznec has said something interestingly relevant to Apple – he has suggested that for a mid-range processor it makes sense to use common TAGE storage (with different front ends) to drive multiple branch predictors and the load/store aliasing predictor (and, we've seen, perhaps also even prefetchers), in (2018) https://inria.hal.science/hal-01888884v1/document *Cost Effective Speculation with the Omnipredictor.*

But wait!
Some time after I wrote the above, I found (2023) https://patents.google.com/patent/US12159142B1 *Managing table accesses for tagged geometric length (TAGE) load value prediction* which discusses exactly this, using the ideas of for load prediction (either address or loaded value).
Some details differ from Seznec, in particular while common history is used, separate tables are (apparently) used for branches and value prediction. Though perhaps at some point that might change and Apple uses common tables for an E-core?
Since the idea has been published, what's patented is a specific implementation detail.

The implementation is split into two sets of tables. One, prediction table, sits in the front end, able to provide values at around Rename time. The other, training table, sits in the Load/Store unit. This makes sense both on area grounds (the front end is getting pretty crowded with different tables!) and because presumably many loads will not result in a stable prediction pattern, and moving their information a long way to the front end is cheaper than using it right where it's acquired. Once a training entry is stable, it is moved to the front end.

So this is fine, but there is a problem. A VTAGE index, like any TAGE index, is a hash of a PC (maybe the

PC of the load/branch, maybe the PC of the Fetch Group that brought in the load/branch) together with some, possibly very long, like a thousand bits or more, history. That's a lot of material to move down the pipeline every cycle so that it's available to the training tables.
The solution is that the details of either the PC or the history are not important, what matters are the indexes (the hashes) that are used for the tables. Even if these were all distinct, their total length would still be substantially less than the full history length; and it may be possible to encode them in a way such that common bits across some indices do not have to be transmitted but can be calculated at the training tables.

Another cute trick is that the full value (address or load value) to be predicted is not stored in the training tables, only a shorter hash which can (usually...) detect that the same address or value is being hit each time. Then, essentially on the last training round, as the entry exceeds the required confidence, the exact value is extracted from the relevant load instruction and passed back to the prediction tables as part of the new prediction entry.

There's also a fair bit of buffering in the system. The problem to be solved is that it's common to have many back-to-back loads in a single Fetch Group (so a lot of traffic could be generated by the front end) but the back end is (still, as of the M4) set up to handle only three loads per cycle. So we need some intermediate storage to smooth between these two extremes.

All this does not replace the existing load value predictors, it augments them. So we have, at least
- the stride predictor that detects linear load address patterns
- the (easy) repeated load predictor which detects a repeated address or value all the time
- the (hard) VTAGE load predictor which detects a repeated address or value, but the repeat is dependent on prior execution history (as encoded in the history register)
- the system that (briefly) holds onto load values after a mispredicted branch, on the assumption that the correct path may perform the same load, perhaps after flow reconvergence.

There are some interesting elements to this that are still unclear. For example load pair is common, so does the system somehow track and enable these, either by cracking these to two entries or by allowing for a value that might be as long as 128b?
One way you could handle this is by having, say the main prediction tables support up to say 1024 prediction entries, but a prediction consists of a 10bit index into a separate *value table*. Then, based on expected stats, say 128 elements of this table are 128b wide, 256 elements are 64 wide, 512 elements are 32b wide, and the remainder 16b wide.

Branching

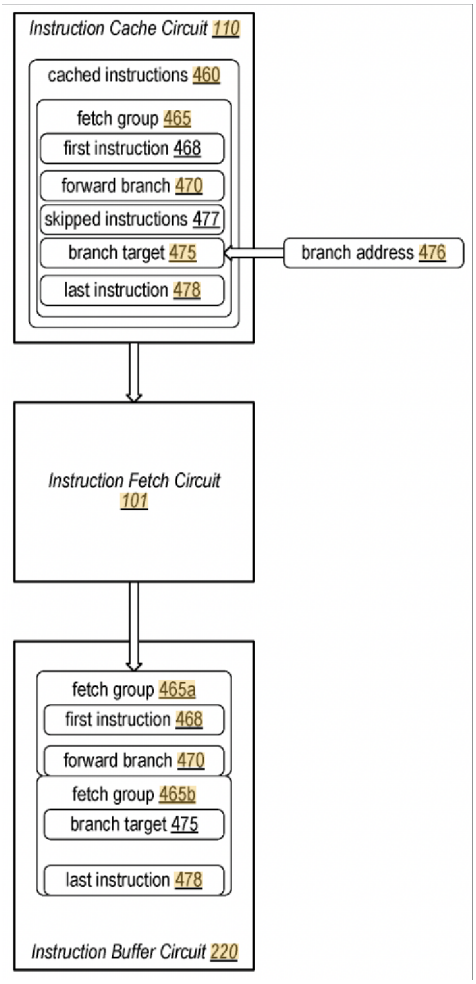
(remove the cost of short branches)

At an abstract level, what a compiler delivers can be thought as a stream of basic blocks (linear sequences of code, about five to six instructions long on average) separated by flow control instructions (calls, returns, unconditional and conditional branches).

The code “in execution” can be thought of as much the same, except that branch prediction transforms the conditional branches to look like either no-ops (not taken) or unconditional branches. So the code “in execution” still looks like a stream of traces (linear sequences of code, now about eight to ten instructions long on average) separated by jumps, ie changes in the PC.
The front-end of the machine, again at the most abstract level, is doing a whole lot prediction to try to ensure that *every cycle* it accurately predicts the next trace (the next linear sequence of eight to ten instructions) along with the trace length, the trace exit instructions [call, return, etc] and various other features. But there is a basic constraint here – the rest of the machine cannot run faster than instructions are fetched, and instructions, with this design, are fetched no faster than one trace per cycle. Many traces are longer than ten instructions, but that means that, for the averages to work, many are shorter than eight instructions...

So how can we bypass this limit? At some point we’ll have to start dealing with fetching two traces per cycle (which is “just” a slightly fancier version of prediction) but before we get there, how about getting better value from our existing machinery? Which gets us to the issue of short branches. I’ve talked about SFBs (short forward branches) before, and the great news is that Apple (doubtless as a result of seeing my brilliant insights!) has delivered! In fact they have delivered a set of ideas. Let’s start with the easiest one.
Consider some code like `if(){one instruction task}`. This will compile down to something that looks like `branch conditional; one instruction; rest of the code` and after branch prediction (if we predict the `if` is never true) will look like `branch+4; one instruction; rest of the code`.
The point, which I have stressed before, is that as the previous fetch engine is described, we have to handle skipping this one instruction by terminating fetch at the `branch+4`, then starting a new fetch at `rest of code`, even though it’s just one instruction that we want to avoid executing. Can’t we do better and just “cancel out” the `one instruction` treating everything else as a linear trace? That’s essentially what the new patent does, in a generalized fashion.

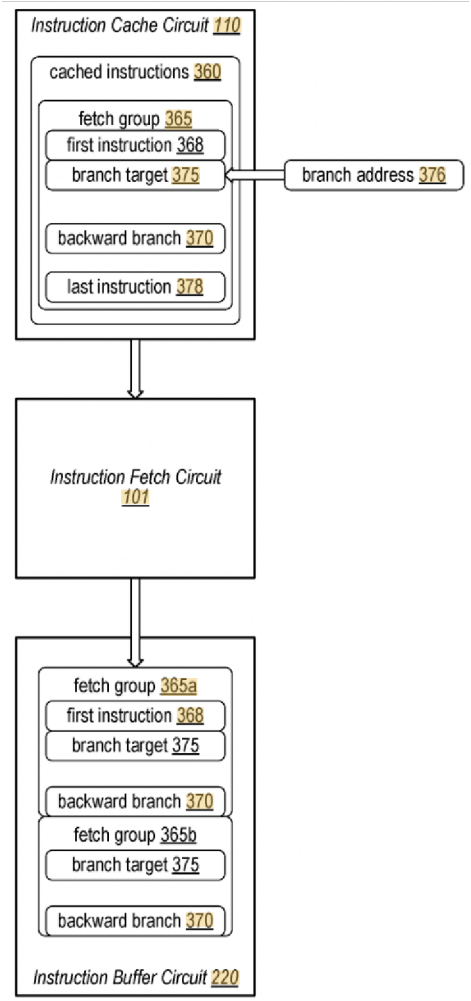
The picture makes it clear:


Instruction Cache Circuit 110
cached instructions 460
fetch group 465
first instruction 468
forward branch 470
skipped instructions 477
branch target 475
branch address 476
last instruction 478
Instruction Fetch Circuit 101
fetch group 465a
first instruction 468
forward branch 470
fetch group 465b
branch target 475
last instruction 478
Instruction Buffer Circuit 220

We predict that we will want to execute branch 470, jumping to target 475 which is in the same fetch group as 470 (ie the distance from the starting instruction 465 to 475 is less than the maximum number of instructions that can be fetched in one cycle. Apple does tell us what this is, but we know it can cross from one cache line to another, so it's probably 16 instructions.)
So rather than terminate the Fetch at branch 470 (the old model) we keep Fetch going till 475 and later instructions. Then between Fetch (pulling instructions from the I-cache) and placing the instructions in the Fetch Queue, we throw away [ie "skip over"] the instructions between 470 and 475.
This gives all the SFB (short forward branch) support I wanted, at essentially any length of branch where the branch and the target sit within the same possible Fetch Group (ie are within up to ~16 instructions of each other).
This is implemented as an additional field within the Fetch Prediction Table.
(As described this will handle various Short Forward Branches, eg also common `if-then-else` patterns. Another way to handle short `if-then` and `if-then-else` patterns to avoid a lost Fetch cycle is by converting them to predicated instructions. The two schemes are complementary. If the branch is *confidently predicted*, we should use branch prediction and instruction removal, as described above. If the branch is non-confidently predicted, we should convert it and the subsequent instructions to predicated instructions.)

However Apple does better than just the above scheme. Once you have this sort of machinery in place [a field in the Fetch Prediction Table to indicate short branches], how else can you use it to deal with the Fetch bottleneck of short traces? Apple describe two more ideas.

First is short backward branches. These occur when you have a short loop [short loop body, not necessarily only a few executions!] that the compiler did not unroll. We have seen various Apple patents for ways of handling loops (loop buffers, micro-op caches, L0 caches, unrolling a loop within a loop buffer) each one trying to optimize performance vs power. For example a loop buffer works if you don't need branch prediction within the loop, but it takes a few cycles to detect that it's worth copying the loop to the loop buffer.
The new scheme wants to optimize a specific type of loop (short, no branch prediction within the loop) right away. So we can use the same sort of machinery as we have already seen to "rewrite" the loop in the instruction fetch buffer. Now the diagram looks like

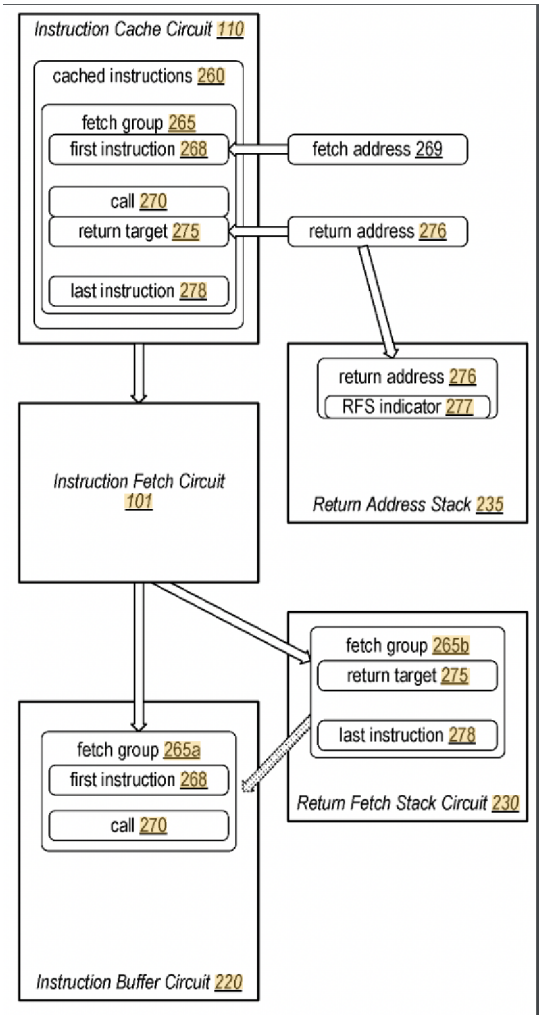

Instruction Cache Circuit 110
cached instructions 360
fetch group 365
first instruction 368
branch target 375
branch address 376
backward branch 370
last instruction 378
Instruction Fetch Circuit 101
fetch group 365a
first instruction 368
branch target 375
backward branch 370
fetch group 365b
branch target 375
backward branch 370
Instruction Buffer Circuit 220

We have now duplicated the loop body within the Fetch Queue. Essentially one Fetch has given us two iterations round the loop so, again we are not throttled by the fact that we can only jump the PC (ie execute one loop iteration) per cycle. The “normal” case we expect is that essentially the loop will executes multiple sequences of two loop bodies for a while then presumably move to an optimized loop buffer of some sort.

Once again this only kicks in once we see it being of value, again via a field in the Fetch Predictor Table, so presumably
- this will not kick in for strange cases like the body is executed only three times, and all this work is not worth doing
- possibly (since we have a field in the Fetch Predictor Table and might as well use it) we only ever run the two-way-unrolled loop from the Fetch buffer once, then we immediately move it to the Loop Buffer, since, if we can indicate in the Fetch Predictor Table that the Loop Buffer is valuable, we might as well start exploiting it ASAP.
- a two way unroll might seem to generate problems if the loop count is odd. The patent suggests that this is not as big a problem as you think. It will be caught by the fancy predictors (TAGE and friends) that execute a cycle or two after fetch, at which point the instruction stream will be edited again, so will not require a full machine flush. You could imagine adding another “loop count is odd” field to the Fetch Predictor Table to handle this, and probably that will be done if it’s overall an energy win.

Overall I’d view this not as a loop optimization (that stuff is already in place) but, like the SFB optimization, a way to avoid one Fetch cycle (either forwards or backwards) in a case where the data you will want to fetch is already easily available.

Finally we get the third use case, which looks more complicated, but isn’t really. Suppose we make a call to a short function. After the function exits, it returns to one instruction after the call site, and we execute those instructions that were just after the call site. Conceptually this is the same sort of problem as above – we can fetch, in the same Fetch execution, instructions that we know we will need after the call returns, rather than later executing a second Fetch. The problem is, how can we use this idea?


Instruction Cache Circuit 110
cached instructions 260
fetch group 265
first instruction 268
call 270
return target 275
last instruction 278
fetch address 269
return address 276
return address 276
RFS indicator 277
Return Address Stack 235
Instruction Fetch Circuit 101
fetch group 265b
return target 275
last instruction 278
Return Fetch Stack Circuit 230
fetch group 265a
first instruction 268
call 270
Instruction Buffer Circuit 220

We add to the Fetch unit a small amount of additional storage, the Return Fetch Stack, RFS, complementing the existing Return Address Stack.
Then the basic idea is at the execution of the call, we move the extra instructions after the call, instructions 275..278 to this Return Fetch Stack. Then at the point where the return is executed (as predicted by the return address on the Return Address Stack), that same return address on the Return Address Stack indicates that we can pull the first few instructions from the Return Fetch Stack.

Once again I think this is best viewed as avoiding a Fetch cycle (which can instead be used to load whatever instructions are predicted as *following* last instruction 278). The patent talks about this as a "Return Fetch **Stack**" but also talks in terms of this being pretty much a single entry "stack" for now, to be used (again based on data being tracked in the Fetch Predictor Table) only for cases of a short call, and one that presumably does not in turn call another function. But obviously if the idea makes sense in terms of saving energy and a Fetch cycle, it could be worth extending this Return Fetch Stack to be two or four or eight elements deep.
(You could argue that the sort of function call that is captured by this fetch sequence should have been inlined. That's true if inlining were free; but Apple is opposed to inlining except for hot loops because their data show that the invisible costs, specifically cache misses from the larger code footprint, are worse than the visible costs of the call/return overhead.)

So all three ideas ultimately have the same goal – to avoid the bottleneck where machine width is limited by short Fetch traces – by effectively extending those trace lengths in cases where it's easy and practical to do so. The patent is (2022) https://patents.google.com/patent/US20240028339A1 *Using a Next Fetch Predictor Circuit with Short Branches and Return Fetch Groups*.

A slightly different version of this Return Fetch idea is presented in (2022) https://patents.google.com/-patent/US11941401B1 *Instruction fetch using a return prediction circuit*, submitted at much the same time and by some of the same inventors.
Here's the problem.
The basic fetch loop we want, cycle after cycle is something like
- load an entry from the Fetch Predictor that looks like

    [target PC1] (how many instructions to fetch, type of taken branch that ends this trace, target PC2 of the taken branch)

- in this cycle:
+ send this data (target PC1, how many instructions to fetch) to I cache
+ send this data to the multi-cycle (but more accurate) branch predictors to make sure that the predictions of the taken branch (branch direction and possibly indirect branch target) are correct
+ use targetPC2 to index into the Fetch Predictor to get next entry to execute, which will look like

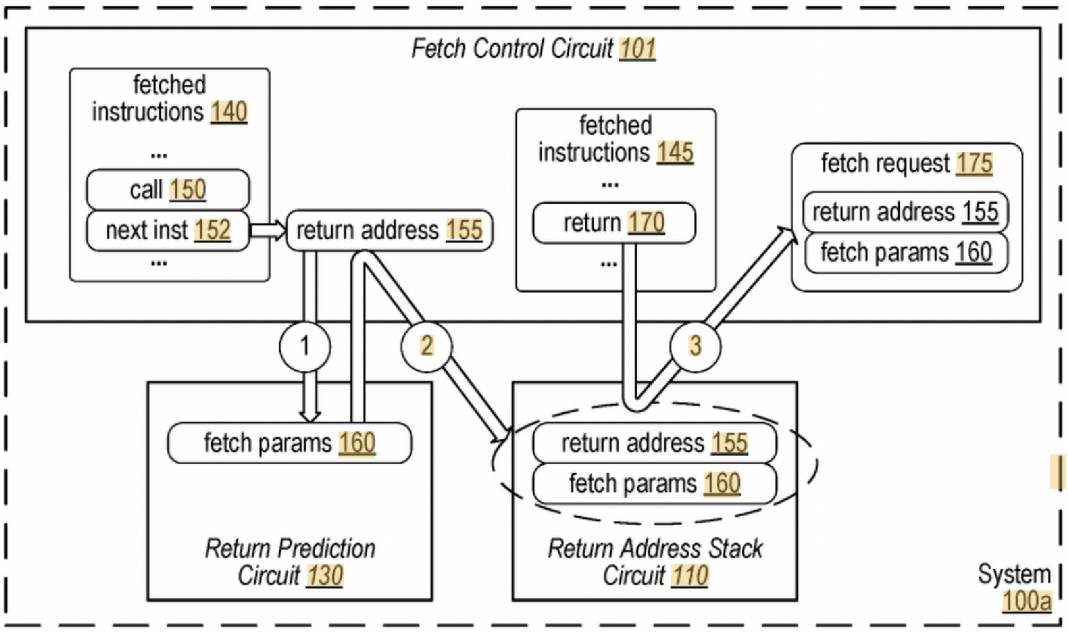
    [target PC2] (how many instructions to fetch, type of taken branch that ends this trace, target PC3 of the taken branch)

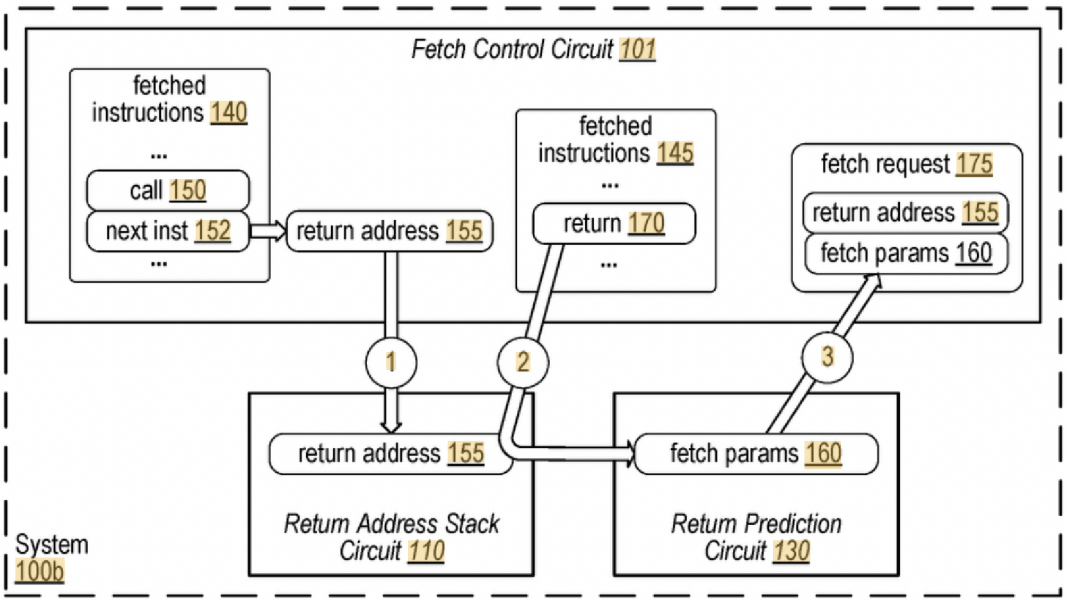
And so we repeat (ignoring details like how this table is built and modified, and how we handle mispredictions).

That's all fine, but there is one case where it's suboptimal, namely if the taken branch that ends a trace is a return.
The patent says that the trace following a return is currently not stored in the Fetch Predictor, and so after a return we have to proceed as though the trace is unknown, making worst case (more power) assumptions like loading the maximum possible Fetch length then discarding the excess instructions. Later we'll discuss why this might have been the case in older designs.

So can we fix this? We need a way to connect the desired Fetch Parameters to the return address in the return stack.
One solution, as described by the previous patent, is to place those instructions in the Return Stack, so that we can get them along with the return address. This means we can avoid both accessing the Fetch Predictor for those instructions, and also accessing the I-cache. That's optimal, but uses some extra storage. Is there an alternative that uses less storage?

The patent suggests two alternatives.
The first alternative is rather than storing the INSTRUCTIONS alongside the return address, we store the Fetch Predictor ENTRY alongside the return address.
This is scheme 100a below. At the point where the call is executed, we know where we are coming from and so can index into the Fetch Predictor to get the entry for the code that follows the call, ie instructions 152 and later. We can copy those Fetch Parameters 160, along with the return address, via step 2 into the Return Address Stack.





Alternatively we can use scheme 100b. This stores Return Fetch Parameters in a separate piece of storage that is indexed by the return address. So now the idea is on return we look up the return address (step 2 above) and then use that address as a tag/index to look into storage block 130 to find the Fetch Parameters.

The differences are (if I have the big picture correct)
- the first scheme uses a single block of Fetch Parameters storage, the same block as always. But on each call we have to copy an entry from this Prefetch Parameters storage into the Return Address stack
- the second scheme uses a second, different block of Fetch Parameters storage, purely for Fetch Parameters after a return, and we index into it based on the return address.

The second scheme may take one cycle longer because of the sequential lookups, first for the return address, then using the return address to index into storage 130. But it uses less energy because it isn't constantly copying data from the primary Prefetch Parameters storage onto the Return Address stack.

Finally an obvious question is why wasn't something like this part of the design from the beginning? I always assumed a scheme much like 100b was part of the M1. My *guess* is that it has to do with the complexity of the state machine controlling Fetch Prediction. This state machine, even for the M1, is extremely complex, having handle not just the usual case of one fetch prediction entry after another each cycle, but a variety of special cases (no relevant entry in the Fetch Predictor table, building the table, handling a mispredict, fixing an incorrect entry in the table, various special cases like TLB entries being destroyed [when an app ends execution and its address space is destroyed], etc etc). Given all this, throwing in an extra special case for Return, where the state machine has to "hiccup", delaying for one cycle while it reads data from another table (the Return Address Stack) may have been too much; easier, for a few years, to just treat that case as an error case until there's time to complicate the state machine even further?
If you have a better analysis/understanding (of this or any other design point!) let me know.

So big picture is that we have two alternative patents describing how to Fetch after a return, the first one makes the Fetch a little faster, the second saves a little energy. Why bother? Who knows what Apple will actually do, but one possibility is that we use the first option (higher performing, but uses more area) in the P-core and the second option, probably the 100b lower energy version, in the E-core?

(Tucked into this patent is one other cute little power saving gem. As described above, the Fetch predictor is going to route a request to the Branch Prediction machinery to confirm the Taken Branch that ends this Fetch trace. That Branch Prediction machinery is spread over multiple memory banks to hold all the storage. Rather than waking up all those memory banks [think of different ways of a set], the Fetch Predictor stores the appropriate bank holding data for this particular branch and sends it to the Branch Predictor, so only that particular bank needs to wake up. Like Way Prediction, for the Branch Predictor.)

AMX

(kinda sorta prefetching for AMX, but based on required addresses not speculated addresses)
(2021) https://patents.google.com/patent/US20230092898A1 *Coprocessor Prefetcher* is not what you think!
Consider how AMX works. Instructions are passed through the STORE pipeline (meaning a maximum of two per cycle can be issued) and sit in the store queue until they are shifted to the AMX unit. This has obvious implications in that only two instructions per cycle (apparently...) can be executed, but, as we have already pointed out, it's not quite that bad because
- multiple cores can (and perhaps will, for large tasks) be sending instructions to AMX, so in a sense up to eight instructions per cycle can be issued

- multiple instructions can be packed into a single L2 packet
- who knows how wide AMX dispatch is, but it's surely at least two and probably growing to three as vector operations become more common

So a consequence of all this, along with standard OoO execution and that AMX instructions issue only when the instructions become non-speculative, is that one expects a fairly deep queue of AMX instructions in the store queue. The patent suggests that this store queue is examined, and AMX data addresses in this store queue are "prefetched".

In the simplest version, this could mean moving them to L2 (either out of L1, or from SLC/DRAM or even another cluster). Simply by being present in L2 we avoid some latency. More ambitious would be to move the data from L2 into storage within AMX, so that when the load instruction is "executed" the data is already present within the AMX unit. This could be done with some dedicated local storage, or by making temporary use of a register that's not currently in use. (We'll see how this could work with the next patent.)

The patent stresses that, unlike normal prefetch, this is not speculative, we know the data will be used. I don't think that's 100% true; the AMX instructions in the store queue could be (and usually are) speculative (eg waiting for prior branches to resolve). But it's true enough for the system to work as it is meant to!

(derive some value from AMX registers attached to cores not currently using AMX)

(2021) https://patents.google.com/patent/US20230095072A1 *Coprocessor Register Renaming* takes up another point I have already suggested. We have four logical sets of registers in AMX for each of the four client CPUs. Surely it would be nice to find a way to use those additional physical registers if their clients are not using them?

The patent suggests a first small step along this path via the following steps:

- we know (because of the AMX SET/CLEAR instructions) when a CPU starts and stops using AMX, so we know when a set of AMX registers is free
- we can with some minor additional hardware (basically a remap table) then treat the pool of available registers like a pool of physical registers subject to allocation/renaming with the attendant benefits. We could also, for example, preload data (as per the previous patent) into one of these free physical registers and then execute a load instruction via renaming.

This is a cautious first step but one can see a path laid out to make the system more aggressive, switching from per-CPU dedicated registers to a single pool of common registers and on-demand allocation regardless of the client.

Consider what happens when an interrupt reaches a core. The first thing the core has to do is drain all pending operations, before it can switch to handling the interrupt, and that can take considerable time, especially if there are a large number of high latency (ie loads that have missed to DRAM) instructions queued up. https://patents.google.com/patent/US11556485B1 *Processor with reduced interrupt latency* tries to improve this situation.

The essential idea is that every load (more precisely every cache request) sent to the cache now has an additional field added which notes whether the request is "abandonable". Most requests are, including

prefetch requests, I-demand requests, and any loads that are speculative.
Then when an interrupt comes in, the cache is told to drop, as most convenient, all abandonable requests.
This allows the CPU to avoid having to wait for acknowledgement/handling of most cache requests (for example all speculative loads can be treated as though the speculation failed, rather than waiting for the data to be delivered to a register), so clearing the core can happen a lot faster. It's not specified exactly what the core does, and presumably it does what's most convenient, but you would hope that most requests will still be handled appropriately as regards interaction with the higher caches and DRAM. In other words, rather than just squelching a load or I-fetch, treat it as a prefetch and still send the request out; so that once the interrupt is handled and it's back to the primary task, the requested data is already available in cache.

The M3 Pro and Max implement a 6-core P-cluster. The optimal cluster size is always something of a tradeoff – you want the shared hardware (L2, L2 TLB, page walkers, LZ engine, AMX, etc) to be in a position where they are always in use (no "wasted" hardware) but never in so much use that cores are kept waiting because of other clients using this shared hardware. There's also the fact that, for better or worse, all cores in a cluster have to share the same frequency.
It's unclear what the optimal number of cores in a cluster should be, weighing all these facts, but Apple seems to have concluded that 6 is better than 4 (or is at least worth trying). We'll see in a year or two if the M Pro moves up to a cluster of 8 cores, or drops to two clusters each of 4 cores.
Now, along with this growth in the size of a cluster, an obvious question is whether the M3 Pro comes with two AMX units (like the M2 Pro) or a single such unit. A single unit would represent the same design as M1 and M2, but you could also imagine something like the M3 Pro's cluster split into two "mini-clusters" each sharing an AMX unit, giving two AMX units per cluster.

The data on this was, for a while, very messy and unclear, but recently has become more clear.
Peak DGEMM (FP64) large matrix multiple performance on a single M1 AMX unit is around 300GFLOPS, so twice that on an M1 Pro or Max.
(One place to see this is https://github.com/philipturner/amx-benchmarks which refers to an M1 Max with two AMX units. Another place is https://www.cs.utexas.edu/users/flame/BLISRetreat2023/slides/GemmFIP-ExoLang.pdf which refers to an M2, and has that large matrix multiple performance on a single AMX unit as more like 350GFLOPS.)
The thread https://twitter.com/GabrielBaraldi3/status/1741519692458578341 is very confused, and ignore most of it as harmful to your sanity, the one number that matters is that it seems that code that achieved about 600GLOPS on an M1 Max achieves 800GFLOPS on an M3 Max, which is more or less what we would expect from frequency scaling. This means, in turn, that an M3 Max presumably has two AMX units (one per P-cluster) not four ("one per mini-cluster").

This is in contrast to Howard Oakley, who saw massive improvements in the M3 Pro vs the M1 Pro for a few Accelerate calls, eg
https://eclecticlight.co/2023/12/13/finding-and-evaluating-amx-co-processors-in-apple-silicon-chips/

https://eclecticlight.co/2023/12/07/evaluating-m3-pro-cpu-cores-5-quest-for-the-amx/
https://eclecticlight.co/2023/12/21/comparing-accelerate-performance-on-apple-silicon-and-intel-cores/
What he found was that the M3 Pro (which we have concluded has just one AMX unit) was consistently faster than the M1 Pro (two AMX units)! How can this be, with only a 4:3 GHz ratio?
The difference is that Howard was testing very small arrays (for FFT) and matrices, whereas the peak AMX numbers result from large matrices (128×128 or larger). So what we see is that (honestly, as expected) when the AMX unit is used as an outer product engine, calculating a full 8×8→64 outer product every cycle, it can only scale faster by the GHz ratio. But when it is used in other contexts (either as a vector unit, so for FFTs) or in the setup overhead for small matrices, that Apple's improvements since the M1 are visible, with a single M3 AMX unit consistently matching to beating a pair of M1 AMX units.

Small in-order cores

We mainly think of Apple's P- and E-cores, but Apple also designs very small in-order cores (which I've occasionally called Chinook cores, after the code name for the first version we know of). These are used as companion cores for the GPU and NPU (and media engines? and in AirPods?). From what little we know, they appear to be 64b ARM, to (at least sometimes) have SIMD/FP, and (at least sometimes) not to have an L2 cache.

No L2 cache 2019

There are very few patents that seem relevant to them; for a long time the only one I knew of was (2019) https://patents.google.com/patent/US10909035B2 *Processing memory accesses while supporting a zero size cache in a cache hierarchy.*This discusses providing a tiny core that omits an L2 cache (but has an L2 cache controller that mediates between the L1 and the rest of the system, to handle things like coherency tracking). Presumably this design allows for more reuse of at least some IP between the larger cores and the Chinook cores.

Now suppose you're designing a small in-order core. Just because it's small and in-order doesn't mean you can't use smarts to make it faster; it just means you care about a very different set of tradeoffs. The first possibility for improving performance is to make the core two-wide (think something like Pentium UV pipeline). You can also (if it makes sense) without much difficulty add an I- or a D-prefetcher. Adding even basic speculation (ie branch prediction) is trickier because you need some way to rewind in the event of misspeculation, which means you need to start adding additional physical registers to hold temporary state, a store buffer to hold speculative stores, and so on; and if you go too far down this path pretty soon you might as well just go full OoO. The above patent suggests that there is no branch prediction for this small core.

Very mild OoO (for loads that miss in L1) 2022

So if we just accept the cost of branches (which is not *that* large if we're only one or two wide and have a short in-order, low frequency pipeline) the other main cause for slowdown is load misses. What can we do about those? Some version of prefetch is one option, as mentioned; but apart from that?
In some situations successive loads all miss on the same cache line, so if we block on the first load, then once that line is loaded the subsequent loads all hit in cache. But, in other situations, successive loads

may hit different cache lines and in that case in-order really pays a price because we wait for the first load, then we wait again for the second load. It would be much nicer if we could, somehow, fire off multiple loads that we expect to all miss in cache, and have them all traverse the memory hierarchy in parallel, so that one delay results in the arrival of two or more lines or useful data. Strict in-order execution does not allow that.
Enter (2022) https://patents.google.com/patent/US12001847B1 *Processor implementing parallel in-order execution during load misses* which tries to get around this. The rough idea is a form of runahead – when we encounter a load that misses in cache, we keep executing, hoping that we'll encounter more loads that also miss in cache and which also fire off to memory. At some point we'll no longer be able to runahead, at which point we wait for our data to return, then we restart execution at the first load miss.

Now obviously there are constraints on this, depending on how much hardware you wish to throw at the problem. (For example are you going to provide a store buffer to hold stores that are now out of order relative to the loads? Or will you just pause at a store?)

The patent suggests that Apple is doing this in an extremely minimal way, much less aggressive than what's normally called runahead. So the conditions that will pause execution appear to include
- any use of the loaded value (so no speculation, including branch speculation)
- I think the ARM memory model would allows for stores to a different cache line from any of the cache lines being loaded, but you can't store to one of the in-transit cache lines without at least a store buffer and I'm not sure these tiny cores have a store buffer?
- running out of instruction storage space.
This third condition results because the scheme (to save energy) exploits the fact that the in-order design already has a slight decoupling between the Fetch front-end and the Execution back-end, with an Instruction Queue sitting between the two. So we reuse that Instruction Queue storage! Once we start this runahead mode, the first load that misses, and all subsequent instructions, are held in the Instruction Queue sitting before the Decoder. So we execute until a problematic condition (including that we fill up this Instruction Queue storage) then we wait for the data to return. At which point we restart execution with the instructions that ave been queued up in this Instruction Queue storage.

Now, if you think about it, as long as the instructions after the first load do not depend on the load, there's no harm in their executing, even conditional statements executing, even writes (to other cache lines, and as long as you are using memory barriers when required) executing. So we can optimize this further; we can simply drop all non-problematic instructions that flow through the Instruction Queue storage after the load (and subsequent loads that miss). We allow them to execute and write back their results to their registers, then they're done; they don't need to wait for the load, *and* they don't need to execute again after the load. This substantially magnifies the power of our scheme. With compiler help, we can potentially do things like execute a load that we expect will miss, then (using different registers) execute a fairly long compute loop while we wait for the data. We're not *really* OoO – we have no branch speculation, no speculating past stores without an address, no rearrangement of dependent instructions, and no register renaming. But we're also not exactly in-order either! A nice compromise to

Faster ISBs

(2022) https://patents.google.com/patent/US12045615B1 *Processing of synchronization barrier*

instructions
This is one of these horribly technical patents that wrings the absolute maximum out of a specification. ISB (instruction synchronization barrier) is used when, approximately, some code is changed (maybe the MMU details of a page are changed, maybe the actual instructions at some address are changed by a JIT). After such a change, along with various cache control instructions (to ensure that the changes of interest propagate either to TLBs or to the relevant I-caches) the last thing we need to do is ensure that we don't possibly have the (now incorrect) instructions hidden somewhere deep in the core, perhaps in the Instruction Fetch Queue, or perhaps in a loop buffer or whatever. An ISB, nominally, acts by flushing all these possible locations of instructions, so that we execute past the ISB by refetching instructions from the I-cache looking through the I-TLB (both of which, by definition, were forced to the correct state by our earlier cache/TLB manipulations).

But of course flushing all these places that might hold instructions is expensive. Can we get away with something cheaper?
The (very simplified) idea of the patent, as I understand it, is that we don't have to wait until the ISB is the first instruction in the ROB before we execute the flush and reload of all subsequent instructions. Instead all we need is to ensure that the various "relevant" instructions have completed, although they may not be at the head of the ROB because the ROB is being blocked by some irrelevant slow instruction like a cache miss. As long as all the relevant work has been done, we can flush the Instruction Queue and suchlike, and reload, so that all that work happens while we're still waiting for the head of ROB blocker to finally complete.

As you can imagine, the details of getting this correct under all circumstances are insanely complicated, and you can look at the patent itself if you want to explore some of them.

Load value prediction (sort of...)?

Value prediction is one of those ideas that's been considered for years but never really gained any traction.
Branch prediction is either predicting a single bit or a target address, and that works well, so why not generalizations that predict the probable results of other instructions? Both the academic and industrial worlds have converged on specifically predicting loads, but the idea still seems bluesky. Why?
Value prediction has a problematic ratio of payoff to cost. A branch prediction saves maybe 8 or so cycles, and saves those cycles across the full width of the machine. A load value prediction saves maybe three or four cycles, and while those cycles probably correspond to the critical path, the machine can usefully execute non-critical-path instructions while it waits. To be worth the risk of misspeculating a value and paying the cost, we need to be really confident!

With that in mind, there are two possible ways we can speculate a load:
- we might be able to guess the value loaded. For example on entry to a function, the first thing it might do is load various (constant or rarely changing) values from some globals.
- we might be able to guess the pattern of loads (the obvious case being walking along an array loading

each successive value). In this case we can “pre-execute” the load.
In both cases the idea would be that at Rename time, after allocating a result register to the load instruction we
- set the value of that register to either the guessed load value (the first case) or the pre-loaded value (the second case)
- indicate to subsequent instructions that the register is valid (so they don’t have to wait for the value)
- we ALSO send out the load instruction, presumably with an additional different destination register allocated, to be executed
- at completion of the load, we compare its result to the value we speculated (or pre-loaded) and if necessary recover from misspeculation. (Ideally this could be handled as a Replay, but that’s probably too ambitious and a Flush may be required.)

So to get this all to work and be worth doing
- we need a scheme to track load values and look for high confidence patterns
- we need co-ordination between Rename, the Load Unit, and the ROB (or whatever handles dealing with mispredictions)
- the cases where this is useful need to be common enough, and the cycles saved from the critical path need to be enough
- when addresses are speculated, we also execute each load twice, once for the “pre-load” and again for the “check load”. This is a problem if we are already executing three loads every cycle or so – we don’t have extra load units free to handle this doubled work! So we also need to somehow detect this situation and, in that case, temporarily switch off load address prediction. Unfortunately this may be a common case when, for example, we are running over an array doing only a limited amount of work...

Is it all worth it in the end? That remains unclear, and is why this idea keeps hovering, always almost in reach, always apparently not quite worth the cost and the hassle.
We have already discussed the idea of load address prediction as a ZCL, and concluded (though possibly my tests were not ideal) that it was not present in the M1. There have been scattered claims (but never great data or careful benchmarks) that the A15 and M2 sometimes seem to execute loads “faster than possible” which suggests that maybe it is active in the M1/A14 successors.
Into this space we get a second patent, (2022) https://patents.google.com/patent/US12067398B1 *Shared learning table for load value prediction and load address prediction*. The patent discusses load speculation generally, but its focus is the use of a common table for learning load speculation of either the value or address form. I won’t discuss the details because, honestly, it’s still not clear to me if any of this is real. It’s possible that Apple looked at the idea (yet again, as so many people have before), came up with this idea as one more way to reduce one element of the costs (a single learning table, rather than two different tables, one for values, one for address patterns) and patented it, but after running the simulations, concluded that it’s still just not worth it – the number of cycles saved when you take into account the accuracy you need, and the inability to use this when load bandwidth is maxed out – means it’s still just not worth the area.

Two recent papers have confirmed the above.
(2025) https://yuval.yarom.org/pdfs/KimGY25.pdf *SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon* tells us that Strided *Address* Prediction has been available since the M2/A15, while
(2025) https://yuval.yarom.org/pdfs/KimCGY25.pdf *FLOP: Breaking the Apple M3 CPU via False Load Output Predictions* tells us that Load *Value* Prediction has been available since the M3/A17 Pro.
As is usual for these sorts of papers, the tone is somewhat hysterical, and almost all the energy is spent on the uninteresting issue of how to craft a (realistic or not) exploit, rather than investigating the details of the microarchitecture. Oh well.

In both cases, unsurprisingly, this fanciness is reserved for the the P core, and is not present on the E core.
What does appear to be the case is that up to about 72 different loads (ie distinct PC's pointing to a load instruction) can be tracked, in what looks like a 4-way associative table indexed by a hash of the PC. It looks like the system required about 250 successful loads before it latches a particular load value as trustworthy (ie worth value speculation) and so moves the load from training to a different "execution" table.
(The paper sees training sometimes succeeding after what looks like only 60, 120, or 180 successive loads. I think this is a byproduct of the precise way they wrote their code, with training sometimes hashing an earlier training run into the same training slot as a later training run. This could probably be avoided if they had used a different value in each of the addresses they load from rather than a common value across all addresses. It's a shame they didn't read this work and then looked at the patents in detail, which could have informed some of their various tests.)

Something you may have wondered about in the paper is why they tested for strides in the load *Value*. There is a Load Value Prediction scheme called EVES (2018) https://inria.hal.science/hal-01888864/-file/CVP1-Final.pdf *Exploring value prediction with the EVES predictor* which builds on the E-Stride value predictor, which in turn assumes a stride in the load value (eg every time your perform the load, the value has been incremented by some earlier store). While the full EVES scheme would require some restructuring, the simpler E-Stride scheme might be able to fit into the existing shared Load Address and Load Value training table?

For Load Address Prediction, we seem to require about 120..500 successive loads before the predictor is trusted, and the stride can be (something like)between -256 and 256. There's some messiness in the plot they derive, which I suspect is due to the same issue as I describe above, that running many successive tests right after each other is occasionally hashing a test into the slot of a previous test and so there's a bunch of wasting time tracking the loads as though they were still matching a prior load pattern, before the system "resets" and tries for the new load pattern. The original 2019 patent suggests a 6bit counter, with one or two values having special meaning, so that only 62 or 63 successive matches are required to latch a pattern. The one thing the paper seem fairly clear about is that the training time jumped from ~120 for the M2 to ~320 for the M3. Perhaps, even apart from the sorts of hash-reuse issues I've described, the training counts were changed as part of the use of a unified table on the M3 to train both Load Value and Load Address prediction?

One final interesting thing they noticed. Load Address Prediction is present in the iPhone 13 Mini but not in the iPhone 13 (announced and launched at the same time). This suggests that perhaps the functionality did not quite work in an early stepping of the A15, but the chip was fixed in a later step-

ping. The details of this are unclear! Was the difference the authors saw a deliberate policy? Or more something like the first few million of both iPhone 13 and 13 Mini used the bad stepping, then they both switched to the new stepping, and you got a (slightly) better iPhone by buying a later one?

Even crazier things are possible. Look at (2024) https://arxiv.org/pdf/2406.18786 *Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution*.
They define a *global-stable load* as one that, essentially, for every execution of that particular load instruction (ie that PC) uses the same address to load the same data. They find that, across a wide range of code, about 30% of loads fit this (!!!) which sounds absurd! And that if you could somehow eliminate these loads you could speed things by around 4 % to 8% depending on your assumptions.
So far this sounds like Load Value Prediction. But Load Value Prediction requires the execution of a load to validate the prediction. It does shorten the execution critical path by a few cycles, but it does still require the energy cost of the validating load, and the fact that occasionally that validating load will use load execution resources that the critical path should instead be using.
Their idea is that if we can tell that
- the registers that go into forming the address of this global-stable load have not changed, AND
- no store or snoop to the target address have occurred
then the value must be the same as the last time it was accessed. You can imagine the rest.

Now this should sound familiar! It's very much like https://patents.google.com/patent/US11900118B1 *Stack pointer instruction buffer for zero-cycle loads* and some of the earlier zero cycle loads – tag a physical register that was stored or loaded with the conditions that led to its store/load, track that nothing has changed, and if we're OK, then execute the "load" via a Rename. So it's a different implementation of the same sort of idea.
Given that the existing rename-based scheme covers many of the cases of interest, and the existing Load Value Prediction covers most of what's left, it seems unlikely that we'll see this implemented on a mac; but we may see it implemented on x86 or ARM as a way to get to compete with Apple.

You may wonder why these sorts of apparently redundant loads even exist, surely the compiler should be removing them? There are various good(ish) reasons, for example the limited ability of the compiler to analyze the entire program; but one underappreciated reason is the prevalence of this sort of code in debug and unoptimized code. It's clearly valuable to developers (who are, after all, an important market for the most powerful machines) if their debug, not just their optimized code, runs as fast as possible!

Exploiting branch reconvergence

It's not uncommon to have code of the form

```
if(){
   short code;}
following code;
```

where the point is that the flow of control converges after a few instructions.
We've already seen one way to exploit this situation, making fetch faster when we predict jumping over

the `short code`, by executing a straightline fetch and deleting the `short code` instructions from the Fetch Queue.
Consider an alternative situation where we predict the `if` is not taken, we execute `short code`, then we return to the main flow of control in `following code`. Unfortunately we later learn that the `if` was incorrectly predicted, so we flush everything and restart at `following code`.
Frequently at least some of the work done in `following code` is useful, and we'd like to take advantage of that. The easy way this happens is when loads in `following code` land up prefetching some data from at least L2. That's nice, but the in-place prefetchers already catch many such cases, so can we do better?

Consider for example loads executed by `following code`. These will have placed a value in a physical register. Suppose also the chain of instructions that led to the load does not depend on registers altered in `short code`. Then the already loaded value in the register should be legitimate! This is kinda a cross between value speculation and a zero-cycle load.
(2023) https://patents.google.com/patent/US20240354109A1 *Re-use of Speculative Load Instruction Results from Wrong Path* discusses the details of how this can actually be implemented.
Deriving value from instructions executed after reconvergence is one of those ideas (like cache compression, or value speculation) that's continually being revisited in academia, but with a general feeling that one day it will probably happen, but not this year. Has it happened this year (ie, is it present in M4?) Who knows? As I've said elsewhere, as these techniques become ever more leading edge, I suspect we may see initial implementations on one or two or three chips, each implementation a test case that's stressed aggressively in silico, but one in a billion edge case bugs are found in the first implementation or two, before it's activated for consumer use. In other words, maybe we as outsiders actually see this in the M5 or M6?

Can we get any further value out of what we executed after the convergence point? This is a balancing act because you don't want to waste resources storing anything related to "easy" instructions that can be executed in the background on your wide CPU. ideally (yet again, I know I'm a broken record on this) we'd track instruction criticality and use that. Without criticality, loads are frequently critical and long latency, so they're a good choice.
The other obvious choice, and also implemented by the patent, is branches. Specifically, we track the outcomes of whatever branch comparisons were performed in the converged region, and pass those back to the I-fetch unit as part of the flushing/resteering when we realized that we had executed down the wrong path. Most of the time this will simply confirm what the fetch or branch predictors were already going to say, but now with full confidence.
SInce this mostly confirms what was already predicted, it's a nice to have, and presumably "easy" given the load infrastructure you already have in place, but it's not *especially* valuable. Much more valuable would be recording the target of *indirect branches* in this circumstance. The patent does not mention this (presumably you add one complicated new feature at a time, not everything you can think of) but it's the obvious next step. And then, once you have criticality in your CPU, a whole new set of options opens up for what you should be recording.

GPU memory queues

This one is simple but cute: (2022) https://patents.google.com/patent/US12079144B1 *Arbitration*

sub-queues for a memory circuit.
Consider the GPU, aggressively generating transactions to be sent over the NoC to the memory system. (This patent talks in general terms, but we all know this is about the GPU, and maybe one day the NPU!) These transactions will be placed in a temporary queue until an arbiter grants a slot on the NoC to accept a transaction.

Recall that the current NoC is split between an address-carrying network and a data-carrying network. The data-carrying network is probably 64B wide, so that a 128B line transaction takes two cycles (the line width of the NoC, as opposed to the 64B line width of the CPU L1 caches). This means that a data transaction (eg a write from GPU to memory) will take two cycles. It also means that for one of those cycles the address network is free for address-only transactions (usually reads, though occasionally fancier things like SLC control transactions). So this is good, means we can get extra value out of the network by sometimes sending down an address-only transaction while the second cycle of the data bus is being used.

Now consider what happens if the GPU generates a sequence of writes. Now we can't make full use of the address bus because the queue feeding transactions into the NoC is a single in-order queue. Obviously this is not great!
The patent suggests a simple fix. The queue is split into three queues. Transactions are placed in one of the three based on a hash of the address of the transaction. The arbiter can now look at all three subqueues and (hopefully) with this distribution of the requests, there will usually be at least one address only transaction available at the head of one of the queues, when such a request is desired to make optimal use of the split NoC. Hashing by address means that requests stay in-order "enough", specifically a read will not be moved relative to a write to the same address!
(Presumably this makes use of the hash-to-3 circuit that we described earlier in conjunction with the M Pro's three memory channels.)
Obviously this scheme could be grown, to eg 4 or 5 queues, if later, faster GPUs make that appropriate.

New in the neural engine

Since the A17 there has been some controversy about whether the A17 was in fact twice as fast as earlier ANE's, and whether that was due to supporting INT8×INT8 at double speed (like the first step nVidia took in speeding up their support for ML code). The situation was confused for a long time because almost no AI benchmarking code seemed to show any performance improvement by using INT8 rather than FP16. What was missing?

When the iOS18 betas came out, the situation became more clear. The relevant app is coremltools which was upped to version 8. This app translates an neural model from some standard format (eg pytorch) into something Apple can compile to Apple Silicon. If you look at the release notes, https://github.com/apple/coremltools/releases/ you will see that they have added support for model calibration which, as far as we are concerned, means that you run the model on some representative data and collect a profiler which is used to estimate the mean and range of the activations of each

layer. If you have this data, you can then covert your model from W16A16 (ie weights are in FP16, activations are in FP16) to W8A8 (weights and activations are both INT8). You need the calibration data because that is used, after each layer executes, to bias and rescale the activations as appropriate based on the calibration of that layer.

Point of all this is that W8A8 was not visible in any benchmarks until this newest coremltools. And you need a W8A8 model to make use of any sort of accelerated INT8×INT8 multiply! With these new models, we did indeed see substantial improvements in some of the INT8 benchmarks (when compiled correctly, and when run on a new enough version of macOS or iOS). So we can conclude that the newest Apple SoCs do in fact have accelerated INT8×INT8 multiplication, presumably by adding an additional INT8×INT8 multiplier to datapath in each Neural Engine. This is not as expensive as it might seem because most of the rest of the existing datapath (the machinery to handle FP16×FP16 and FP16×INT8) does not need to be replicated – none of the machinery to handle floating point and especially not the shifter required for FP add.

The next controversial issue is whether or not Apple has (as of presumably M4) added any sort of support for faster 4b quantization (remember the patent showing a neural engine that splits an 8×8 multiply into two 8×4 multiplies or even four 4×4 multiplies?), or even hardware to handle bitnets (neural networks where weights are either +1, 0, or -1). In both these cases, you'd probably want to use INT8 activations, along with calibration. Bitnets are especially interesting because
- MS has achieved very impressive results using them
- Apple (as mentioned) acquired xnor.ai which did a lot of work on bitnets
- computation of a bitnet replaces multiply-adds (activation×weight) by sums and differences (each activation is multiplied by ±1 or 0, and then added) so a rather simpler datapath is possible)
- a bitnet actually fits well into Apple's existing HW (obviously you have three weight values, +1, 0, -1; but you can encode this as a one-bit plane that specifies zero vs non-zero [which, remember the Apple kernel decompressor supports] and a second one-bit plane that specifices +1 vs -1 as 1 vs 0).
Both these issues (special support for 4b weights and special support for bitnets) remain topics of controversy and it is possible that, like W8A8, the hardware has some support but we won't see any evidence for it until a new OS release with all the required support added to the ANE driver, compiler, runtime, tools, etc.

OK, we now have definitive proof! (2023) https://patents.google.com/patent/US20240329933A1 *Neural engine with accelerated multipiler-accumulator for convolution of intergers* [sic, two misspellings! they put this one out in a rush!] describes exactly what we suggested above – adding to the existing FP16×FP16 multiplier (which can also handle INT8×INT8) an additional INT8×INT8 only multiplier. Along with other elements we suggested, like how the existing 16b-wide path from the activations storage into each FMAC can carry two INT8's, so that part of the design does not have to be revised. One minor tweak to the design that they describe is splitting each multiplier into more than one sub-multiplier. I don't think this is a *performance* element, not yet; rather it's a power element. The 11-wide FP16 multiplier can be split into units that handle the upper 3 bits and the lower 8 bits, which can save power if we're multiply INT8×INT8. Going further, the 8-wide multiplier could be be split into two parts that handle say INT8×INT4 multiplication, which would give you a power (albeit not a performance)

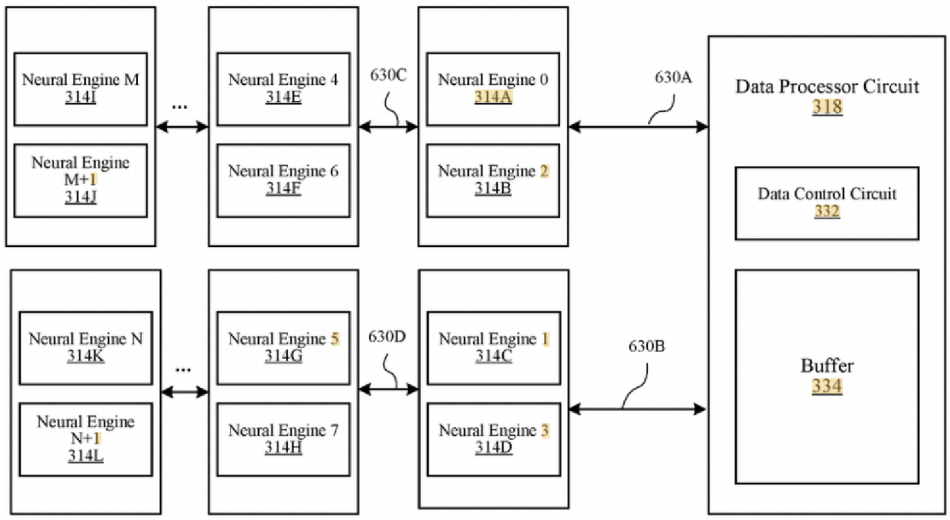
advantage when executing a W4A8 layer. To go further and double the performance of W4A8 will require augmenting the (now two) adders in the accumulator with an additional two adders, for which there's not yet any evidence.

As I described both W8A8 and a possible W4A8 mode rely on linear quantization based on biasing and scaling the INT8 or INT4 values. (2023) https://patents.google.com/patent/US20240329929A1 *Processing of asymmetrically quantized input and kernel coefficients in neural network processor* describes some low level details of how that is actually implemented in the Neural Engine to reduce the required number of computations. In particular some of the work is done in the Neural Engine itself rather than what I suspected, namely doing it all in the Planar Engine as a postprocessing stage.

On the more definitive front, we have:

> *Better neural engine co-operation*
> We have 16 neural engines. How should we distribute work across them?
> The easiest way to do this is to split the work by "channel". Think of the image recognition example we used when we discussed how neural networks work generally – first step is to take in an image (for simplicity assume it's a grey scale image) and apply a large number (say 128 or so) convolutions to it, so that each convolution detects lines at a different angle. Next we repeat this merging the "line" images in various ways to generate multiple channels each of hich correspond to the presence or non-presence of a texture in each part of the image. And so it goes.
> In this model, the easiest way to split work is to give each of the 16 neural engines a subset of the channels, and each works on a copy of the full image.
> This works fine if you have a large number of channels (ideally a multiple of 16!) but this may not always be the case, your work may (in a hand-waving description) look more like multiplying two large matrices together, rather than like multiplying 16 pairs of mid-sized matrices together. So we'd like the ability to split a single task (for example the handling of one channel) across multiple neural engines by splitting the input image (or an input matrix) across neural engines. (2023 https://patents.google.com/patent/US20240320470A1 *Mulicast and simulcast in neural engines* [sic, the patent spells it as Mulicast, maybe that will be fixed in a patent update?]

This patent is kinda vague and mostly talking to itself in language that means little to anyone outside Apple. But the important points that can be gathered from it begin with this diagram:



You may recall an earlier patent which discussed clusters of neural engines, suggesting that there were four clusters of four, but somewhat vague on details. This patent seems to reflect reality as seen in die shots, which show 8 repeated units in the ANE, not 16; the earlier patent may have been initial ideas before simulation suggested the optimal design?

Anyway point is that the ANE neural engines are built as two “clusters” consisting of four pairs of neural engines. Each cluster has a separate connection to the “Smart L2” in the center of the ANE. Data flows from the L2 to the first pair, then if necessary is routed to the second pair, then third, then forth, in a dasiy chain topology.
This has two consequences:
- If data can be reused by multiple neural engines, it can be multicast, that is sent once by the L2, and each target neural engine can grab what it requires as the data flows past
- different data can be sent simultaneously to the first cluster and the second cluster because two separate communication channels are available.

This is the hardware side of things. Most of the patent is then about how the compiler optimizes things to take advantage of these hardware details. Obviously these mean that, if you are going to split the processing of an image (or a matrix) you first do so within one cluster (and perhaps within a neural

engine pair? the diagram and patent suggests these pairs share something, perhaps a common dual-ported tap into the daisy-chain communication channel?); and that it's preferable, if possible, to run two channels, each split across one cluster, than to split a single channel over both clusters.

VQ-based weight quantization

Obviously shrinking the size of weights is an on-going concern.
So far we have seen (supported in the HW and then, perhaps a year later, in the runtime and tools)
- a bitmask that indicates zero vs non-zero weights
- the above could be augmented by pruning (ie looking for values close to zero and forcing them to zero)
- providing weights as "linear" INT8. Linear means that we take the range of the (FP16) weights of interest, subtract a bias to zero the range, then use a multiplier to spread the range from -128 to 127. We then undo this after a layer. You can also do a 4bit version of this. CoreML seems to refer to this as "quantization".
- palettizing weights (ie lookup table), of 8, 6 or 4 bits lookup. Now we use some sort of clustering to figure out the optimal 2^8, 2^6, or 2^4 target values. (Intuition suggests that you want to build a histogram of values, then cluster bins together to make the target number of *equal probability* bins, then choose something approximating a median in each bin.) The looked up value could be an INT8 or an FP16. There are also 5 and 3 bit versions of this.
- the ability to combine the second two of these with a zero bitmask
- Given that newest hardware supports W8A8 multiplication, you might want a mode that does something like a 5 bit lookup that returns a linearized INT8 value. The CoreMLTools documentation suggests this is possible, but the language is somewhat unclear.

This all sounds pretty good. What can we do to make our weights even smaller? If you've ever worked in lossy compression, you'll know the next step after palettization is VQ, vector quantization, which Apple/ML, having redefined quantization, call Vector Palettization.
Recall that basic palettization is we try to cluster some large number of possible values into a small number of clusters, so that we can replace each weigh with an index to the cluster that best matches that weight. VQ extends this by taking, say 4 successive values and seeing if they (that array of 4 values) tend to cluster in space. Suppose we want to use 4bits per weight. That means we can provide 16bits for a 4-weight lookup. This means we need to find 2^{16} clusters of our arrays of 4 successive values. The extent to which this works well depends on whether there is some sort of innate structure in nearby weights – maybe they tend to be close in value, or to have the same sign?
Regardless (2023) https://patents.google.com/patent/US20240232571A1 *Palettization of Kernel Vector in Neural Network Processor* describes an extended lookup table in the ANE kernel decompression hardware which handles VQ lookups. The natural assumption is that this is part of the M4 generation given that tool support for it shipped around August 2024.

This is not the end of the line. Something that Apple should probably work on (though I have not seen any patents) is compressing the zero-mask bitmap.
This bitmap is an obvious, easy idea, and is fine when your weights at 16 or 8bit. But a 1bit mask starts

to cost a lot if your weights are say 4 or 3bits. Is it worth it?
Well, what fraction of your weights are zero? If ~50% are zero, then a bitmap is the right choice. But suppose say 90% of your weights are non-zero.

```
Lb[x_] := Log[2, x]; Ent[x_] := -Total@(# Lb[#] & /@ x);
p = .1; q = 1 - .1;
Ent[{p, q}]
```

Out[ ]=

```
0.468996
```

So Shannon says we only need less than half a bit per bitmap element to represent this data (90% of weights non-zero); but we are expending a full bit. This seems sub-optimal!
The obvious easy fix is to compress the zero bitmask with runlength encoding. The details of how one might do this can vary tremendously, and would depend a lot on the real-world statistics we see of how zeroes are distributed. One could look at, for example the G4 fax standard for a wide set of ideas for how to compress bitmaps. But this seems like an obvious low-lying fruit that Apple has not yet plucked!

Image cropping by DMA

We have already seen the existence of a texture unit in the ANE, so it's not clear quite what the added value is of (2024) https://patents.google.com/patent/US20240330217A1 *Input and output spatial cropping operations in neural processor circuits*.
Maybe the point is that this cropping is now being done by DMA, so it saves memory bandwidth and power, and we only have to invoke the higher power of the texture unit if we subsequently want to rescale or otherwise manipulate the cropped image?

Memory Thermal Protection

This is interesting as one more example of neat things Apple can do by controlling the RAM.
DRAM has a mechanism to report its temperature to the host. However the reported temperature is very coarse (accurate to 5K or so) which has unfortunate side effects. For example if you are using this temperature to control a fan, it means that your fan speed continually oscillates, in an audible fashion between low and high as the temperature you think you are trying to reduce jumps by 5K every few seconds. To fix this Apple places addition (much more sensitive, to .1 or .01K) thermocouples on the DRAM. Accurate temperature sensing not only allows for stable fan control, but also allows the system, especially for fan-free designs like phone or iPad, to run close to, but just under, the danger temperature, by throttling memory requests as required.
(2022) https://patents.google.com/patent/US12055988B1 *Memory thermal protection*.

Improved cache controls (latency vs bandwidth)

We've already seen that the CPU complex has a way to detect whether it's interaction with the SLC/-DRAM is bandwidth vs latency limited. This patent (2022) https://patents.google.com/patent/US12026108B1 *Latency-based performance state control* is the next step along that axis. Much as we saw power control evolve from independent IP block to a central controller, we are probably headed that way in terms of SoC-wide latency/bandwidth tradeoff. But we're not there yet! Right now we're still at the stage of each IP block kinda reinventing its own thing :-(

So this is the GPU version - but unlike the CPU version, it's written generically, in a way that could be applied to other IP blocks. So...
The specific issue is the link from the GPU L2 to the SLC – we'd like to know if the GPU is currently bandwidth bound or latency bound, and adapt as appropriate. The specific adaptations will depend on NoC and SLC details, but you could imagine, for example, flipping the NoC between a low vs a high frequency, or an SLC that can run in a low power state (which takes 8 cycles of latency from a request till when the request has been looked up and moved to the NoC, vs a high power state where this is 4 cycles). Or the memory controller, in high bandwidth mode, might delay requests more, in order to try to launch more requests to a page once that page is open.

So now that we know the goal, how can we detect bandwidth or latency overload?
There are basically two strategies available.
One is to look at backwards, at the history of requests. By counting the bytes in each request as it goes out over epochs, we can build a histogram of both read and write bandwidth (and even bandwidth on the address bus). If we also attach a timestamp to each request, we can then also build up a histogram of latencies.
Alternatively we can be forward looking. In this case we track bandwidth by the requests that are enqueued in the L2-to-NoC interface, and latency by the number of such requests queued up.
The CPU mostly uses the second strategy, but this patent suggests using the first. This might be because if you're doing this by yourself without a central co-ordinator, then NoC latency and bandwidth don't just depend on your requests, they also depend on what the ANE, media, display etc are all doing. A centralized controller would no longer have this issue, and might revert to the CPU strategy.

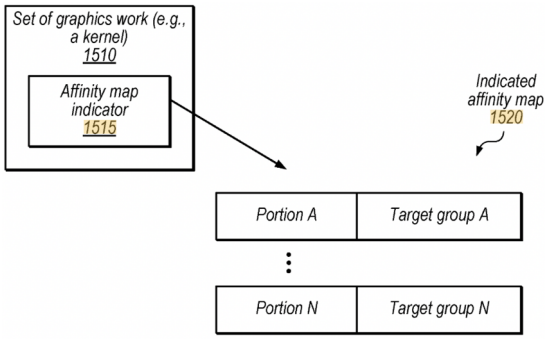
The rest is obvious. Based on the histograms, you classify the current GPU to memory demand as high or low bandwidth, and high or low latency, and send that classification to the Performance State Controller, which will react appropriately.

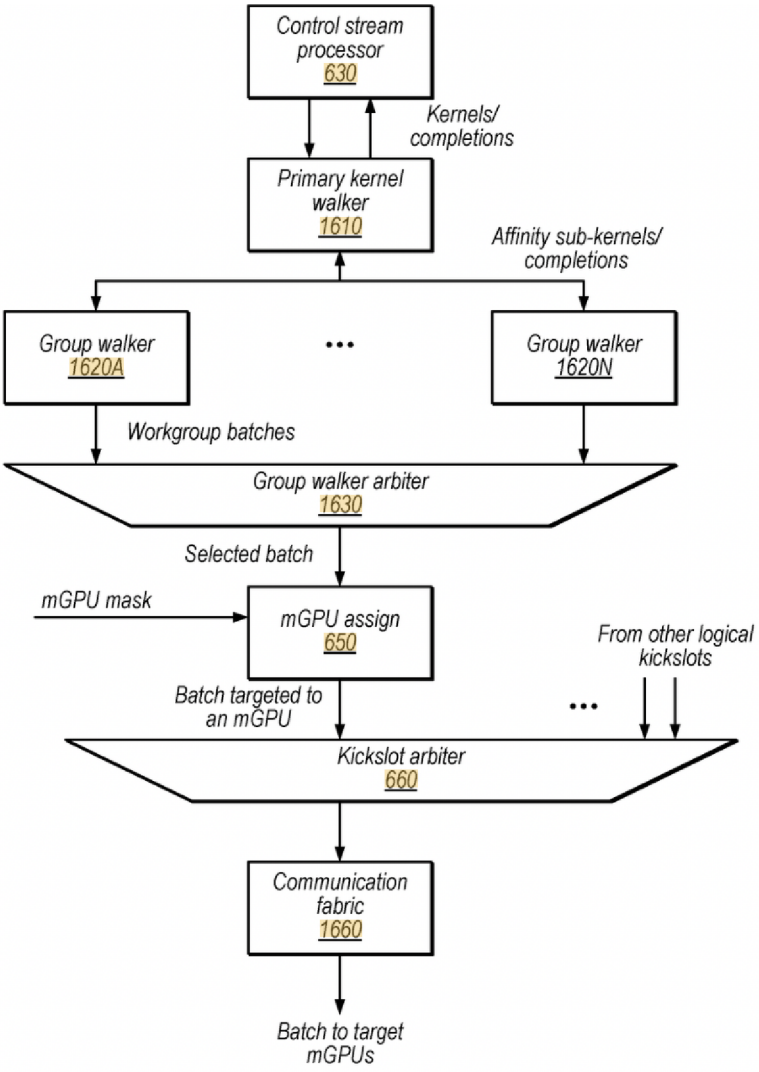
GPU changes (likely as of M4)

Recall that when we left GPUs the state of the art in scheduling looked something like this:
We begin with some collection of tasks that have been indicated (one way or another, at the Metal level, by how the tasks are packed into queues) that these tasks can operate as a whole without having to synchronize with the SLC and CPU.
Those tasks are split into smaller units, I assume through some combination of implicit and explicit features of the Metal language and decisions by the compiler, units called kicks, which now can operate without requiring synchronization or L2 coherence. In concrete terms, before and after a kick, the L1 of a GPU core is synchronized with the GPU L2, but no such synchronization is required during the execution of the kick – if two kicks depend on each other they need to indicate that one must execute before the other. A kick may be small, only needing to execute on a single core (or even just a single quadrant) or large, so that it can execute on many cores (but whatever it does, the work has to be independent enough that what's done by one core doesn't require automatic hardware visibility by any other core. Kicks are more of a hardware concept, generally consisting of multiple threadgroups, where

threadgroup is the SW concept seen by the developer.

So, with this in mind, consider the problem of a scheduler that's given a list of kicks to execute. There are obvious constraints (some kicks depend on others) and obvious optimizations (if a few cores are free, and small kick is available, schedule it on those cores). Where things get more complicated is once we have an Ultra design, where, in some circumstances we might want to restrict a kick to one SoC, while in other circumstances we might wish to spread a kick across both SoCs. When we left things, this was handled by having a single queue master queue up kicks in a single virtual queue, and then (based on various flags associated with each kick) decide to which of multiple (in the case of an Ultra, two) logical queues to send the kick.
At a more abstract level, we could imagine the GPU as hierarchy of compute entities: a core consists of four quadrants, maybe say five cores (size of an A-series GPU) are linked as a "supercore" sharing an L2, an M consists of two supercores with two (mutually coherent) L2s, a Pro consists of four, and so on. You could imagine a design like this has some advantages in terms of nearby cores having higher, lower latency mutual bandwidth, and being able to power down supercores and their associated L2s as necessary.
This scheme of a virtual queue that decides, based on flags, where to place kicks, obviously generalizes at a conceptual level to such a hierarchical design. The main thing is we want to maximize affinity as much as feasible, with the option to add in affinities later even if today we don't do so. It's more or less summarized in


Set of graphics work (e.g., a kernel)
1510
Affinity map indicator
1515
Indicated affinity map
1520
Portion A
Target group A
Portion N
Target group N


Control stream processor 630
Kernels/ completions
Primary kernel walker 1610
Affinity sub-kernels/ completions
Group walker 1620A
Group walker 1620N
Workgroup batches
Group walker arbiter 1630
Selected batch
mGPU mask
mGPU assign 650
From other logical kickslots
Batch targeted to an mGPU
Kickslot arbiter 660
Communication fabric 1660
Batch to target mGPUs

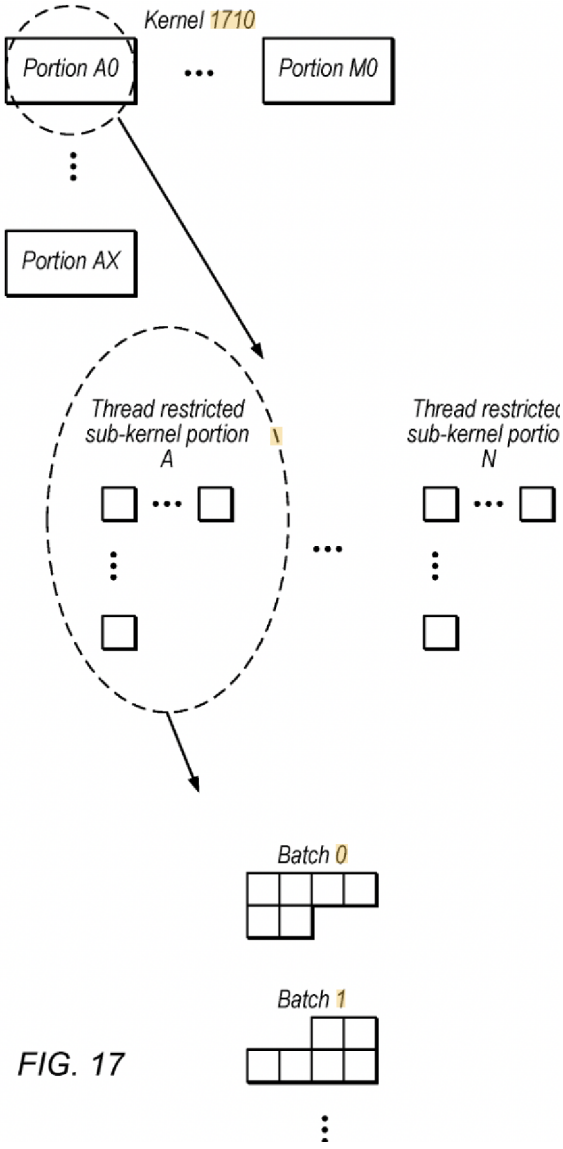

Kernel 1710
Portion A0
Portion M0
Portion AX
Thread restricted sub-kernel portion A
Thread restricted sub-kernel portio N
Batch 0
Batch 1

FIG. 17

We see that (presumably the compiler, maybe with an assist from the driver at run time) associates an affinity map with a kernel. The affinity map suggests how to split the kernel grid into “portions” that should each target a separate SoC. The three diagrams show the flexibility that’s available, along with the confusion in the patents! The middle diagram switches terminology from *kicks* to *batchs*, making it seem like these are basically the same thing, different terminology used by different subgroups within Apple. The third diagram (and accompanying text) suggests how this fits into earlier batching:
The portions, A0..M0..AX--MX we understand, they’re each portions of the grid that should go to different SoCs. then each portion is split by the previously described (per-SOC) kernel walker infrastructure to create batches (think threadgroups, or collections of threadgroups) that are sent to each GPU core.

This gets us to the M1 (and probably the M2?) How can we improve on that? That’s the goal of (2023) https://patents.google.com/patent/US20240272940A1 *Pipeline Techniques for Dependent Graphics Kicks*.

The first big issue is that the above scheme as implemented on M1 (and maybe M2) only allows for three types of ways to spread a kick – it can be allocated to a single core, to an entire SoC, or to an entire GPU. The new scheme allows allocating a kick to any number of appropriate cores, for example four cores of a ten core M-class device. The patent suggests this allows more small and medium cores to run faster. (Previously you either you spread a kick over ten cores, which wasted energy; or ran it on one core, which might be slow. Now you can run it on four cores while another kick runs on six cores, and both are “right-sized”.)
This seems like an obvious thing to do, but for it to be worthwhile, you need some intelligence, probably split between the compiler and the runtime, to know the optimal number of cores (which may change from generation to generation) over which to distribute a kick.

The second big change which kinda fits with the previous one) is that we no longer have to wait for a kick to receive its full complement of allocated cores, we can start a kick allocated to four cores on two cores as they become free, then schedule the other two cores later.

The third change is that the previous two changes encourage more, smaller kicks as able to fit into a variety of different-sized GPUs more flexibly. But what if we are limited by the size of the queues that hold the kicks? The third change allows these kicks to be queued in RAM (which I expect, in reality means in GPU L2 cache). This allows effective use of a different style of GPU coding (or GPU compilation) which creates many, possibly dependent, kicks placed in many queues (each queue ordered by dependency) so that arbitration has a wide range of choices to look at each time it tries to match a new kick to available resources.
One way to view this is that it continues the virtualization of the GPU. The first round of this was virtualizing registers and threadblock scratchpad, now we’re virtualizing scheduling resources. In both cases we don’t want performance to be limited by not having provided enough of a particular resource as dedicated hardware.

Another improvement in scheduling is that scheduling can now schedule based on kick *deadlines*, rather than just the previous scheduling based on priority (and within that some combination of round robin and first come first serve).

Kicks are fairly heavyweight objects, and not schedule cycle cycle, so it's possible all this scheduling is in fact done by the companion core, not dedicated HW? In which case what we are seeing here is actually more advanced firmware, which might (at least in principle) be back-ported to M2 and M1 as system updates.

The flip side of scheduling is handling completion at the end of a kick. This has also been improved, by providing virtualized completion queues. The big picture is that when a kick is "completed" it is moved to a completion queue (likewise in address space, so probably in L2, not in dedicated SRAM). Later, when the scheduling machinery (which, like I said is, I suspect, the companion core) has some free time it will walk those completion queues and deallocate resources. This allows us to defer completion to a more convenient time, rather than having to delay scheduling new kicks while we complete old kicks.

"Completion" may refer to a kick reaching its natural end, but it may also refer to context switching or to a kick being killed (eg an app was force-quit), and these additional "completion modes" have also been tweaked in various ways. The most complex "completion mode" is when a kick discovers that it requires more resources than are available. The patent assumes a lot of background knowledge, but it appears that this can happen with geometry kicks, and that handling this has been improved. The idea, as best I can tell, is that when this happened in the past, the kick was cancelled, we wait until the entire GPU is free, and we try again, and this has been improved in two ways. Firstly by allowing the kick to halt, then drain the machine of existing work, then continue (rather than restart). Secondly, by splitting geometry processing kicks into smaller elements whose results are "stitched together" in a second geometry phase.

All the details of this are bound up in Apple internals, but the big idea seems to be some combination of:

- we understand how tiling works, and how it distributes work across multiple GPU cores
- but Geometry, as an earlier stage that, in principle, could distribute triangles to any tile, is not as obviously split across multiple cores
- what's new *seems* to be

+ the frame is split into large units ("Macro-tiles"), and some initial (sequential?) code rapidly splits geometry across macro-tiles
+ then serious geometry processing (which these days is not just triangle co-ordinate transformations, but is also tesselation and meshlets) runs on separate cores for each macro-tile
+ generating bins of triangles, to be fragment processed in each tile

Additional elements of this seem to be splitting a single "Geometry handling" phase in multiple smaller phases (parsing, segment processing, stitching, ...) which again allows for more parallel execution.

The above is all for handling large complex geometry. The patent also refers to a "streaming mode" which may be appropriate for less complex geometry? The idea in this case seems to be that partway through geometry processing (based on what? presumably some sort of indication in the geometry stream, perhaps placed there as part of the geometry construction?) we know that a particular tile is "complete". We now pause the geometry processing and fragment shade the tile, then resume geometry. Presumably this saves energy and memory bandwidth because the shading can immediately read, then discard, the newly created triangles, without them ever having to be written back to SLC (or

if we're lucky, maybe not even L2). This might work well if the app queueing the geometry has a pretty good idea of how to localize it so that, after some point, eg all geometry relevant to a particular $\frac{1}{16}$th of the screen has been enqueued and a marker indicating that can be enqueued? Perhaps we'll see something about this in an update to the Metal API.

Another GPU pain point has been the cost of synchronizing work between the GPU and other IP blocks. To improve *some* aspects of this (unfortunately, so far, this seems to apply only to graphics, not to generic computation shared between GPU and CPU) interrupt-like flags have been defined, so that on the conclusion of a piece of work by the GPU, an interrupt is triggered to immediately shunt work to some other IP block (eg media, ISP, or ANE). The idea is that rather than moving data between IP blocks at frame granularity, the data is streamed at macro-tile granularity, again with latency and energy (more cache reuse) benefits.

The above are various ways of a general point (which we have also seen in the most recent changes to the ANE) of trying to "stream" work through an IP block. Obviously it's easy to enforce dependencies between kicks by requiring kick A to finish before kick B starts, but just as obviously this limits the degree of streaming possibly. The question then, at the HW and SW level, is how can you indicate an appropriate level of granularity to allow part (but not all) of kick A to complete and then part (but not all) of kick B to begin utilizing what kick A has already created? An additional scheme described in the patent is not fully general, but is certainly useful. It allows a "parent" kick to specify an "Early Dependency Release" point, and a "child" kick to specify a "Late Begin" point.
The idea is that many kicks have a startup phase and/or a cleanup phase during which the work they're doing no longer touches the work passed between a parent and child. So a parent can execute the Early Dependency Release point (indicating that all the *shared* state to be used by the child is now stable) and the child can begin using that state while the parent does whatever final work it wishes to perform. Likewise the child can begin setting up whatever it needs (calculating tables, loading constants from RAM, or whatever before the parent state is stable, up till the "Late Begin" point, after which it must not proceed until the parent has indicated a Dependency Release. Obviously this won't always be applicable, but when it is applicable it allows some degree of overlapping the startup and wind-down phases of successive kicks.
This mechanism, called *kick gates* is somewhat generalizable (you could imagine a parent kick executing in three phases, and after each phase allowing the child kick to proceed to consume the phase that was just generated) though it's unclear how aggressively we will see this over the next year.
Another way you can make use of these ideas, even if your kick is not annotated by the compiler to indicate a Late Begin point, is to launch the kick early and let it execute the first stages of loading code, and resource allocation (which now means the equivalent of some memory management to allocate and populate "TLB" entries for registers and Threadblock Scratchpad) while the parent kick is finishing up. Similarly we can defer the last (hidden) stages of the parent kick, namely releasing various resources like register and Threadblock Scratchpad mappings, to happen as convenient while the child begins "real" execution. Saving overhead 2% here, 5% there, 3% somewhere else – each piece may seem small, but soon enough it adds up to 15% faster, without a process bump or very much more PPA added :-)

If you want to be really ambitious and forward looking, think of all these ideas in the context of MLIR. If you have been following the literature, you will know that the hot thing in AI/compilers/LLVM right now is converting a description of your task (eg a neural net specified in PyTorch) into an abstract representation (an MLIR, ideally a good compromise between expressing comfortably the task primitives and the HW primitives). The "compiler" then optimizes the MLIR in various ways (for example fusing tensor operations, converting a complex tensor operation into sub-operations that match the HW, removing redundant operations, etc). Frequently there is a cascade of multiple levels of this, for example Swift, compiled to the CPU, goes through an SIL (Swift Intermediate Language) stage which performs Swift-specific high-level transformations (eg concurrency stuff, or tracking mutability and refcounts) then SIL is converted to LLIR which in turn is optimized for the target CPU (ie some flavor of ARMv8 code).

Imagine now specification of a task of interest in some high level language which, using one or more MLIRs, converts the task not just into a stream of GPU code or CPU code, but into something that can execute across all these IP blocks (eg Media, ISP, ...) in the streaming fashion described above.
Neural Network compilation is closest to this ideal for Apple, with a single net compiled to execute, as optimal, across all of CPU, GPU and ANE; and probably soon (maybe not quite yet) in a streaming, rather than layer-by-layer, fashion. Google also appear to be close to this ideal in their particular context (having the compiler split large tasks across not just a single TPU but also TPU boards and pods.
It will surely be some time before Apple developers get to see this sort of level of abstraction, but it's also, surely, where things are headed. Maybe in a decade?

GPU Memory Management

We've seen that from the M3 onwards, GPU tasks are given a (probably multiple) private address space(s) into which are mapped registers and threadblock scratchpad, and these private address spaces are mapped into virtual pages. There's a whole machinery in the GPU to do this that is mostly analogous to the equivalents mapping virtual to address space. A new set of patents (different names but saying the same thing), exemplified by (2024) https://patents.google.com/patent/US20240354249A1 *Page Pool Descriptor Cache* provide a few tweaks to improve this design.

I may be attributing to these new patents an item or two from the first design for this memory management system, but as far as I can tell the improvements include
- the previous memory management system was not scalable. The various task required (maintaining how private pages are mapped to virtual pages, allocating and de-allocating mappings, that sort of thing) are now split into a few centralized tasks, while delegating as much as possible to distributed (presumably per core, but possibly per SoC?) hardware.
- specific page mappings were already cached in the analog of a TLB. But now a higher level construct, describing *pools of pages* (I guess this is the equivalent of the page table mappings maintained by an OS, along with queues of free vs zeroed vs active pages) is also cached.
- pages are "prefetched" and "pre-allocated". Consider a program on a CPU. When the program asks for a memory allocation from `malloc`, `malloc` requires an allocation of virtual address space from which to hand out smaller allocations. If `malloc` does not have such virtual address space available, it will

call `sbrk` to allocate a new virtual address page to the process. The OS will do its thing, eventually `sbrk` will return, `malloc` will do its thing, and eventually `malloc` will return. The new GPU memory allocation hardware wants to avoid this delay in this process, so it tracks the number of available but not yet active pages in each page pool (essentially virtual address pages that "malloc" could make use of) and when this number goes too low, some new virtual address pages are acquired and mapped into private space, so that hopefully private allocation is rarely to never delayed. I think prefetched essentially means acquiring virtual pages from the OS, while preallocated essentially means mapping such virtual pages into private address space.
- this adds to what I discussed above with the improved kick scheduling, allowing kicks to have resources (including memory resources) allocated before the kick actually "launches" and then have the resources (including memory resources) de-allocated after the kick has finished executing, and in all cases off the critical path of the main kick execution.

SME and hardware matrix multiply loops

Of course the big news recently has been that the M4 supports SME. This doesn't change the hardware much, just the instruction decoder details. It will take us all some time to work through the SME2 instruction set to figure out what new functionality is present, and how it performs. The official ARM documents is here (2024) https://developer.arm.com/documentation/109246/0100/Introduction , along with a simple introduction (2024) https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-scalable-matrix-extension-introduction that rephrases AMX concepts in SME vocabulary.

Now I want to discuss this very newly published patent (2022) https://patents.google.com/patent/US20240103858A1 *Instruction Support for Matrix Multiplication* which both ties a few strands together and suggests options for the future.

Using the language of SME, we know that AMX (SME2/M4 version) gives us 32 Z-registers (vector registers) that are 512b long (ie can hold 8 FP64 values), along with 16 predicate registers, along with 64 ZA vector registers again 512b long, which can be interpreted as being the equivalent of 8 "tile" registers that cover an 8×8 matrix of FP64's. Details in the ARM intro article above, including things like how this changes to four tiles of 16×16 FP32 values, two of 32×32 FP16 values, and one of 64×64 U/INT8 values.

The fundamental matrix operation is an accumulating outer product as shown below, ZA0+=Z1⊗Z3. Possibly with some lanes predicated out (this is most useful for handling edges as in the example below, where we only want a 5⊗4 tensor product and can save power by not performing the edge multiply-accumulates.

OK, that's all familiar. And you should be well aware of how we can use this primitive to perform the multiplication of A ($8 \times K$ in size) by B ($K \times 8$ in size) to give us C (8×8 in size) by a sequence of K repeats of (load a column of A, load a row of B, tensor-product colA with rowB and accumulate the sum into ZA). If we can perform two loads and an outer-product-accumulate per cycle then we can achieve the matrix multiply in K cycles. Very nice.

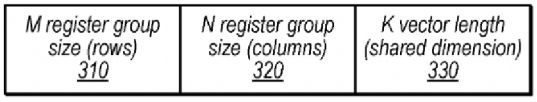
But of course, once we have very nice, can we make it even nicer? The patent suggests some ways of doing so though it's rather vague on the details (feels like a land grab patent to try to cover ARM and RISC-V potential implementations, rather than the tech details we expect from most Apple patents). Given Apple's implementation, two loads and one full tensor/accumulate per cycle is probably about as much as one can really expect in terms of execution. But what we can do is reduce the power and overhead cost of execution, by providing more powerful instructions that take up less space down the store pipeline and then when transferred from a CPU core to the AMX unit.
Two obvious ideas present themselves.
The first is, why only specify a single vector in an operation. Our inspiration here is the LOAD PAIR and STORE PAIR ARM instructions. So why not allow a similar load pair and store pair of Z vectors? Once you have that idea, you can generalize it to load/store of say four vectors, why not? And if fact we can go further. SME defines a concept of a Vector Group (VG) and in many instructions where you would indicate a single vector operand, you can indicate a VGx2 or a VGx4 operand. A VGx2 corresponds (in the simplest case) to Zn and Zn+1 and likewise for a VGx4. (There's a more complicated addressing version which we will ignore.) And not just for load store but for other vector-vector operations, we can indicate something like add this VGx4 to that VGx4 putting the result in some third VGx4. This is all very nice and obviously helps limit our instruction traffic.
This is in fact implemented in the M2 version of AMX (vector pairs) and the M3 version (also vector quads).

You could, in principle, be even more ambitious with this. Why not allow the tensor-product-accumulate of a VGx2 with a VGx2, so that our basic A×B matrix multiply described above now becomes a sequence of $\frac{K}{2}$ repeats of (load a pair of columns of A, load a pair of rows of B, tensor-product col1A with row1B, and accumulate the sum into ZA, then do the same with col2A tensored with col2B)? And likewise with a VGx4.
On admittedly very quick skimming of the SME I have not seen that this is possible with SME2 as currently defined, apparently vector groups can only be used for loads/stores and vector-vector functions. Maybe this will come later? Or maybe SME has hit a wall in terms of instruction encoding and just has no more bits left?

However the patent is more ambitious. The patent suggests concepts like the above (tensor-product-accumulate of a vector group). And you can be even more ambitious. Suppose that the FP64 matrices you wish to multiply are now $16 \times K$ and $K \times 16$. You could split them each into two matrices, 8 high and 8 wide, and proceed as before. But what if you instead load two successive Z (Z0, Z1) vectors as an entire 16-element column of the first matrix, and likewise for the second element (Z8, Z9). And what if you had a tensor-product-accumulate operation that performed four successive tensor-accumulates of Z0⊗Z8, Z0⊗Z9, Z1⊗Z8, Z1⊗Z9? Now you're doing even more work with one one instruction, and getting better reuse of your vectors. (If you didn't see quite how this works, look at the SME introduction above, page 67 et seq, where they describe the idea, though without describing the fancy instructions I am suggesting. The SME intro also describes some basic points like how you can most efficiently load columns of the A matrix before performing the tensor accumulate.

So the most generic version of the patent suggests that we provide hardware like AMX/SME, along with a matrix multiply instruction that implements both version of the ideas specified here – load wide vector groups (eg 16 or even 32--wide of FP64) in multiple vector registers, and execute a single tensor/accumulate instruction that performs the relevant vector-by-vector outer product and accumulate for all the different pairs of vectors in the two vector groups. And sure, why not load multiple such wide vector groups in one instruction, and then accumulate-loop over them in the later instruction!
The win in this is obviously less instruction traffic. The trickiness (at least one part of it) is knowing when the instruction has completed. Right now the AMX/SME instructions complete like other instructions, and since the ROB is so large, it's OK that they sit around in the ROB for a while. Other instructions will just flow past them, and they behave somewhat like a load that misses to L2 – takes a few cycles to complete, but nothing catastrophic. Extending this model to a tensor/accumulate instruction that accumulates a single wide VGx4 against a single wide VGx4 executing 16 successive outer-product accumulates is probably still feasible, but as you extend this to even deeper loops of say 64 or 256 successive outer-product accumulates you may hit problems! The patent talks in terms of a generic setup that could handle any sort of matrix, but I think the widest practical case is something like 16 (maybe, possibly 64) successive outer-product accumulates.
The patent also suggests that if you have a problem packing the details into your instruction address space, you define a separate register that holds the matrix details, so effectively you execute one

instruction that configures the “extended matrix multiply loop” instruction, something like



, and then a second instruction that actually begins the extended matrix multiply loop, .

In terms of real hardware, I think we can view this patent as kinda sorta relevant to the VGx2 and VGx4 capabilities in the M2 and M3.

In terms of *future* SME instructions and hardware, who knows?

(What future for AMX?)

At this point it’s interesting to consider the future for AMX – what are some (more or less reasonable?) options for future improvement?

- consider the changes made to the A17/M3 GPU, “virtualizing” registers. AMX has the unfortunate situation that it has to provide 4 (now 6 as clusters have grown larger) replicates of a large footprint of registers, almost none of which are ever used. Sure, as described above, one can make some attempt to extract value via early loads, but that’s limited in what it can do.

An obvious idea, at least in principle, is to copy what the GPU did! Provide an Operand Cache that’s a register file the size, or just slightly larger, of the state required by one core. Virtualize registers as living in an address space that’s mapped into physical address space. Provide a small SRAM cache (not visible to code) that can move data between AMX and the L2. With all these elements in place, the usual execution pattern should be that one core starts SME, no other core does so, and the local register file/operand cache is all that matters. If a second core does begin using SME, the virtualization mechanism can begin swapping registers between operand cache, local SRAM, and paging that local SRAM to L2 as required. There shouldn’t be much risk of serious thrashing because the OS should detect that two threads are trying to use AMX and should schedule appropriately.

This doesn’t speed things up per se, but it frees up unused resources giving us more area for more useful things.

- as of the M3 and M4, the capacity of AMX seems to be something like the ability to execute simultaneously two load/store type instructions (so 128B of data movement to L2) along with either an outer product or two vector-vector operations. What can we do with this?

One possibility is to hook up AMX to perform large (a page or more’s worth) of zeroing or otherwise filling memory. On the M1 about the best we can do for such zeroing is a sequence of `DC ZVA` instructions which will zero 64B per cycle (and M1 issues limit this to about 2/3 of a `DC ZVA` per cycle). The core could be boosted to aggregate two of these into a single transaction, executed every cycle; but doing this via AMX VGx4 instructions gets the same performance in many fewer instructions. If you want to fill with something non-zero, the savings are even better. Of course there is overhead, so some experimenting needs to determine the break-even point.

Another idea copied from the GPU might be to place a permute network between the Operand Cache and the execution units. This would allow a wide variety of data rearrangement “for free” if these permute instructions are fused with the subsequent “real” instruction that acts on the permuted data, ie the permute happens as part of Operand Cache lookup. This allows the provision of a whole new set of SVE instructions as SSVE with not much additional cost, and opens up the SSVE/SME unit to non-HPC but high throughput work like sequence matching and some forms of text processing.

Maybe (?) it’s also worth putting in that same data path a few very lightweight operations (`NEG`, `ABS` are obvious candidates, maybe also changing datatypes) so that, once again, we can fuse away as much trivial work as possible and ensure that every cycle our array of FMAs is active.
Given that AMX is a throughput engine, albeit a very low latency one, there’s much to be learned from the GPU and ANE in terms of ideas that might be infeasible on the CPU, where an additional two cycles of latency is a catastrophe, but are reasonable in other contexts.

OS Scheduling of threads

There are a few new patents in the area of OS scheduling.
(2020 https://patents.google.com/patent/US20210157700A1 *Adaptive memory performance control by thread group*
(2021) https://patents.google.com/patent/US20230040310A1 *Cpu cluster shared resource management*
(2022) https://patents.google.com/patent/US20230067109A1 *Performance islands for cpu clusters*
Details of these would detract from our primary hardware concern; the main points are
- all are based on tracking various data across the SoC; data which, even though it may not be available to developers (yet?) is available to the OS
- to some extent they represent previous ideas, but updated to match the existence of multiple clusters (ie the Pro/Max and Ultra lines).

- the first tracks the extent to which each core is stalling because it is waiting of either SLC or DRAM, and in response to this boosts the frequency of the NoC and/or DRAM. This seems to be an updated and more flexible version of the 2016 patent that (likewise, depending on if a core was frequently stalled, doubled the speed of DRAM); and the 2019 Cache Telemetry patent.
Feast your eyes on this glorious diagram!

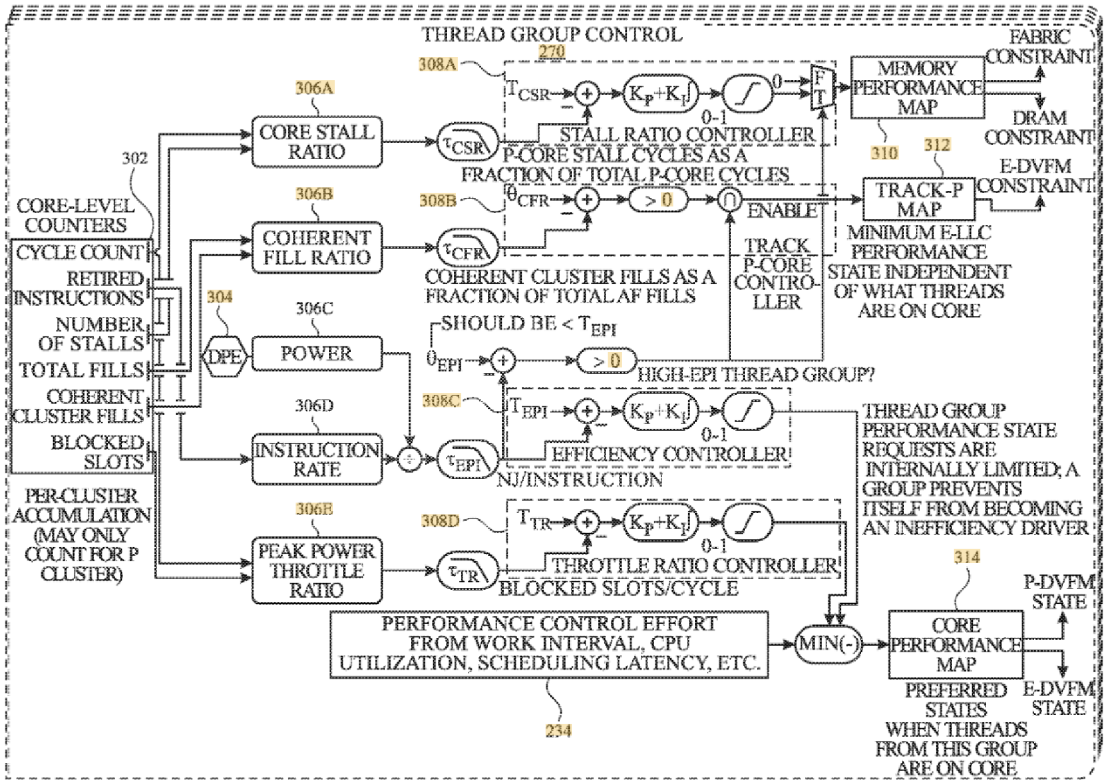


FIG. 3

There are a few non-obvious elements to this.

One, a fancier version of something we have seen before, is that if it's determined that a large enough fraction (the "coherent cluster fill ratio") of the cache loads are actually being produced by the E-cluster (or more generally, some other cluster), in other words we have something like a producer thread on an E-core feeding a consumer thread on a P-core, then the E-cluster gets a performance boost. (What about the reverse situation, if the P-core is producing faster than the E-core can consume? I suspect that will also be detected, and the relevant thread moved to a P-core?)

A second element is: consider what to do when your P-core is constantly waiting for memory.
One response is just to run the NoC and DRAM faster. This is appropriate if the code is throughput code, ie it is running through a large stream of data as fast as possible (think, eg adding one long vector to another).
Another response is to throttle the core. This is appropriate if the code is latency code, think something like pointer chasing. Making the NoC or DRAM faster won't speed up the return of successive pointer lookups much; most of the delay is intrinsic to every step in the process from core out to DRAM.
So what's the appropriate response? The heuristic the OS uses is to essentially look at how much energy the core is burning, which is rather clever! If heavyweight instructions (ie lots of SIMD) are being used, we are likely in throughput code, and the extra bandwidth provided by higher NoC and DRAM frequency is beneficial. But if only a few lightweight instructions are being used, we are likely in pointer-chasing style code, we are limited by latency, and giving the NoC and DRAM extra bandwidth is mostly a waste of energy.
(Of course these NoC and DRAM decisions are made across the entire set of clients! If the GPU is driving the NoC and DRAM at maximal bandwidth anyway, the CPU latency code will take advantage of that, even though it's still worth slowing the CPU core down until we're no longer running code that's throttled by DRAM latency.)

- the second considers the question of where to schedule the elements of thread groups. Remember that threads are grouped into (possibly short-lived) thread groups that are working together on a single task, and are possibly independent processes (eg a command line pipeline, or an app using OS threads to perform async IO). Ideally these threads are scheduled together in time (so that communication between them happens with a delay of a few cycles, rather than constantly making OS calls to context switch to a dependent thread then back again), and on the same cluster (so that communication can happen via L2 (faster and lower energy than via SLC).
But this scheduling on the same cluster may not be optimal when multiple of these threads all require the same resource. The most obvious case of this is AMX – if we have two threads that want AMX, it may be overall more performant to schedule them on two separate P clusters rather than on the same P cluster, giving them twice the AMX resources.
The same could also be true if either thread has a truly massive L2 footprint, or very high bandwidth requirements (either to L2 or to SLC/DRAM).

The way this is implemented is more clever than you might expect. You don't want to split up a thread group just because two threads use AMX, because maybe their AMX usage is light, or occurs at different times. So there is a monitor within the AMX unit that is detecting, on average, how full is the queue of pending instructions, and whether the instructions are split across more than one source core. There's something similar for the bus interface unit. These then report to the OS if we have a situation that's both an overload **and** that the overload results from more than one client in the cluster. At the next scheduling period, the OS can then look at the per-core metrics acquired over the last execution period and decide which thread(s) should be moved.

It's interesting to note that it's fairly easy to see a situation where threads should be moved for either AMX or bandwidth; in both cases you can see that a queue is filling up, and is occupied by requests from more than one core. While it's easy to understand, in theory, that an L2 could also be "filled up" by two threads working on different data in such a way that there's value to moving one thread to a different cluster, it's much less obvious to see how to actually discover this situation! Elements of a possible solution exist (tag in L2 which core first requested a line, tag lines that have been moved from L1 up to L2 and back again [so you can see L2 reuse activity, rather than just one time stream-through activity]) but it's not easy to see the entire solution. And in fact the patent punts on this point! It mentions that the same mechanism to be used to track AMX or bandwidth overloads can be used for L2 oversubscription, but leaves it that giving no details as to how you might detect this third condition.

The above regarding AMX sounds pretty clever, but there's actually a subtle problem!
Suppose the work to be done involves the multiplication of small matrices, for example two 16×16 FP32 matrices. This operation will involve some initial loads, then the primary outer-product-add loop, then some stores. After the initial loads, the subsequent loads can be executed simultaneously with the outer-product-add's, but there will be a four cycle latency between the each multiply-add and the next time we can submit such an add. (You can play fancy games with splitting the add off from the multiply to improve the situation slightly, but basically the adds form a series of dependent operations.)
If you were multiplying larger matrices you could interleave the work across multiple tiles and so avoid this bottleneck, but not for small matrices. This doesn't matter if you're multiplying one and only one matrix, but maybe you are multiplying a long sequence of small matrices.
And you may have a similar situation when acting on long arrays.
So, point is: you may have a situation where it looks like one thread is making aggressive use of the AMX unit (because the instruction queue is full of instructions from that one thread), and so by the rules set out above you schedule a second AMX thread to the alternate unit. But that is power-suboptimal! You burn extra power to start up the second AMX unit, but you don't in fact need to because you could schedule the second thread on the initial AMX unit, using all the unused execution slots where the sequence of dependent operations from the first thread are just waiting for their dependencies to complete.

This appears to have been done on the M4 (but not earlier), which is much more aggressive about co-scheduling AMX threads on the same AMX unit, especially when it's clear that one thread is leaving unused execution slots available. Presumably this is made possible by more informative metrics passed on to the OS, which describe not just how full the instruction queue is, but also what fraction of execution slots are being left unutilized.

- the third patent is essentially another power optimization patent. A single cluster runs at a single frequency, so if you have two threads that you believe should run on a P-core, both will run at the same frequency. But what if one of those threads is known to have limited performance demands (based on its work interval objects), while the other wants maximum performance? For the M1 you just have to

suck it up and schedule both on the P-cluster at high speed, but for the Pro/Max/Ultra you can put the demanding thread on one P cluster, and the less demanding thread on a lower clocked different P-cluster.
The patent also points out that there's value in occasionally "rotating" the clusters, so that the hotter cluster has a chance to cool down. (This allows the demanding thread to always run at highest frequency without thermal throttling.)

Both these two previous patents, interestingly enough, show as their prototypical design a system with two E-clusters and two P-clusters (each of four cores)...
It seems like Apple's first thoughts (M1 Pro/Max) were that maybe there was not enough "background" work for those sorts of machines to get full value from a 4-core E-cluster; and obviously that changed with the M2. Perhaps experience has shown that a surprisingly large amount of work (OS, interrupt servicing and IO, media, even UI and animation) can run well on E-cores, enough so that it's worth boosting a future Pro and Max up to 8 E-cores? (In which case maybe even also boost the lowest member of the family to two E-clusters, given how small an E-cluster is?)

In trying to understand scheduling, the most important point to remember is that these devices, iPhones, Macs, or whatever all engage in many very different tasks. You might be most interested in the goal of running a single thread fast, but the device also spends a lot of its time engaged in playing media, or what looks like asleep (screen off, but occasionally checking the network or running various demons), and scheduling also wants to optimize for these cases.
So recall the basic structure:
- threads are grouped (dynamically) into thread groups that are executing towards a common goal, and are ideally executed at the same time. (Obviously if a single thread group does not fill up a cluster and there are other threads/thread groups waiting, then they will also be scheduled.)
- each thread of a thread group is the subject of a constant stream of performance information from the core (energy usage, IPC and stalls, things like that), from the cluster as a whole (AMX or bandwidth oversubscription), and from other parts of the OS, (eg rate of storage or network IOs).
- this stream of data is filtered, integrated and extrapolated ("averaged") to produce a best approximation to what these values will look like in the next scheduling period
- based on these various values a "control effort" is suggested for each thread, basically how hard do I want to push the SoC over the next cycle, along with a suggested cluster (either P vs E, usually the previous cluster for cache affinity reasons, possibly avoiding some other thread because both use AMX, etc)
- the control effort is aggregated over all threads
- the control effort is possibly cut back if some safety machinery demands this; for example if the temperature indicators are rising too high or if the average power draw looks like it might be too high
- the control effort is converted into a DVFS plan (which clusters to fire up, how many cores to wake on each cluster, what frequency to run each core) along with some additional tweaks (possibly run DRAM and NoC faster if throughput code is executing, possibly run another cluster faster if a dependent cluster is constantly waiting for results from that dependent cluster).

This mapping is not always quite as simple as you might think, for example suppose you have two low-control effort threads. You might think the best thing to do is fire up two E-cores at minimum frequency, but if the work to be done is small enough, why not just fire up one core and execute the tasks sequentially?

I've primarily concentrated on the CPU-adjacent aspects of OS thread-scheduling and how the unusual aspects of M1 (E vs P cores, or core clusters) get handled, but if you're interested in the specific OS software details, you might want to look at (2020) https://patents.google.com/patent/US11422857B2 *Multi-level scheduling*. This describes scheduling done by first grouping threads into buckets (real-time, user-interface, user-initiated, background), then within each bucket grouping to an appropriate cluster (ie creating thread-groups) then the ultimate per-thread scheduling.

(OS Scheduling for the GPU and ANE)

It's easy to get so overwhelmed by this CPU scheduling that we forget that there are other coprocessors (eg GPU and ANE) that also require scheduling! (2019) https://patents.google.com/patent/US11119788B2 *Serialization floors and deadline driven control for performance optimization of asymmetric multiprocessor systems* discuss some of the relevant issues.
Much of the machinery is the same (eg constant stream of performance data which is filtered and converted into a control effort, a separate such effort for GPU and/or ANE; and a mapping of the control effort into the activation of some number of GPU or ANE cores at some frequency. But a different aspect to this is co-ordination between devices. In particular the patent describes tracking and handling the frequent situation where a core does a burst of work to create a GPU workload, then has nothing to do until a buffer is freed and it can begin the next GPU workload (and vice versa for the GPU). If the CPU and GPU are each independently optimizing their behavior, the CPU might constantly be deciding to power down (which costs energy, eg in flushing the cache) and the powering up again. By treating the problem as a unified flow of computation across devices, the OS can do a better job of understanding these idle periods and making better informed decisions about how fast to drive both the CPU and GPU (maybe run the CPU slower if the graphics job is the only real client? or maybe, since you know how long it will be until the CPU is required again, put it in a less aggressive sleep mode?)
The patent works through and considers various different cases, both the very predictable case (eg playing a game) where the alternation between CPU and GPU is fairly constant, and less predictable cases (perhaps something like training an LLM?) where there is alternation between CPU and coprocessor, but the exact timing is less predictable.

Packaging

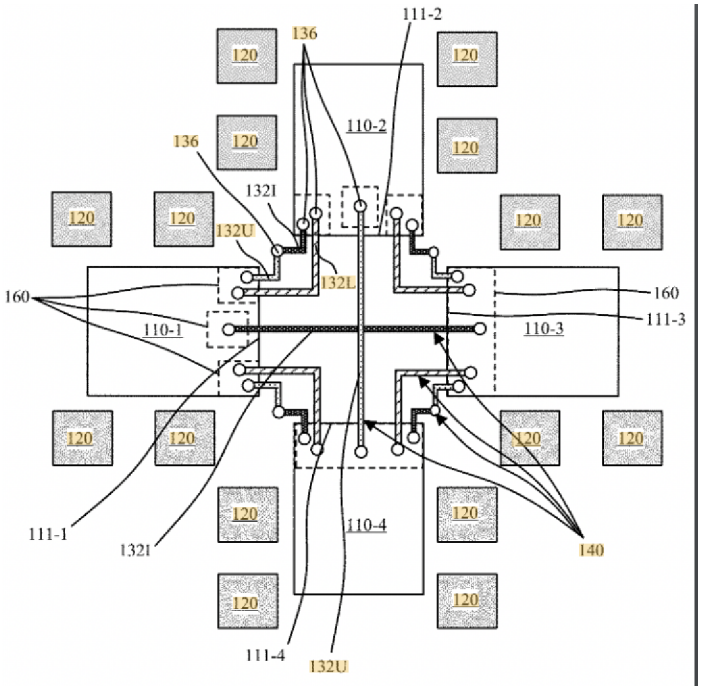
Packaging as always remains unclear.
The ideas we saw (manufacture pairs of Max chips, and build an RDL directly on top of the pairs) seem to remain relevant and (2021) https://patents.google.com/patent/US20230085890A1 *Selectable Monolithic or External Scalable Die-to-Die Interconnection System Methodology* builds on top of them. The idea now is that the "connection points" where the UltraFusion links will be placed, can be used in two

ways. One way is via an RDL built on top of the chip, and the suggestion is that this can be used for 2× and 4× sized sets, but this becomes problematic (yield reasons; you have to have enough good dies located adjacent to each other) for larger sets and for those an alternative connection will be used (eg 3D stacking and TSVs).

We know, from teardowns of the M1 Ultra, that it is not built using the above BEOL technology, but with a small interposer (ie something like EMIB). Apple also seems interested in tracking this technology, with an alternative set of patents like (2023) https://patents.google.com/patent/US20230223348A1 *High density interconnection using fanout interposer chiplet*.
The big idea of this patent, as I understand it, is we use the interposer fine pitch connection for signals, but the BEOL RDL for connections that don't need such fine pitch (presumably power/ground, maybe also clock?). Which means (perhaps?) that we're already using this sort of split system with the M1; an interposer that's obviously visible, along with an RDL that's not visible unless you know to look for it, said RDL handling the less sexy coarse pitch connections?

Who knows how ambitious Apple plan to be. But, assuming business conditions justify it, the idea seems to be that you
- fab Max-sized chips (basically limited by reticle, especially once high-NA EUV limits reticle area to ~400 mm^2)
- pairs of these Max-chips are tied together by back-end processing that builds an RDL between pairs (or in theory even quads) to form "monolithic" Ultra-chips
- these Ultra's can be further tied together using bumps under the chip (rather than along the side of the chip) to communicate with small EMIB-like router chips that tie multiple Ultra's together.

Another data point in this evolution is (2022) https://patents.google.com/patent/US20230299007A1 *Scalable Large System Based on Organic Interconnect*. This describes moving the UltraFusion bumps *below* the chip rather than along an edge, and moving more of the connector logic (buffers, routing, arbitration, etc) into a separate chiplet that connects to these bumps. Among other things, this allows for new geometries:



The diagram above shows how four Max-class chips can fit their DRAM along the sides, while still connecting together, with the 4-way logic embedded in the central (separate) chiplet that sits under the four Max chips connected to bumps in the dashed area of each chip.
In fact this scheme allows for “long distance” routing which can be as short as having four separate chiplets one beneath each Max, and some wiring (say a cm or so long) between the four router chiplets; or you could imagine one one of the Max chips, say the one on the right, being removed, and longer wires extending to a second cluster of three chips, giving a fairly compact layout for six Max chips plus DRAM.
This may be what we see with the M3 Ultra (and Extreme?) given that 3rd party die shots of the M3 Max do not show the UltraFusion edge connecter that we saw with M1 and M2 Max.

To the extent that there’s a big architectural idea here, rather than simply a scalable packaging

scheme, I think the idea is to move chip-to-chip routing off-chip.

If you think about something like AMD's Infinity Fabric, or nVLink, the building block is something like PCIe in that each chip provides a set of *n* (say 3, or 4, or 8) "links" where each link is a fixed number of lanes, and the wiring, buffering, PHY's etc for each link is on the chip of interest. This all takes up area on the primary chip; and even with multiple links, still somewhat constrains the connections and topologies that are possible.

The Apple alternative idea seems to be something like
- provide raw (and somewhat simple, in terms of buffering, routing, and arbitration) maximum bandwidth that a chip can handle via area bumps. Because the bumps only need to support a short distance connection very simple and small PHYs can be used
- move all the intelligence (PHY as needed for longer distance, buffering, routing, and arbitration) to an external EMIB-like chip which can be made on a cheaper process
- so now the smaller units (Max, Ultra) can be cheaper because they aren't paying the costs of highly scalable connectivity when not sold in large configurations. And the EMIB routing chip can grow in size and capability as required.

A scheme like this has the potential for somewhat arbitrary scaling independently of what happens to the base SoC's, by having a separate team separately design these routing/connectivity chips on a separate schedule. For example a later, more advanced, version of one these separate routing chiplets could handle additional buffering and protocol to allow for rack-to-rack connectivity (like nVLink between DGX boxes), or rack to rack optical links, without requiring this functionality to be built into the Max chips themselves.

More patents in this genre include (2023) https://patents.google.com/patent/US20230214350A1 *Die-to-die Dynamic Clock and Power Gating* (update of a 2021 patent) and (2023) https://patents.google.com/patent/US20240085968A1 *Dynamic Interface Circuit to Reduce Power Consumption*. The first is an extension of an idea we've seen already on the SoC: isolate IP blocks from the rest of the SoC via a small "interface block" which can negotiate things like when the IP block can power down, can handle requests to power it up, and can buffer other requests while power up occurs. This idea is extended to a collection of SoCs. The second patent gives more details of a possible implementation. The point of interest in both cases is that the examples give are based on four (not two) linked chips, ie on the mythical M Extreme, not an M Ultra.

Of course this all assumes that Apple have ambitions to scale up very large – rack-sized machines or machines like nVidia's DGX systems.
The patent trail certainly supports this. No business announcements do!
But of course Apple is always silent about any new project till the big announcement. Presumably all these scalability patents (and there are so many of them, going back years) have some end goal in mind...

Another unsurprising, but comforting, patent is seen in (2022) https://patents.google.com/patent/US20230299001A1 *Dual Contact and Power Rail for High Performance Standard Cells*. If you have any

interest in semiconductor logic, you know that the next two big steps are GAA transistors and backside power delivery; and Apple is looking ahead to once these can be used. This patent describes some ways of modifying standard cells given that backside power is now an option. A number of interesting ideas are presented including
- allowing power delivery from both backside and frontside, which reduced resistance
- alternating SRAM control signals (bitlines and wordlines) across backside and frontside. This allows bitlines to control a longer range of cells, reduces line-to-line capacitance Since lines are now twice as far apart), and requires only half the cells in a line to be toggled when a cell is to be changed, rather than all the cells.

Strange new (never to be seen?) products

Then there are the inevitable "what's this new product???" patents. For example:

(2021) https://patents.google.com/patent/US20210297653A1 *Displays with Viewer Tracking.* This appears to be a (non-head-mounted) 3D display that uses eye tracking to perform at higher quality than the usual such displays. It's *conceivable* that the grand plan here is to provide macs with 3D displays that allow for the same sort of 3D interaction as is provided by Vision Pro, though I suspect this is a long term plan, depending on a few years to see consumer response to Vision Pro and whether it enables new and interesting interactions.

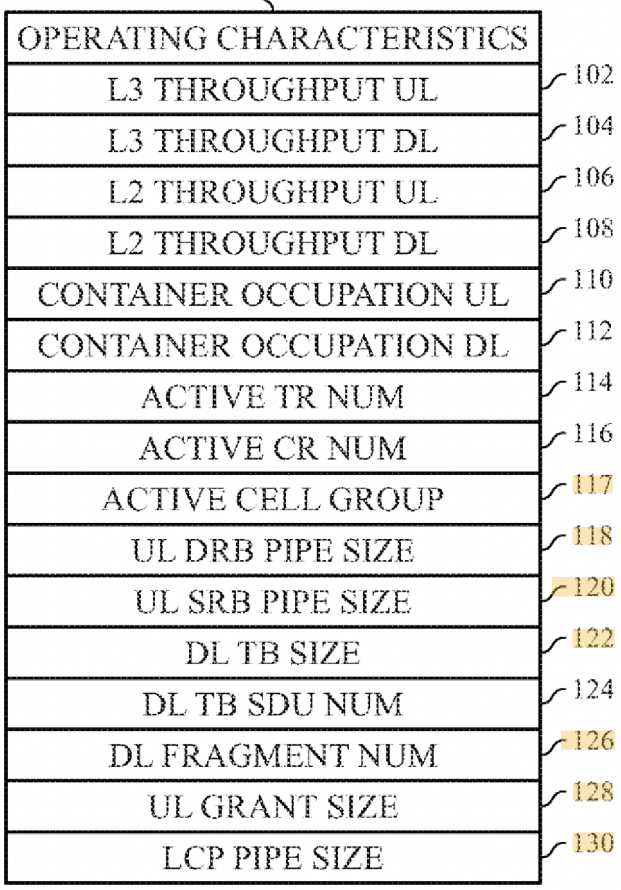
(2022) https://patents.google.com/patent/US11619830B1 *Display with a time-sequential directional backlight* is about a single display that displays two different images to people looking at it from two different angles. This seems to be a followup to the previous patent where someone realized "huh, you know what we could also do with this sort of display?" and it was patented just in case it one day might become useful.

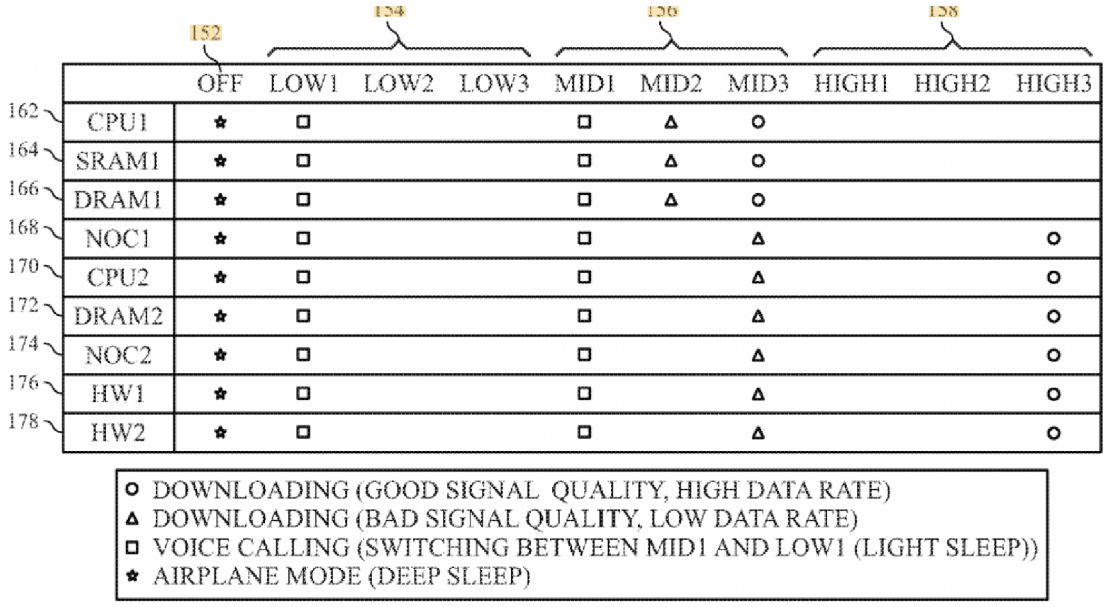
(2022) https://patents.google.com/patent/US20220244901A1 *Tiling Display System* describes display "tiles" that operate independently but can be joined together to create a larger seamless display. One could imagine strange possibilities here like Apple envisioning entire walls at home that are covered with a display. (Already a monolithic 65" display is at the limit of what's easily manageable in terms of fitting in a car, taking home, and setting up. Larger sizes like 75" or 85" seem like they need specialized white glove service which, as much as the cost, is an impediment to wider adoption. So a high quality tile-based solution might actually be quite popular. Not to mention over time you can grow from, say 2×2 to 3×3 and make your "TV" bigger without throwing away what you already have.

(2022) https://patents.google.com/patent/US20220375428A1 *Methods for color or luminance compensation based on view location in foldable displays* which is about what it says – foldable displays...

Here's another where who knows where it will lead: (2021) https://patents.google.com/patent/US20220366278A1 *Systems and methods for dynamic hardware configuration*. The idea is that in these

systems we have a large number of configurable parameters,





and while we won't go through what all of these mean, clearly "optimizing" the system, in the sense of making it fast enough to match user needs (but not faster that will be noticed) while using minimal energy is not trivial. So how to do it? In the past this has been done by more or less fixed rules, some hardwired, some programmed into registers at boot time, some set by the OS. But an alternative is to use everyone current favorite panacea, namely machine learning. So the idea is to (based on unexplained criteria, perhaps something like "are these familiar use cases") sometimes use the fixed rules, while at other times we use the suggestions of the ML model.

There are similar in spirit patents for other difficult decisions like what's the current optimal wireless mode (WiFi or cellular) or what's the optimal way to encode this block of video. Each of these is essentially an optimization problem within a huge search space, and right now "ML" seems a good generic algorithm for finding a *good enough* solution in a huge search space.

Then there is this: (2018) https://patents.google.com/patent/US11316594B2 *Robust ultrasound communication signal format*. Yes, exactly what it says, communication using ultrasound! What's this for?

Apparently (so claims the patent) there is a slow standardization going on to allow the connection of hardware to various devices in a wireless way. Of course we can do this today via eg Bluetooth, but Bluetooth is non-directional and can spread beyond a room, so that you land up having to choose from a list of item and to enter PINs and suchlike. The idea seems to be that a combination of getting the audio amplitude correct and directionality will make it fairly clear that the device you want to connect to is the one that's nearby and, in some sense, perhaps the one you are "pointing to", something like how RFID connections (Apple Pay, or iPhone proximity to a HomePod) are somewhat more robust in terms of a connection simply happening without the extra steps of a bluetooth or wifi connection. The bit rate is not great, so I expect this will be used just to negotiate the very first connection ("I want to talk to you") followed by a BT connection to negotiate higher bit rate stuff.
Will this be a great usability improvement, or will it, like Bluetooth beacons, turn into a usability clusterf-sck as every idiot company out there insists on using its particular hardware, API, and app, with zero genuine interoperability? I guess we'll know in ten years.

There's a collection of related patents, of which (2018) https://patents.google.com/patent/US11392409B2 *Asynchronous kernel* is an illustrative example. To my eyes (and I'd be the first to admit my eyes glaze over when it comes to the narcissism of small differences that governs most OS discourse) this looks like an attempt to move macOS into a more microkernel direction (think something like QNX). Clearly, at an abstract level, Apple (and anyone else in the OS space) should want to have the OS, like any other large performance-sensitive code, split into as many small parts as possible that can each run independently, to take advantage of the existence of E-cores, along with opportunities for truly hard security and isolation (eg dedicate an E-cluster purely to running the kernel parts of the OS, with user code simply unable to ever even see that cluster...) Many such projects have been tried, most have failed, though a few (like QNX in some use cases, or L4 running on the Apple SEP) have had their successes. Maybe Apple is trying again?

I assume something like this will first be used in non-visible situations (eg as a revamp of RTKit, Apple's tiny OS that, as far as we know, runs on the various companion core processors) to see it plays out in real situations. Then maybe, without ever actually announcing a change, more and more of Mach will be modified to fit this new architecture?

What's the competition (ie Nuvia) doing?

On a very different subject, you surely know the politics behind the creation of the company Nuvia, followed by its absorption into Qualcomm. Who knows how that will turn out as a business issue, but Nuvia have now filed enough patents that we can start to get a feel for how they view things, and where they might be slightly different from Apple.
At the 10 mile level, their big different idea that's visible so far is to make control hierarchical, so that whereas Apple have SoC-wide power and performance monitors, Nuvia also add cluster-wide such monitors. So, for example, Apple (at least to the extent they describe things in patents) provide a power budget for a cluster, with the SoC-wide controller deciding things like the cluster frequency, voltage,

and extent of L2 power savings. Nuvia delegate that to a per-cluster power controller. This allows them(in theory) to run each core at a different DVFS.
Now the obvious question is how this is valuable. The whole reason we run the Apple cluster at a single frequency (apart from sleeping cores) is that we don't want to pay the time costs of shifting from one frequency domain to another every time we want to access the L2 (or AMX, or L2 TLB, or one core communicates with another, or ...) The patent doesn't answer this question, but two possibilities suggest themselves.
- There may be occasions where it makes sense to run some of the cores at half (or even a third) the frequency of the other cores, for example if the task they're engaged in is low priority, or if the core is constantly missing to DRAM. If a core (or alternatively the L2, or L2 TLB, or an accelerator) is running at an integer sub-multiple of the primary frequency, there is less of a cost moving between frequency domains.
- If a core is engaged in low-complexity work (eg is mainly following pointers, at low IPC) then even at the same frequency as the other cores, it can probably run at a slightly lower voltage than a core engaged in heavy-duty SIMD.
By delegating the monitoring of each core (and L2, and so on) in the cluster, we allow for this sort of fine-grained management without overwhelming the complexity of either the SoC-wide monitoring, or the OS; both of those can model the cluster in a particular way, without worrying that at a very fine-grained level, either the frequency or voltage of an element of the cluster may occasionally shift from what's being modeled.
(2022) https://patents.google.com/patent/US20230093426A1 *Dynamic Voltage and Frequency Scaling (DVFS) within Processor Clusters*
(2022) https://patents.google.com/patent/US20220413581A1 *Dynamic Power Management for SoC-based Electronic Devices*

A second version of this is that we track the performance of prefetching into L1, L2, and SLC, both for each cache, and for the predictor that is driving each prefetch (eg stride predictor, region predictor, etc). This allows us to monitor, for each cache, whether prefetching needs to be throttled, and to implement differential throttling. For example for core 1 we can say that the first two quality levels of prefetching are allowed, but the next two quality levels are not good enough and should be throttled. We can implement the same sort of control at each L2, and likewise for the SLC.
(2022) https://patents.google.com/patent/US20220365879A1 *Throttling Schemes in Multicore Microprocessors*

(2022) https://patents.google.com/patent/US20230012880A1 *Level-aware cache replacement* is a differently worded version of something we've already described a few paragraphs above; the idea of tracking that certain cache lines (in this case cache lines from page-table) are critical and need to be given special cache treatment. It's kinda an obvious idea, so everyone tries to get a patent by using different language! Apple talks about "critical" lines (which has the advantage of also including I-cache lines), while Nuvia talks about "levels" meaning elements of the different page table levels. I'm surprised they didn't try harder with this one, like at least talk about an MMU cache and implement some

cleverness there.

The last two are fairly obvious, but much more relevant to Nuvia's (original...) target market of data warehouses.
The first is allocating clusters into "partitions", and giving each partition a bandwidth to memory. It's just like all the agent-bandwidth credit-based schemes we've seen for Apple, only now with the agents as (dynamic) collections of clusters.
The second is suppose a cluster or core is given a TLB invalidate instruction. This is (if it's at the OS level, not the hypervisor level) relative to a particular virtual machine. So add a small storage in front of each TLB recording all the VMs that have been allocated to this TLB (ie that this TLB might be relevant to). If the entries to be invalidated belong to a VM that's not present in this TLB, then we can just ignore the instruction.
(2022) https://patents.google.com/patent/US20230067749A1 *Methods and Systems for Memory Bandwidth Control*
(2022) https://patents.google.com/patent/US20230064603A1 *System and methods for invalidating translation information in caches*

Summary

One way to summarize everything we've seen, along with some idea of where Apple is headed, is to consider something of an aspirational design for the near future. As you read this, you can compare elements to the ARM Neoverse V2 as described in (2023) https://www.servethehome.com/arm-neoverse-v2-at-hot-chips-2023/ ARM seem to be lagging Apple a few years in terms of the algorithms they are using (and more in terms of their structure sizes), but it is nice to see that they do keep advancing, and are willing to keep evolving their design to the known state of the art.

We start with instruction flow.
Apple seem to now be using three decoupled address-generation engines to perform I-fetch/branch prediction.
Ideally all three operate as asynchronous queueing engines so that the addresses generated go into a queue serviced by a separate machine on the other end of the queue. In each case, doing this allows us to use a multi-level prediction scheme whereby most predictions are fast and low energy, while a few take an extra cycle or two, but hopefully values built up earlier in the queue address will buffer that delay.

- Prefetch is handled by using a variety of branch predictor that only predicts function entry points. For the purposes of prefetch, it's mostly good enough to treat a function as a single contiguous blob and not worry about the internal flow of control. So what you want is an engine that generates a stream of future function addresses, along with a length giving at least some approximation to how many lines from that address to preload.
You can make this work better by evolving it to a TAGE like scheme (ie incorporating varying amounts

of history). You can imagine various different elements to include in the history hashes indexing into the tables: function calls, function return points, and, perhaps, branch taken history.
Perhaps you can share some storage with the primary branch predictor (especially the indirect function call ITAGE predictor?)
It doesn't have to generate an address per cycle, so it can afford to be slow, and to share storage with other clients.

As instruction flow (ideally via prefetch) into the L1I they are pre-decoded for various purposes. One is to detect and tag all branch-type instructions. A second (in future, no evidence of this today) might be to classify instructions into various classes so as to simplify subsequent fusion.
Instruction (pre)fetch is accompanied by criticality indicators that mark the line being an instruction line (hold onto it hard in L2 and SLC), a line that was demand fetched (ie hard to prefetch, ie hold onto it even harder in L2 and SLC), a line that is being reused (ie has been previously used in L1I so, once again, yes, hold onto it a little harder because it might be reused again) and so on.

- Lagging behind the Prefetch Address stream is the Fetch Address stream. This generates a (Fetch address, trace length) pair, pretty much per cycle. Most traces are short enough that simply fetching across two cache lines will acquire the entire trace. You want to pull in, on average, more than 8 instructions per cycle to feed the machine downstream. This is do-able (on the edge...) if we assume something like average traces are about 10 instructions long, and the maximum fetch width (straddling two lines) is about 16 instructions long. But we can improve this fairly easily as I'll explain soon.

An important, and non-obvious point, is that going forward for optimal design we want to split prediction duties between Fetch Address prediction and the subsequent prediction stage. In particular Fetch should be trained on and should handle strongly biased branches without these ever making it to, and polluting, the full prediction machinery, and the full prediction machinery should handle Short Forward Branches (up to about 3 or 4 instructions forward) without this ever polluting the Fetch prediction machinery.
Done optimally this will mean that
+ strongly biased branches (like error/null tests) take up a few bits in the Fetch prediction machinery, but no bits in the much more expensive TAGE machinery, and no (entropy-free) bits consumed in the global history register(s).
+ Short Forward Branches are ignored by Fetch which always pulls in the instructions after the SF Branch. This will on average lengthen our traces (not just by the instructions after the SFB, which may not be executed; but by continuing the trace after the SFB basic block). The one to four instruction after the SFB will be marked as "tentative" in the instruction queue. (This is easy enough given that you already have all the info in the predecoding of the instructions.)

- Like the Prefetch queue, addresses generated by Fetch go into a queue, the other side of which is the machinery that accesses the L1I. As always, by decoupling via a queue either side (either the cache side or the address generation side) can occasionally stall without forcing the other to stall, and hopefully

the queue will buffer the stalling.

- The instructions from the I cache flow into an Instruction Queue headed for decode. During the stages while the instruction Fetch was being performed (TLB lookup then cache access) the full prediction machinery (TAGE etc) was thrown at the problem. This can result in the following outcomes:
+ if it's concluded that either Prefetch or Fetch have gone off the rails, everything after the offending (mispredicted) branch can be flushed and Fetch/Prefetch can be restarted.
+ if the misprediction is minor (a shortish Forward Branch) we may not have to resteer, we can simply mark the offending instructions as invalid and drop them in Decode
+ an SBF that has high confidence prediction is handled in the same way; either the one to four instructions after the branch are marked invalid (ie taken branch), or they are allowed through as valid (ie non-taken branch) BUT
+ an SBF that has low confidence prediction has these instructions tagged as predicated, to be decoded to predicated instructions
Net result will be that
+ both Fetch and Full Predict have larger effective capacity (each is doing what it does best, and not replicating the storage of the other for biased or SF branches)
+ traces are effectively longer, thus delaying the day until we have to predict two Fetch addresses per cycle
+ "trivial" branches (ie SBFs), even those hard to predict, will no longer cause flushes.

If we compare this to Neoverse we see the same sort of decoupled architecture, but V2 is requiring the Fetch queue to do double duty, both decoupling Fetch Address Generation from I Cache Access AND acting as a prefetcher. This seems elegant reuse, but it's sub-optimal as I hope the above walk-through makes clear; it doesn't allow for optimal performance of the three different jobs to be done because one engine is doing two different jobs.

On to Decode. IMHO it's about time for Apple to move to 10-wide decoding. Decoding is fairly easy to run in parallel, and the big inefficiency Apple has today is that the expensive hard part of the pipeline (8-wide Rename) is frequently not given a full 8 instructions to work on. This is a horrible waste of expensive silicon.
The most important reason I care about for this is fusion; if Decode converts two instructions into one. But there are other reasons it can occur.
My suggestion is (once again...) we separate Decode (10-wide) from Rename (8-wide) via a queue. Sometimes Decode will dump 10 elements in this queue, sometimes fewer than 8. If the decision is made to drop post-SBF "invalid" instructions at Decode, then we may want to widen Decode all the way to 12; alternatively maybe the machinery that moves instructions from the Instruction Queue to Decode could drop these tagged invalid instructions.
This machinery opens up the way for even more aggressive fusion. Right now fusion is of some, but, limited performance benefit because, mostly, the instruction slot that was removed by the fusion at Decode can never be reclaimed...

It's worth noting that, with Sierra Rapids Intel has moved to a similar idea, 6-wide Decode but 5-wide Allocate.
As far as Apple goes https://zhuanlan.zhihu.com/p/675322260 claims that the Blizzard core (M2 E-core) is 5-wide Decode, but 4-wide Allocate, so like Sierra Rapids in this respect. This doesn't necessarily imply Apple (or Intel) have an actual decoupling queue in place between Decode and Allocate, but it does suggest that Apple is aware of the issue, and so presumably will at some point implement such a queue when it makes sense.

Next is Rename (what I prefer to call Allocate). There are two big ideas I propose here.
- Make Allocate flexible in how much work it does. Right now Rename is scaled to execute 8-wide come what may. But a variety of different types of resources are allocated (everybody gets a ROB slot, but some ROB slots are different from others), most instructions (not all) require a source register name lookup, most (not all) require a destination register allocation, some require an LSU slot, and so on. Suppose we allow all these different resources to allocate each as fast as their machinery allows, with no promise that 8 have to happen per cycle. We can now get away with this because we have a decoupling queue between Rename and Decode. My expectation is that we can save some power, can allow that rare problematic sequences maybe only allocate 6 instructions worth of resources, but on average we're allocating more like 9 or 10 instructions worth.
- Virtualize all resource allocation in Allocate. Remember what I said that Rename performs basically two separate tasks.
+ One is to allocate identifiers that maintain ordering and dependency relationships.
+ Second is to allocate physical storage resources.
Traditionally these two go together, but that means that the expensive resource (physical storage in a register or an LSU slot) has to be allocated (and thus is unavailable) some time before it is actually required (at the point of instruction execution). We can make better use of limited resources by allocating ordering identifiers (which are just numbers) at Allocate, and allocating the ultimate resource at the point of execution. We've seen that Apple already seem to be doing this with LSU slots, and the academic literature has described how to this for registers for a while now.

Traditionally after Rename instructions are Dispatched (placed in some sort of storage waiting to be executed). I propose a new stage at this point, called something like Pre-Schedule. This will classify instructions into various (speculative) classes.
- One classification will be Criticality. Critical instructions will be marked as such (as per various papers I've referenced) and subsequently scheduled according to criticality.
- Another classification will be a prediction that the instruction will be long-latency (most obviously loads expected to miss in cache; but other possibilities like divide may also be useful here). This will be used to implement Long Term Parking (ie shunting off to some storage buffer all instructions dependent on a long delay instruction, so that they do not hog expensive slots in the scheduling machinery while we wait for their result).
- Given that this stage is sitting here anyway, and probably will not be time critical, maybe this can also be a location for implementing value prediction of some sort (if it's ever decided that that's feasible

and worth doing)?
- Finally, as mentioned, at Decode or Allocate we can detect which instructions are already executable (don't need to wait in an Scheduling Queue) and can bypass those instructions past the cost of the Scheduling Queue.

Scheduling and Issue then operate as before, though with scheduling driven by instruction criticality, not age.
With aggressive fusion, I think it's feasible that at least some pairs of instructions (eg logic+logic, logic+cmp, logic+add/sub) be double-pumped, ie both halves of the fusion can be executed in a single cycle.
As another aspect of execution, Intel on Granite Rapids managed to shrink FP MUL from 4 or 5 cycles to 3. So there may be scope, if Apple is willing to spend the transistors, to shave latency off some NEON FP operations.
V2 attempts to save power by providing an execution-unit local result cache to act kinda as a store for bypass results and to hold values if register writeback ports are oversubscribed (something we've seen Apple are also doing). But one could imagine an extension of this idea for Apple that provides some register storage local to the NEON unit. Right now even the simplest NEON operation requires two cycles, but maybe you can shrink this to one cycle if most of your operations are back to back writing to, then reading from, this local register storage?

We've seen a bunch of ideas for how to get more value out of the LSU, most aggressively by using it as an L0 cache. Given that V2 can perform load-store forwarding at the same latency as their L1 access (but LSU access is lower energy) maybe it's worth considering this seriously. It won't improve performance, but will help some with power.

It's been unclear (at least to me) how fancy a cache replacement policy Apple is willing to use for the L1D. But if V2 is using DRRIP (Dynamic Re-Reference Interval Prediction) https://en.wikipedia.org/wiki/-Cache_replacement_policies#Dynamic_RRIP_(DRRIP) for their L1D, I expect Apple is also using fancy policies (including the various flags I suggested earlier, like whether the line was fetched as critical, was prefetched, or has been reused). There's probably also scope for a zero-content cache slapped on the side of the L1D. I continue to believe that my hash-based suggestion for an effective way predictor has great promise, including, eg, making it easier to integrate a zero-content cache, or some similar pattern-based cache, into the L1D without a latency hit.

It's an interesting fact that, of all the improvements ARM describe to V2, the single most significant are the improvements to data prefetching.
Obviously Apple have the same functionality (tracking in virtual space, not physical space; Apple are ahead in capturing all load/stores in order at the LSU, though the noisier stream seen going into the L1D, and not the even noiser, and mostly filtered, stream missing in L1D). Everyone has stride prefetchers, with various fanciness to handle multiple simultaneous strides. I assume ARM's page prefetcher is essentially Apple's region prefetcher.

You can see something of a quick description of the different ARM prefetchers here (2015) https://users.cs.utah.edu/~rajeev/pubs/micro15m.pdf *Efficiently Prefetching Complex Address Patterns* and (2017) https://inria.hal.science/hal-01254863/document#:~:text=Best%2DOffset%20prefetching%20is%20intended,prefet%2D%20ches%20%5B33%5D. *Best-Offset Hardware Prefetching*
My understanding is that Apple's AMPM scheme handles the same sort of cases as SMS (essentially we reference a data structure via a pointer, then a pattern of offsets relative to that pointer) and BO (fancy version of a next line prefetcher); and Apple's Content Directed Prefetcher is a more powerful form of CMC (Correlated Miss Cache, and basically to handle walking down pointer based structures like lists and graphs).

So it seems like, in terms of basic algorithms, Apple and ARM are mostly tied for D-prefetching. The one big difference is the following:
- if you do your prefetching based on what see at the L1 cache (eg the stream of L1 cache misses) essentially all you can train on is *addresses*. You can try to correlate these in various ways, and that will often get you effectively what looks like a way to prefetch graphs or other pointer-based structures, but only if those don't change, and only after a few misses to the data structures.
- if you do your prefetching based on what you see at the LSU (as Apple does, but apparently no-one else) you can also train on the *content* of what is loaded (ie what's returned from a load, and how it is used. This allows you more rapidly to detect patterns like pointer chasing, and to try to react proactively.
Apple does this using the 2016 *Content-directed prefetch circuit* patent. Superficially this is based on lines coming into the L1, and so could be utilized by an L1-based prefetch design, but the details to make it work (in particular the undescribed "filtering" step) seem like they rely on knowledge provided by the LSU.

The nature of the problem is described in (2016) https://people.inf.ethz.ch/omutlu/pub/enhanced-memory-controller-for-dependent-loads_isca16.pdf *Accelerating Dependent Cache Misses with an Enhanced Memory Controller,* which, very unusually, has an Apple co-author! (Note that, apart from describing the problem, the rest of the paper is not [yet?] relevant to Apple; the paper describes a way of detecting the pointer chasing in the CPU and then moving it up to the SLC or memory controller so that the chasing loop is executed there rather than on the CPU. Who knows if that idea will ever make it to a real design?

The precise details as given in the paper seem to me crazy in terms of maintaining consistency and suchlike; I'd implement this not as *moving* instructions to the memory controller but as *executing shadow instructions* at the memory controller, to generate a stream of loads that are sent as "prefetched" lines to the L1D and which, hopefully, speed up the execution on the CPU.
BUT how about this?

Give the SLC an accelerator framework (like AMX uses the L2 accelerator framework) so that a CPU can explicitly send simple short codeblocks to execute at the SLC, primarily things like

+ pointer chasing loops
+ searching for a value, or
+ simple blits/data transforms

You'd want to think through the precise details carefully because the only way to make this work well is to have it thinking somewhat like a GPU – ie independent instances of these codelets will run on each of the multiple SLC blocks on a Pro, Max, or Ultra, with synchronization barriers expensive and hopefully rare.
But this seems like an interesting idea going forward, a half-way step on the way to PiM.)

Of course there remains ample room for different implementation details (including non-obvious tweaks, like the way Apple runs the prefetchers more aggressively after the L2 cache is powered up after having been powered down and lost its contents).
I'm unaware of promising and practical really new D-prefetch ideas; perhaps the next step is to accept this and concentrate on making the cache effectively larger, via zero-content, pattern detection, and compression?

If you want to compare the V2 SPEC2017 results with Apple, you can find Apple SPEC2017 results at:
https://github.com/TomyLemon/Apple-CPU-Benchmarks
Eyeballing it, Apple is mostly still slightly ahead on IPC (as of M1 and M2), but not dramatically so [apart from dramatically better x264 and exchange2, which may reflect compiler improvements – that's usually what massive SPEC performance jumps mean]. Let's see what M3 brings, both for SPEC2017, and for my wish list above of CPU core improvements!

At this point you may want to look at the equivalent RISC-V situation: (2023) https://www.servethehome-.com/ventana-veyron-v1-risc-v-data-center-processor-hot-chips-2023/
Veyron V1 has made some, uh, idiosyncratic choices...
The use of a single large I-cache is interesting. I could see Apple growing their I-cache larger over time (perhaps with some smarts so that it could be scaled by quarters from say 128KB to 512KB, depending on use patterns). I'm not sure bypassing the L2 is worth doing for Apple, but maybe so?
A second interesting data point is their use of clustering in the TLB. Time for Apple to start doing this, once the current round of TLB modifications (better support for hypervisors, and support for various different page sizes) is done?
You can see how they're definitely a small team, making expedient but far from optimal choices (as we see in, eg, the symmetric pipelines, the simple prefetch, or the low SPEC IPC compared to ARM and Apple).

How is performance evolving?

With each new Apple SoC release there is grumbling that performance is no longer increasing, often linked to various claims about how TSMC is slowing down, or how every competent engineer at Apple left to join Nuvia.
Is there any reasonable substance to these complaints?

Let's look at Geekbench 6 single-threaded which, while not perfect, has the merit of having data easily available, and averaged over a wide range of devices.

Below is a more or less reasonable average over devices, trying to stick to iPads but using an iMac for the M3 results; Pro, Max and Ultra designs sometimes run .1 or .2 GHz higher, which combines with larger SLC to give us 2 or 3% higher score.

Out[•]//TableForm=

	M1->M2	M2->M3	M3->M4	M1->M4
Overall	1.12	1.20	1.21	1.62
GHz ratio	1.09	1.17	1.07	1.38
	IPC	IPC	IPC	IPC
File Compression	0.99	1.01	1.04	1.04
Navigation	0.95	0.97	1.12	1.03
HTML5 Browser	1.22	1.03	1.12	1.40
PDF Renderer	1.04	0.99	1.18	1.23
Photo Library	1.01	0.99	1.22	1.21
Clang	1.06	1.01	1.04	1.11
Text Processing	0.98	1.00	1.10	1.08
Asset Compression	1.00	1.01	1.06	1.08
Object Detection	1.18	0.96	2.03	2.30
Background Blur	1.05	1.04	1.25	1.37
Horizon Detection	1.00	0.97	1.02	0.99
Object Remover	1.07	1.00	1.11	1.19
HDR	1.00	1.02	1.11	1.13
Photo Filter	0.98	1.05	1.12	1.15
Ray Tracer	0.98	1.15	1.07	1.21
Structure from Motion	1.06	0.97	1.04	1.07

SPEC2017 numbers basically replicate this pattern of ~60% improvement from M1 to M4.

The first, obvious, point is that picking up ~20% improvement each design is hardly something to complain about! Those improvements add up.
So complaints about raw single-threaded performance are unfounded.

The next complaint one hears is that since the M1 it's all been GHz, not IPC, and that this is not sustainable.
There's more truth to this complaint, so let's look at it in detail.

About 17% of the overall improvement has been IPC, so call it about 5% IPC boost per generation. Not nothing, but also less than the GHz contribution.
To be fair, simply maintaining flat IPC while substantially boosting GHz is non-trivial! One criticism of prioritizing IPC over GHz has always been the claim that IPC at low frequencies is easy, but it gets harder as you boost frequency because it's harder to hide DRAM latency (along with other issues like it being harder to run the cache at the relative latencies). If we look at the individual scores we see that Apple has mostly been able to sustain at least flat IPC; occasional slippage in one benchmark, but usually picked up the next year.

The other part of the complaint about relying on GHz is that it's not sustainable. Well, of course nothing

is sustainable, relying mostly on IPC will also hit a wall at some point. But one's intuition is that there's less runway left for GHz, and of course it is true that in spite of amazing things Apple has done to limit the energy costs, higher GHz does cost more. So the more interesting issue isL why did Apple start doing this? Just to chase PC benchmark scores?

Obviously I think that's a silly analysis, so can we do better? I think we can.

M1 and M2 were on an N5 process, M3 and M4 on an N3 process. Perhaps the most striking feature of N3, widely commented on, is that while frequency goes up some (or equivalently power goes down some), and while logic density increases by about 60%, SRAM density increases by about 5%. This has implications for any IPC improvement that is based primarily on more SRAM, ie things like branch predictors that try to memorize a lot of context.

So what to do? An obvious immediate solution is to do what Apple did; pivot to retaining the same sorts of structures in the same sorts of sizes, but restructuring the pipeline (and a whole lot else) to reach higher GHz without exploding the power budget. That's a good solution for a few generations, but you can only slice a pipeline so fine.

Another solution is to add more logic around the SRAM, to get more value out of it, more logic per kB of storage. This is the world of, eg, smarter caches (cache compression, zero-content cache, and suchlike).

Yet another solution is to add more execution rather than more prediction. Thus 9-wide (M3) and 10-wide (M4), along with 8 rather than 6 integer execution units. One can also make the execution more powerful (ever more fusions, ideas I've suggested for even wider Fetch along with even wider Decode coupled to Allocate via an elastic queue). ARM with the Cortex X925 have added a load unit so that they have essentially three loads, a store, and an ambidextrous unit, compared to Apple's two loads, a store, and an ambidextrous unit. There might be value to Apple adding a second ambidextrous unit that can give either four loads per cycle or three stores or can provide an extra pipe to handle AMX/SME/SSVE instructions while the other load store pipes are doing load/store work. The same X925 added two more NEON pipelines for six pipelines. Once again, something like this starts to make more sense in a world where logic has become a lot cheaper than memory. Another version of this for Apple might be to boost the existing four NEON pipelines to handle SVE. SVE as an ISA has the advantage over NEON that some degree of overhead (branches testing the end of loops, trailing loops to handle values that didn't fit in the main NEON loop, clever things that can be done with predicates) become available. If Apple adopts 256b SVE they also pick up some degree of flexibility; NEON is available in four pipes for known short vectors (of which there are quite a few) while SVE is available for loops of unknown, or known long, vectors, initially perhaps in two pipes (two NEON pipes work together on the upper and lower halves of an SVE vector); later this can be expanded in various directions, eg boosting to six NEON pipelines like X925.

Along the same lines of course SME and SSVE are present for even longer vectors, and once again logic over SRAM justifies adding logic-heavy functionality to that unit every year.

Yet another strategy for boosting performance is to add more instructions. Obviously SVE and SME are the extreme version of this, but ARMv8.8 adds the so-called MOPS (Memory Ops) instructions which allow for higher performance copying and flood-filling of memory (memcpy, memset). These won't

double the speed of your memory movement, but they should allow for the same speed of memory movement as today, but with much simpler code. The result will be less I-cache pressure and fewer branch prediction slots used, so some nice second order effects. (The instructions have been designed to learn from Intel's mistakes in this area, so hopefully they will always be at least slightly faster than the pre-existing alternative, meaning there's no reason not to use them.)

Likewise ARMv8.9 adds the CSSC (Common Short Sequence Compression) instructions which are a few integer instructions performing common tasks that so far have required multiple instructions (abs, min, max, popcount, count trailing zero's). As far as I can tell clamp is not on the list, which seems a strange omission, hopefully to be rectified in an ISA update. BTW M2 is at ARMv8.6 (technically it omits one crypto instruction only used for a Chinese crypto algorithm); it's possible that M3 or M4 are at ARMv8.7, I haven't seen anything either way.

The primary reason SRAM has not shrunk with N3 is wiring density; the N3 transistors are smaller, but the wiring is not, and what limits how close the transistors can get together is now the density of the wiring. The immediate solution to that is to move some wiring to the backside of the chip (so-called BSPD, backside power delivery). Even just moving power to the backside helps alleviate congestion, but of course the more wires you can move the better. Apple has been thinking of the consequences for designing an optimized SRAM given this possibility (2022) https://patents.google.com/patent/US20230298996A1 *Backside Routing Implementation in SRAM Arrays*.

We can expect BSPD (maybe with some backside signal, maybe not) with TSMC's A16 node. Between now and A16 we will see the N2 node, which gives us GAA (gate all around, higher performing transistors, but not much alleviation of wiring congestion) so I expect another two years or so of primarily the sort of thing I've described above – somewhat more GHz with N2, and probably a lot more logic in execution units (SVE?), but I don't expect a dramatic boost in IPC, mainly just the same slog as the past three years, a percentage here, a percentage there, many changes boosting one benchmark (and so some subset of workloads) substantially but not affecting others.

Another way you could analyze the above data is that Apple has internal priorities as to what's most important to improve (obviously everything browser related, maybe PDF picks up a lot of generic "make the user interface faster" functionality, from the start AMX/SME has been supposed to improve aspects of ML, especially layers that for whatever reason cannot run on either ANE or GPU and ML is only going to get faster). Meanwhile other functionality is ignored, for example Horizon Detection) is probably limited by cache bandwidth, and Apple has no real reason yet to bother improving it (though adding an additional load pipe probably would have a ~30% IPC boost for that benchmark as a side effect...) Likewise the two compression benchmarks may be limited primarily by cache bandwidth, already discussed, and the speed of recovery from branch misprediction (hard to improve, and the data unpredictability driving the branch mispredictions is also tough to improve upon).

An obvious aspect of this is that it's very difficult to benchmark code with a large instruction footprint; neither Geekbench nor SPEC in its different versions, even pretend to do this. But large instruction footprint (and related issues like the speed of dynamic linking, and calls from one dynamic

library to another) are an important part of real Mac app performance. Improvements Apple makes in these areas will mostly be invisible to benchmarks (except perhaps well-crafted browser benchmarks). Likewise if they make changes that specifically speed up issues related to Swift or SwiftUI (perhaps things that improve bounds or NULL-testing? or to the hashtables used by Swift for various purposes), that has real user benefit, but might be irrelevant to C/C++ code.
Likewise for changes that primarily speed up OS functionality (eg changes to the MMU/TLB system).

In this analysis it's less that Apple can no longer find IPC and more that design is getting harder and slower, so the IPC gains they are achieving are targeted at particular types of code. There are no longer many interventions available that substantially lift *all* code, so they pick and choose, which has the effect of ignoring benchmarks that don't match areas of Apple's concern.

A book recommendation

At this point, if you can't get enough of this stuff, I'd recommend you read the book *The Goal* by Eliyahu Goldratt.
Superficially this looks totally unrelated to the issue of designing hardware, it appears to be a book for MBAs about how to handle finances, run a plant, and perhaps something about work/life balance. But what it's *actually* about is queueing theory! How the interactions between dependencies and fluctuations result in non-obvious outcomes. One of the signs that you're operating with math at a dangerous level is that you think things "average out" in most circumstances; one of the signs that you truly get math (and plenty of STEM PhDs are not at this level) is that you understand in your gut how non-linear processes dramatically reshape stochastic input.

So, you might think you don't care about manufacturing, but manufacturing is like running a CPU pipeline; items have to pass through a sequence of stages and any lost time at an earlier stage can never be recovered (this is where the non-linearity comes in, in the form of a ReLU function). And the exact same vocabulary and concepts of manufacturing are relevant to a CPU - identify your bottleneck (having correctly defined bottleneck...) and then ensure that
- it's never idle
- every item it works on is valuable (quality control before the bottleneck, not after)
- every item it works on is needed *now* (don't use your bottleneck on non-critical items)
- do items *really* need to be processed by your bottleneck? There are no alternatives?
In the case of a CPU, for example, the bottleneck is probably Rename. So let's feed it with a front-end that's bigger than it (apparently) needs to be so that there's always a deep queue of instruction to be fed into Rename, even in the face of glitches in Fetch. And ensure as best we can that that every instruction is valid, not a dud (branch misprediction) by augmenting the rapid predictors used by Fetch with slower but more powerful predictors that execute after Fetch, but which can exert quality control by removing mispredicted instructions before they reach Allocate.

Allocate has to operate in order, so there are limits to what can be done in terms of the third point, but for some code bottlenecks are either the four NEON pipelines or the 4 load/store pipelines. In both

those cases tracking criticality (the one missing part from everyone's designs, including Apple) can result in better execution by delaying non-critical instructions until there's a temporary lull in the use of this particular resource.

A version of the fourth point is the recent changes to the branch units. We still have one complicated branch execution unit that handles error conditions (like branches that are mispredicted, with everything that's then required to handle the misprediction *and* retrain the predictors). But most branches are not mispredicted, so we can first send them through an alternate lightweight branch execution unit that simply validates the prediction was correct, and only Replay them if the prediction failed – a way of bypassing the bottleneck of a single complex branch execution unit.

And for gods sake use appropriate metrics! So much CPU discussion is like the most foolish behavior in the book; an obsession with narrow metrics that have some limited value but which miss the big picture, which try to optimize something local while ignoring the system as a whole.

2025

CPU

Use of biased branches

It has long been known (since at least the 90s) that many branches are either almost always taken, or almost never taken, ie so-called *strongly biased*. Even if we strip loops out of the reckoning, the reason for this is fairly obvious: a lot of branches are essentially some sort of error checking, "`if(ptr!=NULL)`", "`if(error)`", "`if(index<length)`", "`if(divisor!=0)`" and so on.
I don't know the numbers, but I would guess that (especially in a safe language like Swift, which is providing a lot of extra safety checks behind the scenes) maybe a third to a half of all conditional branches are strongly biased in this way.

How can we take advantage of this fact? Two 90's ideas were
- split the strongly biased branches into a separate branch table (for reasons that now seem irrelevant, having to do with hash collisions)
- use strongly biased branches to build longer traces (this is the idea of trace caches).

Apple seems to have decided to look at biased branches maybe around 2020, and came up with a stream of ways to exploit them. The end result is a fairly sophisticated restructuring on the entire Front end, with many new elements. Following the patents in the right order, we can see something of how their thinking evolved, from an initial idea (a nice tweak which maybe is in the M3) to the substantial redesign which is maybe in the M4.

(2022) use biased branches only in the Next Fetch Predictor

Let's start with (2022) https://patents.google.com/patent/US20230244494A1 *Conditional Instructions Prediction.*

Recall that the current instruction flow is that when instructions are loaded into the I-cache (ideally by prefetch, but if necessary by a fetch miss in the I-cache) they are pre-decoded and various interesting data about each line is recorded in the line (like the presence of branches and their types). This can then be used in various way by subsequent Fetch.

Suppose now that
- we have a *conditional branch bias* predictor associated with this part of the front-end, which predicts that certain branches are strongly biased
- along with all the other pre-decode hints we store in a cache line, we also mark these strongly predicted branches (one bit) and their branch direction (one bit).

So we now have a flow that looks like:
(By Prefetch the diagram simply means the stage at which instructions flow into the I-cache, for whatever reason.)

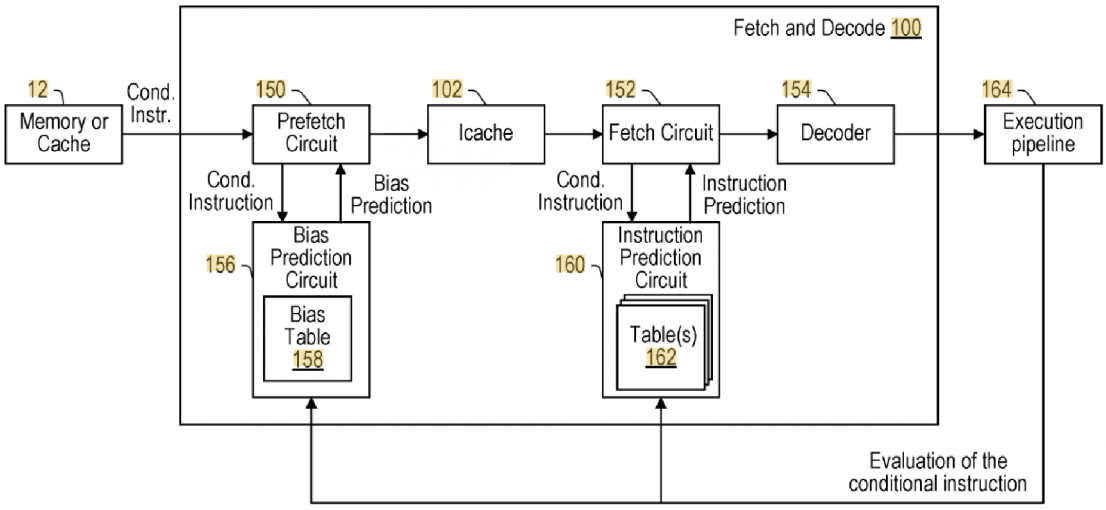


FIG. 1

A strongly biased taken branch is almost equivalent to a goto (an unconditional branch); a strongly biased untaken branch is almost equivalent to nop.
How can we then use this info?

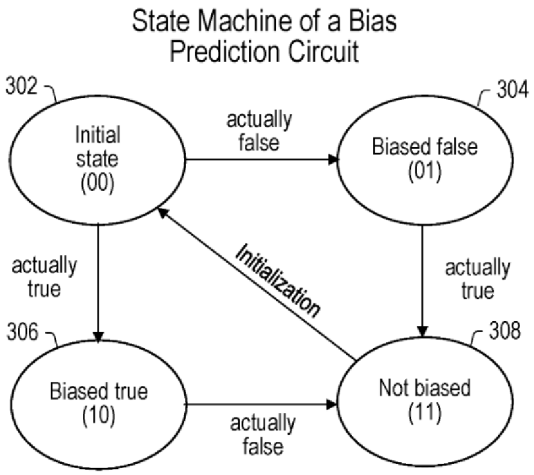
The first trick is to realize that, *as far as the front end is concerned*, these instructions have become either a goto or a nop. They can be treated that way right up to *execution* of the branch – at which point, of course, we need to perform the comparison to be sure that the bias prediction remains correct.

The second trick to realize is that a branch unit handling these can be very simple. The expense in a branch unit is not the branch comparison, it is all the machinery to handle
- training all the front-end branch predictors (this is on-going for something like TAGE, even for correctly taken branches)
- handling what happens when a prediction was incorrect
We don't need the training machinery for a biased branch, we just need *something*(...) to handle the rare case when the bias prediction was incorrect.

This initial patent, and it's companion below, makes use of these two ideas in the simplest way.
Using the first idea, all branch prediction associated with these strongly biased branches can be switched off. For example we
- don't bother to run TAGE after fetch to check the TAGE prediction (power saving).
A second possibility is that we
- don't bother to *train* TAGE on these trivial, information-free, branches [because they are always taken or always not taken], meaning that our branch machinery and branch history can store substantially less information but still be just as accurate!
Apple don't explicitly mention doing this, but everything else about the design suggests this is an additional feature.

Another way you could use this biasing (the patent doesn't mention this but it seems an obvious idea) is to inform the I-Prefetching, as a kinda fancier next-line prefetcher. If we fill in any branch in a line newly added to the I-cache as a strongly biased taken branch, it makes sense to use that as a hint to I-prefetch the target address of that strongly biased branch.

The patent is somewhat vague on exactly how this front-end bias predictor works. The part that's obvious is we have a table, indexed by a hash of the PC, and a state machine that updates the (two bit) contents of each table entry.



Then there's obvious correction machinery – as soon as we detect that a bias mispredicted, we want to move the state to not-biased (as in the state machine above) and update the relevant bits in the I-cache.

What's not specified in the patent (but probably present) are a few various pieces you'd use to improve performance. For example

- I'd hash the ASID along with the PC, so that the table would continue to work well across context switches.
- I'd probably add a tag (not necessarily a full tag, just a micro-tag, again from the PC and the ASID) to limit aliasing.
- I'd have to look at the numbers, but it might be worth making the table 2-way associative (ie two entries for each hash index).

Overall very nice! Probably a substantial win in both TAGE power and area, at the same prediction performance.

(2022) use biased branches to provide a simplified branch unit

The above patent has a second part (as is common, these two things are kinda mixed together in two patents that mostly look the same, but I will treat them separately) as the patent (2022) https://patents.-google.com/patent/US20230244495A1 *Conditional Instructions Distribution and Execution*.

The first patent was a way to exploit the fact that most branches are strongly biased.

The second patent is about exploiting the fact that most branches are correctly predicted, and that we can easily track the confidence of our predictions. (A trivial example of this is any simple two-bit predictor from the 90s with strongly taken/strongly not taken states and weakly taken/weakly not-taken states. We can treat the weak states as indicating low confidence predictions.)
Obviously strongly biased branches are high confidence predictions, but you can also be very confident in a prediction that alternates, for example each time through it flips from being taken to not taken, depending on whether a loop counter is odd or even.

The issue now is that (at least for M1 and M2) we have two branch handling pipelines, and both of them need
- a tiny amount of circuitry to "execute" the branch (test a flag, or whatever),
- a small amount of circuitry to route the comparison result to train the branch predictors
- a large amount of circuitry to deal with a misprediction (in which case we need to send instructions to the front-end to flush everything queued up and start fetching from the correct address, to the ROB to flush various instructions and roll back to a checkpoint, to the middle-end to flush instructions, a different set of signals to the branch training/updating machinery, etc etc. Huge drama.)

The insight of the patent is that *most of the time* this machinery is not required; most branches are correctly predicted. So why provide multiple sets of this recovery machinery?
Instead we provide one or two lightweight branch execution units and a heavyweight unit. We preferentially send branches to the lightweight unit, only sending them to the heavyweight unit if we're not confident in our prediction.
If the lightweight unit says the branch was mispredicted (hopefully a very uncommon event, requiring both the predictor to fail *and* our confidence in the predictor to be incorrect) then we resubmit the branch to the heavyweight predictor. My guess is the actual implementation is the same as we use for speculative scheduling – the branch is issued from the scheduling queue, but is also held in that queue. After one cycle of execution either the hold is released (no replay required) or we need to re-execute the branch down the other, paired, pipeline, in the heavyweight branch execution unit.

It's interesting to consider if this idea might be applicable to other pipelines. The case that springs to mind is the load/store pipeline where most load/stores happen without drama, but a few require replay (load/store aliasing) or even an exception. Does it make sense to use the existing load/store aliasing predictor to predict load/stores that are problematic and replay them to the heavyweight ambidextrous pipeline, which is set up for handling exceptional cases, then make the other load and store pipelines lighter weight? We are already holding loads for a few cycles to see if need replay, so forcing them, and stores, to replay for any reason (including exceptions) by routing them to the ambidextrous pipeline seems like not much effort, with the possibility of saving some area in the other three pipelines?

The general consensus seems to be that the M3 and M4 both have only two branch pipelines (so one heavyweight and one lightweight) but the patent explicitly points out that everything about this works fine with more pipelines (eg two lightweight and two heavyweight), you just need to slightly tweak how an instruction gets routed and replayed in the heavyweight pipeline.

f there is a common theme running through both of these patents, it's "support for a wider machine"... I've suggested, in various places, ways in which Apple could bump the effective width of the A17/M3 to 10-wide without paying much of an area or frequency cost, and these patents, while by themselves very nice additions to say an M1 or M2 class machine, also very much fit into the sort of thing you would need to do if you wanted to widen the M3. (Which is of course what we saw with the M4.)

(2023) more aggressive use of biased branches

In 2023 we see more sophisticated use of the biased branch idea. Once you have a good idea, where else can you use it?
Another class of biased instructions is indirect branches of various sorts. One example of these is where your code includes various functions optimized to the target CPU (does it have SME? (soon maybe) does it have SVE? should it route to either a SW AV1 decoder or to the HW decoder, ...) where at initialization time a function procptr is set to the appropriate version of the function and never touched again.
Another version of this is virtual ptrs where the class is set once and then rarely or never changed.
Yet another version is a function A that is only ever called from function B, so its return is basically always a goto the call site in function B.
Moreover, just like with error detection and safety, I suspect Swift has a number of additional places where it uses indirect branches to provide some abstraction (generics, protocols, that sort of thing), but the indirect branch is almost always to the same location.

(2023) https://patents.google.com/patent/US20250036416A1 *Biased Indirect Control Transfer Prediction* handles all these cases in essentially the same way as biased conditional branches.
We track, after execution, whether each indirect branch is strongly biased.
If so we place a record in a table at the entrance to the I-cache, and rewrite indirect control transfer instructions as effectively direct control transfer instructions (goto's rather than procPtrs or returns).
These indirect branches then become "invisible" to the front end of the CPU, including now the IT-TAGE indirect branch prediction machinery.
And once again we have two types of branch units, with these biased indirect branches going through the "lightweight" unit (with a replay mechanism) than the standard branch unit.

We're still not done!
If you follow any sort of ISA discussion on the internet, you will know that there is tremendous arguing between those who despise the use of the conditional select instruction and those who love it. The basic problem is that `csel` (in all its variants) is a fantastic instruction for handling *unpredictable* data (because it doesn't use the control flow mechanisms, and doesn't result in flushing or mispredictions when data is unpredictable), but it's sub-optimal when data is predictable because in that case using a branch will be handled by the branch predictor reducing some latency from your instruction flow.
Making things worse is the problem that even if you know which is optimal of csel or a conditional branch, how do you tell the compiler?
Can we get the best of both worlds?

A third version of the idea (stuck at the end of the first patent) gives us a partial solution.
Once again we have a table at the entrance to the L1I. Once again
- during execution we look for biased (ie almost always executed the same way, either always the true or always the false option) csel instructions,
- place a record of such instructions in the table,
- and rewrite the instructions as the non-conditional variant. This allows the non-conditional variant (basically a `mov` to execute in zero cycles without having to wait for the condition flag to be evaluated).

Of course at some point we will have to validate the instruction! I assume this is done when the `csel` is executed (now ignoring the two bits that indicated it was a biased instruction subject to zero cycle execution at Rename) and the result is handled like the other cases we have described – remove the bias entry from the table, modify the relevant I-cache line, and flush the machine state).

How common is this case? It doesn't seem like it would be as likely as the first two, but maybe by making it cheap, this frees the Swift compiler to use `csel` rather than a conditional branch in many more situations, so that this will actually become a fairly common pattern?
In particular suppose the compiler did not want to use a `csel` because it knew a branch was biased (meaning the branch prediction machinery would usually predict correctly; in that case in the past use of a branch would have been optimal, but now `csel` is optimal). For example if this can be used in many error-handling or array-bounds-testing situations, it's yet another way that we can remove pressure from the Fetch and Branch Prediction machinery and effectively grow its tables because they're no longer handling these trivial branches.

(2023) trace cache

(2023) https://patents.google.com/patent/US20250021332A1 *Trace Cache Techniques Based on Biased Control Transfer Instructions* is a rich patent that's worth close study.

To understand the point, recall the basic problem of Fetch.
Our CPU is 10-wide. So we'd like to deliver an average of 10 instructions *per cycle, every cycle*, into Decode. We do this through multiple layers of sophistication.
The idea is to view the instruction stream not as sequence of instructions but as a sequence of branch points. We don't care about what happens between the branch points, it's obvious we just load those instructions; what we we need to do is predict the sequence of branch points and to load the instructions between them.

The simplest level of sophistication is a Fetch Predictor which takes in the "current" PC and spits out a "sequential run of instructions" which is the number of instructions up to the next branch, and the predicted target of that next branch. This pair (predicted target=next PC, num instructions to load) is sent to the I-cache, while the next PC (target of the branch that ends that sequential run) is also fed back into the Fetch Predictor for the next cycle's lookup.

The Fetch Predictor has to operate within a cycle which means it cannot be a very sophisticated branch predictor. Behind it we have multiple much more sophisticated branch predictors which take multiple cycles, and which are “validating” the sequence of predictions made by the Fetch Predictor. These essentially execute simultaneously with the time taken to access the I-cache.
If one of these sophisticated predictors doesn’t match the Fetch Predictor, we have to flush the appropriate instructions that have just arrived in the Instruction Queue (sitting before Decode). Of course each such flush loses us three or so cycles, not as bad as a flush of the out-of-order machinery, but still substantially cutting into our desire for an average of 10 instructions per cycle.

Note two things about this, some obvious some not.
- There are on average about one branch per five or six instructions. (Remember branch is not just a condition branch, it’s also eg goto’s, function calls, and returns.)
Since, by definition of how this Fetch Predictor works, we can only load one sequence of instructions, that between a branch and the next branch, we can only load on one average five to six instructions each cycle.
- We are only processing one branch per cycle. This means all the hard machinery of our branch prediction (training the predictors, and performing the sophisticated branch lookup based on long term branch or path history) only has to perform one lookup per cycle.

So clearly this first level branch predictor is maybe good enough for the 6-wide A7, but not for recent CPUs. What’s next?

Well (approximately) half of conditional branches are not taken. The Fetch scheme we described above stops a Fetch at the next branch, whether that branch is taken or not. (This is necessary if we can only train/predict one branch per cycle.)
Let’s drop that requirement. We’ll say that if we predict the next branch in our sequence of branches is not taken, then we’ll ignore it for the purposes of Fetch and Fetch past it to the second branch in the sequence. This gives us a linear sequence of instructions in the cache that’s on average maybe 9..10 instructions long, which gets us closer to our goal of feeding a 10-wide machine.
But the price we pay for this includes that we have to widen our branch prediction machinery to now handle *training* and *predicting* two instructions per cycle (the first of which we hope is a non-taken branch).

We can keep doing this, allowing our run of sequential instructions fetched to incorporate say two non-taken branches before the end branch (which is taken) but now we have to make our training and prediction machinery three-wide, and we don’t gain much advantage (there was a 50% chance that most conditional branches are not taken, but only 25% that two successive conditional branches are not taken).

So how can we do better?
- We can clean up various technicalities that slowed us down. For example if a sequential run ended with return, this used to mean a one cycle delay looking up the return address in the Return Address Stack before being able to feed that address into the Fetch Predictor for the next Fetch. So we can

(somehow) remove that delay.
- Some condition branches skip over just one or two instructions. Rather than ending Fetch at those forward branches, we can treat these *short forward branches* as non-taken branches for the purposes of fetch, and then remove the skipped over one or two instructions from the Instruction Queue.

Both of these strategies have already been described further up in this PDF.
But ultimately what we are running up against is that we can only access the I-cache once per cycle, so every point in our sequence of branches that corresponds to a substantial jump in instruction address (ie more than just a short forward branch) limits how many instructions we can fetch.

Well that sucks! What can we do?
The obvious alternative is to modify the Fetch Predictor to generate two records (a PC and a number of instructions to load) per cycle; and to make the I-cache capable of supporting two reads per cycle.
The second prediction is expected to be less accurate than the first, but that's life; it's good enough, and this has been tried.
But there's a smarter alternative! And this gets us to the trace cache.

The idea of the trace cache has been corrupted in most people's minds because of how it was used in the Pentium 4, where it was trying to solve a bunch of Intel-specific problem (variable length instructions, difficult to decode instructions, etc).
Forget all that, forget what x86 did. This is about how Apple uses the idea! And it's very smart indeed!

So let's consider different types of branches.
One set of branches are those which might call *high confidence*, branches where the predicted target is (almost) always correct.
A subset of high confidence branches is what we might call *stable* branches, which are branches where the target is (almost) always the same. (Note that a high confidence branch might always be correctly predicted, but the target might keep changing, for example the branch keeps alternating between taken and not taken.)
(One thing that's a little confusing is that this patent refers to *stable* branches, a set of other patents at much the same time call these *biased* branches. It seems like the bias terminology is winning out overall; but in these discussions I used both terms depending on the patent. It seems like Apple is still mining this idea so some of the patent ideas don't seem yet quite fully meshed with the ideas in other patents.)

We have already seen a few ways that stable/biased branches can be used.
Now consider a range of "stability".
- For example we can consider an "ultra-biased" branch to be the sort of branch we have already seen, one that the instant it changes direction we demote it from the category of biased branches. In a hand-waving way we can say it's 99% predictable, and because it's so predictable we're comfortable optimizing a lot of machinery around the assumption that it never changes.
- We can also consider "well-biased" branches as a new category, that are say 95% or more fixed in direction. We want to take advantage of this (generally) fixed direction but we will still retain such

branches in branch history and in the prediction machinery, so that the first stage or two of Fetch can make use of the generally fixed direction, while the 5% or so times that direction is incorrect can easily be fixed a cycle or two later by the slower, more accurate predictors.

Given these ideas, how can we use them?

We have already seen above how Apple uses the concept of a stable branch to optimize the branch prediction machinery. If a branch is considered stable, the branch is treated as a goto or a NOP, and if a goto is stored only in the Fetch Prediction machinery, without being stored in the expensive multi-cycle Branch Predictor machinery; and we don't bother to train on it or look it up in the multi-cycle Branch Predictors while the Fetch is being executed.
Since it does not count against our budget of two (or however many) branch training and lookups per cycle, we can also use *stable non-taken* branches (effectively NOPs) to make any particular Fetch Group a little longer (allow it to incorporate one or two *stable non-taken* branches along with the usual, less predictable non-taken branch) when possible. Of course we may be mistaken about a stable branch, it may actually at some point flip its target, but we'll catch that when the branch is executed and we treat it like any other misprediction. This is all about making the *Prediction* system more efficient (less energy, less area) by exploiting the fact that many branches are in fact stable.
That's all great, but can we use the concept of a stable branch even more?

Consider the set of *stable* branches that are *taken*. In particular consider a sequence of instructions that begins with jumping to some location, then we have four or five non branch instructions, then a *stable taken* branch, then another four or five instructions, another *stable taken* branch, another four or five instructions, then a non-stable branch. This sort of instruction sequence we will call a *trace*.

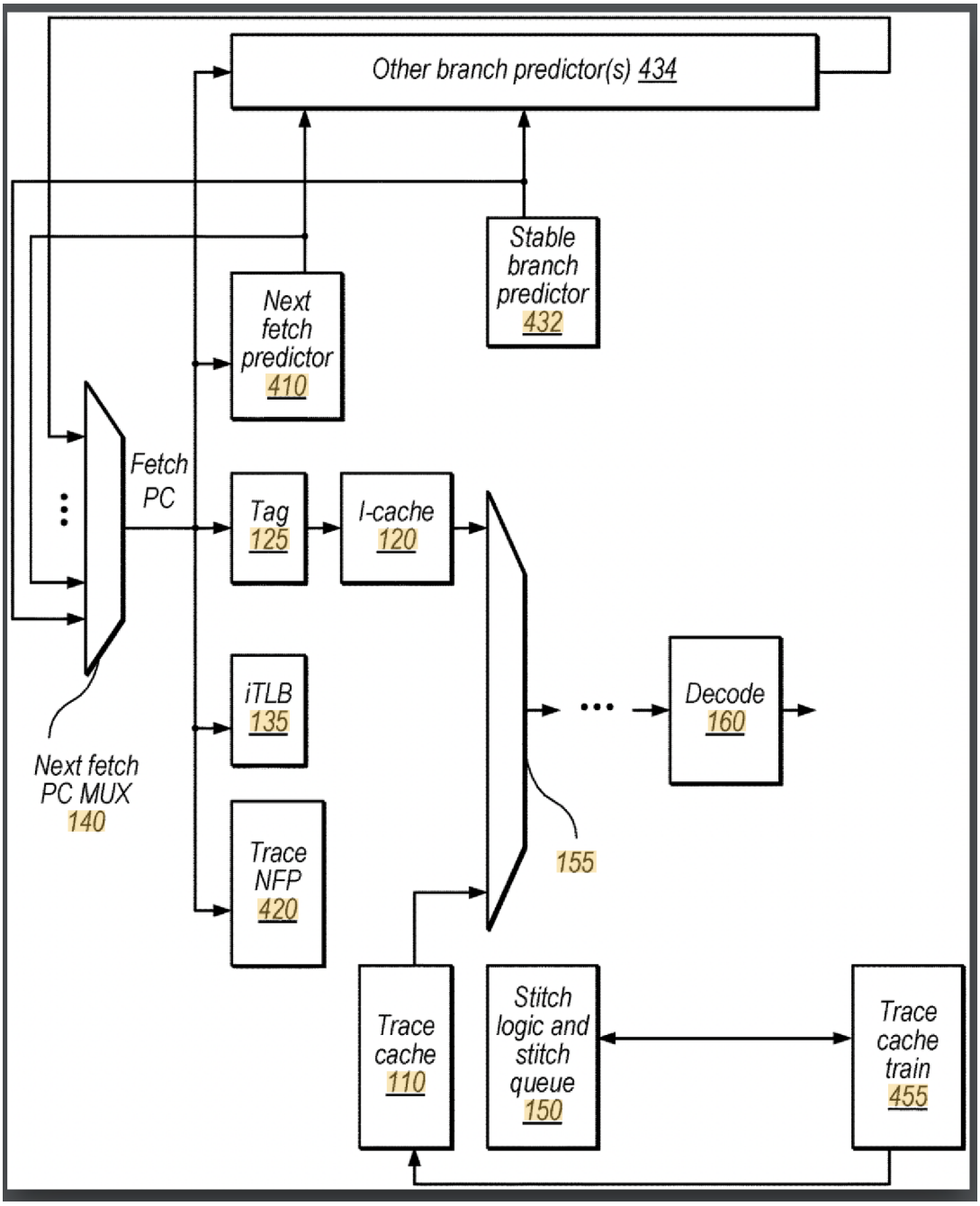
Note that, in terms of P*rediction*, there is no problem in executing this trace of 16 instructions or so long in one cycle, because, by definition, the stable branches don't exist as far as prediction is concerned, we just assume they will behave as they are supposed to and, if anything goes wrong, well we catch that at branch execution and we'll flush the ROB and restart.
No, the issue is with *Fetch*, that the trace of 16 instructions is split into three places in the I-cache, so we will require three cycles to access it.
This is the point at which we bring in the Trace Cache.

Suppose that these traces of ~16 instructions were stored in a separate cache, as a single linear sequence of instructions, indexed by the start PC of the cache. Then we could load the trace in a single cycle.
And now we hope that with all these various tricks (longer sequences of non-taken branches, skipping over short forward taken branches, traces of stable taken branches, etc) we're picking up enough Fetches that are say 12 or 16 cycles long to cover the cycles when we are not Fetching (front-end misprediction and resteer, miss in the L0 Fetch Predictor so we have to wait for the L1 Fetch Predictor, etc) so overall, hopefully, we hit our target of about 10 Fetched instructions per cycle.

The diagram below is fairly complex but shows the elements we've added to the system.
At the top we see the base system. The Next Fetch Predictor normally generates the Fetch PC, which

goes to the iTB and I-cache for instruction Fetch, and drives the Next Fetch Predictor for the next cycle. The predicted Fetch PC also goes to the “Other” branch predictors which will override it in two or three cycles if required. Along the way we also have block 432 tracking which branches are stable so that, once we learn a branch is stable, we can utilize it in various ways.

The instructions that are loaded from the cache going in the instruction queue 150 (which is already not exactly a queue since it occasionally “masks out” instructions that are skipped over by a short forward branch.


Other branch predictor(s) 434
Stable branch predictor 432
Next fetch predictor 410
Fetch PC
Tag 125
I-cache 120
iTLB 135
Next fetch PC MUX 140
Trace NFP 420
155
Decode 160
Trace cache 110
Stitch logic and stitch queue 150
Trace cache train 455

The new stuff that's added is
A trainer 455 that looks at the pattern of branches in the instruction sequence sitting in 150. When it sees a sequence that looks like a trace (so non-branches separated by one or two *stable taken* branches) it flags this sequence and the trace is copied from the instruction queue to the Trace Cache 110. (Of course a sequence of instructions separated by stable *non-taken* branches could also be a trace, but there's no point in wasting limited Trace Cache space on such a sequence; it can be fetched perfectly well from the normal I-cache.)

The one final element we add is we augment the standard Next Fetch Predictor with an additional similar table called the Trace Next Fetch Predictor, which knows which traces are sitting in the Trace Cache. So when a PC comes in, it goes to both the standard Next Fetch Predictor and the Trace Next Fetch Predictor. If it hits the Trace NFP, that prediction will win, and the lookup will go to the Trace Cache.
(The lookup may hit in both Fetch Predictors, because the sequence of interest will have been looked up multiple times in the Next Fetch Predictor before it is recognized as a trace and moved to the Trace Cache. For this reason, and general common sense, the entries of the Fetch and Trace Fetch Predictors, and the Trace Cache, have some sort of "usefulness" indicator, perhaps a decaying counter, perhaps a most recently used flag, perhaps a "used in the last 1000 cycles" bit that is periodically set to zero. It's expensive to detect and mark precisely when an entry in one of these three tables becomes not useful, but you hope that by using one of these "loose" usefulness indicators, fairly soon a table entry that's no longer being used will be replaced by a more useful entry.)

This all describes the basic system. Patents sometimes include, along with the main idea, an additional element which looks more complicated, but maybe can be added in the next revision, and we have such an idea here.
The the most obvious way to improve how effective this system is is to allow more branches to be part of a trace. This is the point of the "well biased" branches I described above.
Suppose we also allow traces to incorporate a branch that confidently predicted (eg by TAGE), but say 5% of the time does not go in its bias direction. What then will happen? 19 out of 20 times we get the long trace from the Trace Cache and life is good. The remaining 1 in 20 times, we still get the long cache but a cycle or two later the sophisticated (history based) predictor detects that this well-biased branch has gone in the wrong direction, and we resteer. We lose two or three cycles, but overall it's a win for the Trace Cache. Recall that only ultra-biased branches avoid the branch prediction machinery, well biased branches will be in the prediction machinery so there is an opportunity for this subsequent correction by TAGE.

So this is great; it allows us to get more value out of our Trace Cache.
There is, however, one small problem.
As part of the background operation of the prediction machinery, it's maintaining an on-going history register that tracks taken vs untaken branches, and this register is used to index into TAGE and similar

branch predictors. (There are actually multiple versions of these alive at any time, eg one that holds the most up to date [and therefore speculative] version of the branch history, others that are reliable up to a particular branch, so that if we have to flush and roll back to that branch we can recover an accurate version of the history at that point.)
When we had only taken branch (and perhaps one untaken branch) per Fetch, we could do the amount of work required to update this history register within one cycle.
Even adding ultra-biased branches doesn't hurt things because we've decided they don't add useful info to the on-going history path.
But if we have two or three well-biased branches (either taken or untaken) in a trace, now we have to add a whole lot of new information (maybe three or four bits to the end of the history register in a single cycle).
Doing this the way it was being done, adding one bit at a time based on working through the branches encountered in this Fetch group doesn't scale. To deal with this apart from the obvious fields that are stored in the Trace Next Fetch Predictor, one less obvious field is a short bit field holding the bit pattern corresponding to the relevant conditional branches stored in the trace. This so-called "pre-hashed history" token is the these bits shaped in an appropriate form to be added easily to the end of the history register.

Another complication we now have to deal with is how the rest of the branch prediction machinery (the stuff that kicks in after the initial fetch Predictor) handles traces.
The reason this is a little complicated is that these subsequent predictors are indexed by a hash of the Fetch PC and the history register.
So consider the first (faster) branch predictor. This is basically a gshare counting predictor, *except*
- the standard gshare predictor predicts the outcome of a single branch
- the Apple version predicts the outcome of all the branches in a Fetch Group.
So the Apple version is a set of bits corresponding to each position up to the maximum length of a fetch group; we match each bit against the fields of the Fetch Group that are conditional branches, and see if there is a mismatch. In non-Trace-Fetch all the internal branches in a Fetch Group need to be non-taken, so all the bits need to be zero (corresponding to non-taken), up to the last match, the last instruction in the Fetch Group, which is usually a conditional taken branch (and hopefully that matches the predictor), though occasionally for very long runs of instructions it could be a non-taken branch or a non-branch instruction.

Extending this to the Trace Cache is fairly simple. The theoretical problem is that the trace consists of instructions (including branches) that come from multiple contiguous runs of instructions. It's not absolutely obvious this same mechanism, using a single index, the Fetch PC corresponding to the address of the first instruction in the trace, will work well. But I guess simulations show that it works OK, well enough to do the job. Especially since, by definition, the internal branches of the trace are supposed to be especially stable. Only the last bit of this sequence of bits per instruction is expected to be an unstable branch for which the trace might occasionally mismatch the Branch Predictor.

The patent makes a big deal about this base predictor having to extend the number of bits by one, for reasons I don't fully understand. As far as I can tell the issue is that a Fetch group may have some maximum length of, say, 16 instructions, whereas the Trace Cache can provide a trace of say 17 instructions.
So maybe the Trace Cache contains up to 16 instructions in each "row" along with an *additional* auxiliary structure that holds the final (less predictable) branch that ends each trace and whose target governs the next Fetch PC? So this small change, adding one extra bit, had to be made to the base predictor? This doesn't seem crazy if we expect to (occasionally) be changing our prediction in the Trace cache for the target of that final branch, while we also expect to not have to modify the rest of the trace nearly as frequently?

The TAGE predictor operates a little differently. Suppose we have say an internal branches and the final branch ending a Fetch Group. Each of these branches might have a different optimal history length predictor and so might live in a different TAGE table. This means some sort of bitmask representing multiple branch outcomes is no longer feasible.
The optimal solution would index each of the branches by a hash of its PC and the exact branch history that led up to it. In practice what's done is that the tables are indexed according to the Fetch Group PC, not the branch PC. This may means in principle, that the same branch might be stored twice, once based on a Fetch Group that begins at address X, another time at a Fetch Group based on a jump into this Fetch Group say at address X+3(*4). On the other hand, these different paths to the same branch might also act as another bit of information distinguishing different histories to this branch, so while this takes a little more space, it also increase accuracy slightly. Overall, maybe it's a wash?

If the branch is indexed by the Fetch Group PC and there is more than one (eg an internal and a final branch) in the Fetch Group, how do we know to which of these branches a hit in a table refers? Apple says that along with the other fields associated with each entry (an address tag to handle aliasing, the prediction value, a useful value for training, etc) there is a position value which describes the position of the branch in the Fetch Group.
Which suggests that maybe the history used is slightly less accurate than I suggested, so a one-time hash of the Fetch PC and the branch history *up to the Fetch Group,* but absent information about what happens *inside* the Fetch Group? This would be slightly less accurate, but also slightly less energy because you only have to perform one lookup, see what hits you get, then sort them out by position; instead of multiple lookups with different branch histories.
However this does mean that we can index into all the tables simultaneously, we don't have to index repeatedly, first for the internal branch(es) then for the final branch.
(These details are interesting to muse over. You think you understand something like TAGE in theory, then as soon as you consider the practicalities of how to implement it in a wide (multiple branches per cycle) machine you realize just how many compromises are required, compared to the purity of the design in a paper!)

The patent also suggests a mechanism for indicating that a given entry in a TAGE table corresponds to

a Trace Group rather than a Fetch Group. (The simplest such is a single bitfield, but other mechanisms are possible, overriding other fields in various ways.)
Why is this necessary? The example they give is that if a prediction is supposed to match say a Trace Group but there is no branch at the appropriate position then we ignore the prediction (and presumably mark it as not useful).
I *think* ultimately what this is about is training the TAGE tables. The tables may start out populated by branches according to Fetch Groups. But once a given Trace is detected (with an internal taken branch, so that the PC changes substantially in the middle of the Trace) then any branch associated with that previous Fetch Group address becomes obsolete, and any branches that occur in the Trace (especially the end branch) associated after the jump in PC now are associated with a different Fetch Group address and need to be retrained.

What are the consequences of allowing say one well biased (ie less stable, only 95% stable) branch in the trace cache, rather than only ultra-biased branches?
Things get a little complicated because now that branch also has to be handed off to training, and it has to be predicted. (Because we need to catch a 5% mispredict rate in the front end, either via the base predictor or by TAGE, and handle it with a resteer; allowing 5% mispredict to proceed to OoO and requiring a flush when detected will be catastrophic.)
But maybe that's OK – for a trace we're now training and predicting two branches, the non-stable branch that ends the trace, and the less-stable branch embedded somewhere in the trace; and we have already said that our training/prediction machinery is scaled to handle two branches a cycle. And of course allowing this flexibility allows for more instruction sequences to meet the criteria for entering the Trace Cache.

The final piece of all this support for well-biased (~95%) branches is some machinery to count how often they mispredict and, if they mispredict too often, place them in a Bloom filter so that they'll be blacklisted and not reused (at least until the Bloom filter is reset).

So what does all this buy us?
The Trace Cache saves a little energy because we can load more instructions in one single activation of the SRAM. But of course it requires energy to run all the new Trace Detection hardware, the nTrace Next Fetch Predictor, etc; so at best we probably break even.
However we do have, on average, longer Fetches, as discussed in detail.
We *also* now have effectively a larger Fetch Predictor. Presumably we still have the old 1024 entry L0 Fetch Predictor and the 2048 entry L1 Fetch Predictor, but we also have a (?256 entry?) L0 Trace Fetch Predictor, *and* we need fewer entries in all these predictors because longer sequences of instructions are being associated with each entry, so our effective Fetch Predictor capacity should rise quite a bit.

The Trace cache means we (mostly) no longer need to distinguish between taken and untaken branches, we can say that we can "handle" (Fetch through, Train, and Predict) two branches a cycle, allowing for Fetch/Trace Groups of about ~12 in length. But it's actually slightly better than that. If we allow say a third of these to be ultra-biased branches then we get to about ~18 in length, even more so

if we allow more aggressive use of `CSEL` to replace certain branch patterns. Hopefully the bottom line is that wide Fetch is mostly a solved problem for now, and is not a limitation to widening the rest of the pipeline.

More sophisticated Fetch Predictor

Recall the basic idea of the Fetch engine. Every cycle we predict a Fetch Group (generally a run of instructions until the next taken branch), this being up to something like 16 instructions long. This has to be done with a Next Fetch Predictor that can generate a new prediction within a cycle every cycle. Such a predictor will not have perfect accuracy, so running "behind" the Next fetch Predictor we have a slower large and simple history-based branch predictor that takes two cycles and can (if necessary) correct a mistake made by the Next Fetch Predictor, wasting only a cycle (and a little bit of energy). Still further behind, taking anywhere from 3 to 5 cycles, we have the extremely sophisticated TAGE and Indirect TAGE predictors that can correct the two earlier predictors, at the cost of a few more cycles, but hopefully rarely, and before instructions enter the OoO machinery and correction is much more expensive.

With that background, we'd obviously like the Next Fetch Predictor to be as accurate as possible, but we also need it to complete within a single cycle.
The existing Apple designs have all been basically to repeat what was done last time this program counter address was encountered. This is a problem if we know that, for example, the branch that ends a Fetch group a branch alternates between taken and not taken. There've been some attempts to limit the damage this causes (for example a hysteresis that only changes a Next Fetch Predictor entry after two incorrect usages, rather than after each incorrect usage; that means an alternating pattern will be incorrect half the time which is better than being incorrect every cycle, which you'd get without hysteresis).
Putting all this together with what we saw previously of the Trace Cache, we're fairly well set for, on average, Fetching long enough runs of instructions, but it is still unfortunate to lose the occasional cycle (and some energy) when the Next Fetch Predictor makes an error and a subsequent cycle has to resteer Fetch.

Can we do better? What we would like is a way to incorporate at least some of the previous branch history into the Next Fetch Predictor, but in a way that's fast enough to be useful. A suggestion is given in (2023) https://patents.google.com/patent/US12353882B1 *Next fetch prediction using history*

The most obvious way to do this might be something like the gshare predictor of the early 1990s; where the index into your Next Fetch Predictor table is based on a hash of the PC and the branch history (ie a bit vector of taken/not taken branches). But what Apple do is a little more sophisticated.

The first element is that they incorporate at least two tables.
One is the current table, which is based purely on the previous Fetch Group's PC (ie previous PC X always leads to this subsequent particular Fetch Group). This not only matches a fair amount of real

code (after all the existing Next Fetch Predictor works reasonably well) but also provides a target for pre-constructed Fetch Group entries (remember that when lines are loaded into the L1I they are scanned for branches and, based on what's found, a best-guess new entry is added into the Next Fetch Predictor so that when code eventually targets that line it hopefully won't have to build the Next fetch entry from scratch wasting time and energy). Such a pre-constructed Fetch Group entry obviously has no idea what an appropriate branch history leading to it will look like!

The other table incorporates, in some fashion, a hash of PC and history. Again one could imagine gshare as an example, but what they describe is substantially more sophisticated, unlike anything I've ever seen elsewhere.
Instead of using a *branch history vector* (ie bit vector of taken vs not taken) they use a *branch path* (a hash constructed from some number of low order bits of previous branches). One could imagine at least two ways to construct such a path, a source path (based on the PCs of the branches that got us to this path), or a target path (based on the targets that got us up to this point). The Apple scheme uses both!
The index into the table is based on the source path, but the tag in the table is based on the target path. I guess this scheme is a way to use just a few bits (maybe two or three?) from each of the addresses in a path, but still have a pretty robust way of distinguishing different contexts?
Ultimately the goal is that we match in the history based table for variable Fetch Group targets, while matching in the base (history-free) table for non-variable Fetch Group targets.

One final tweak is consider Fetch Groups ending on a branch such that (depending on history) the final branch is either taken or not taken.
If we simply implemented things as described so far, this would mean that some number of entries in these tables would be storing a target address that is simply the next sequential address after the Fetch Group.
That's a waste of space!
So we actually split the above table into two tables, one of which holds branches that are taken, and which store a target; and one which holds branches that are not taken. The history context will decide which table gets hit (both tables will be searched in parallel based on the hash of current PC and source path; and a maximum of one will hit with appropriately matching tag).

One element they do not cover is how this interoperates with the L2 Next Fetch Predictor...

More sophisticated caches

We can simplify caches (eg L1 and L2) as being a collection of lines ("ways") associated with some address-based index. For convenience we'll assume the number of ways is 8 (as it is for L1D, though it's 12 for L1I, and varies for different models of L2).

One of the questions your cache has to face is, when a new line is accessed, which of the eight lines in the target index will be replaced? How to answer this has depended on how much logic/storage and

energy you are willing to throw at the problem.

The easiest solution (requires no storage) is choose a *random* one of the eight.

Next easiest is *non-MRU*; mark the most recently accessed line and randomly choose one of the seven lines that is not most recently accessed. This requires extra storage of either one bit per line or three bits associated with the set as a whole.

The natural choice everyone gravitates to after some thought is *LRU*, ie replace the line that was least recently used. That sounds good, but is not trivial to implement. You need to store an array that's more or less the equivalent of the order in which all of the 8 lines have been accessed, eg
MRU[7, 1, 3, 6, 2, 4, 5, 0]LRU
and each time a different line is accessed, you need to move that line's way number to the beginning of the array and move the others down.

All this is, of course, not even exactly what we want. What we really want is something like "of the eight lines, replace the one that will be accessed furthest in the future"; but of course that's not possible; we are using "last accessed a long time in the past" as a proxy for "will next be accessed a long time in the future". This works well under many conditions (probably because most apps have phases, and if the data has not been used for a while, perhaps it's no longer relevant to this phase and will not be used till the next phase). But of course it fails, and fails badly, in some circumstances like simple linear streaming, where MRU would be the optimal replacement [basically confine the linear stream to one way which is reused over and over, while the rest of the ways are not touched and are available for reuse after the streaming], and even random would be better than LRU.

So once we have hit the implementation level of genuine LRU replacement, are there even better alternatives? And the answer is "it depends, because what exactly is your cache doing?"...

One direction of improvement we've already discussed is perhaps you can detect a linear streaming pattern in incoming addresses, and so switch from LRU to MRU replacement.

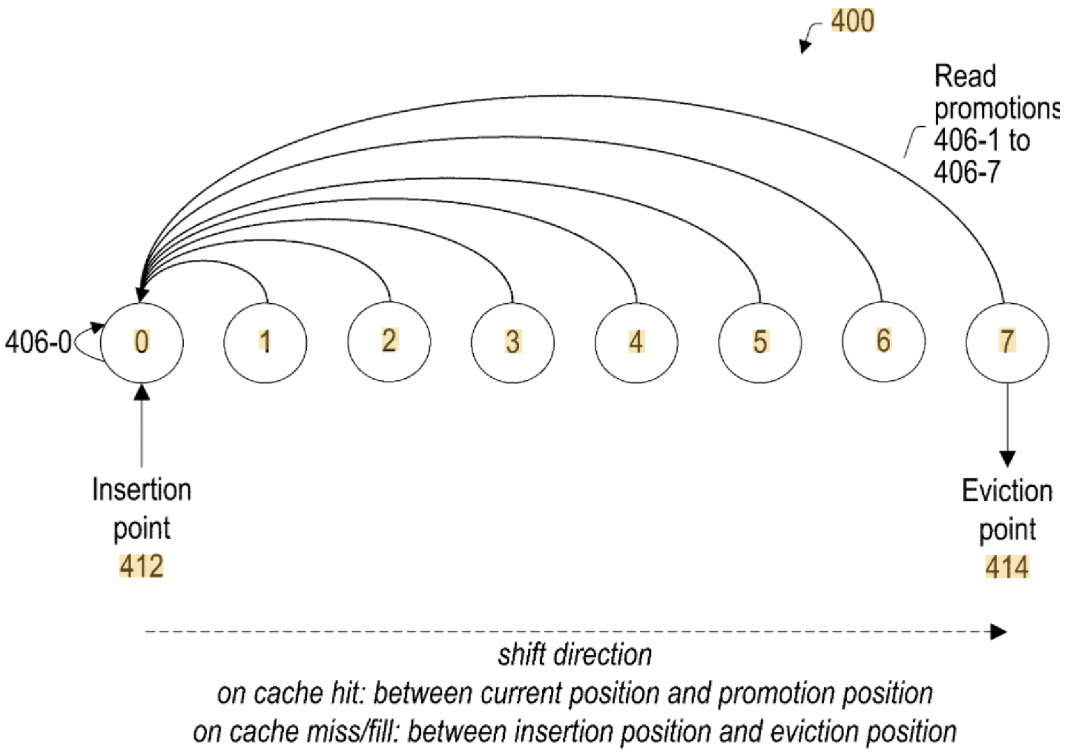
Another tactic is schemes that try to predict when a line is *dead* or the *re-reference interval*, ie how many cycles between bursts of access to a line. Both of these attempt to predict lines that are no longer worth keeping in the cache; if we can mark these as invalid then our replacement problem is simpler because there will often be an invalid line amongst our collection of eight lines, and clearly that's always the one to pick as replacement. These schemes actually work fairly well, but have a fair bit of overhead and are much more appropriate to an L2 or L3 than an L1.

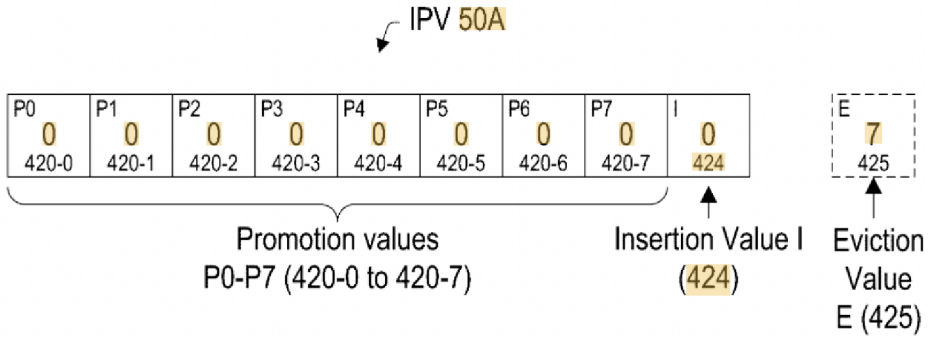
You can, however, see a different angle on the problem as soon as you remember that even the L1 cache, let alone the L2, has different types of lines in it.
- Some lines are prefetch lines (and haven't yet proved their worth);
- some lines are demand lines; and
- some lines are known-critical lines (either because of the line type, so an I-line, or a line from a page table; or because the CPU indicated the load was critical, which can be approximated by things like knowing up how many instructions are backed up in various queues).

So (to simplify things; we could imagine more options) we have at least three categories of line in either our L1 or L2. Should these categories inform our replacement algorithm? If so, how? Obviously, in some sense, we want critical lines to be “stickier”, demand lines to be intermediate, and prefetch lines to be only slightly sticky -- give them a chance to prove their value, but stand willing to assume that the prefetch was a mistake and the line will never be accessed.

If you want to do this sort of thing, simply maintaining an ordered list from MRU to LRU no longer does the job. What might be a feasible alternative? It’s worth thinking a little about this before going further...
This is, ultimately, the question answered by (2023) https://patents.google.com/patent/US12222875B1 *Insertion/promotion vectors to update replacement data in caches based on criticality*

To understand their solution, look at the diagram below, which depicts two ways to imagine the LRU algorithm.


400
Read promotions 406-1 to 406-7
406-0
0
1
2
3
4
5
6
7
Insertion point 412
Eviction point 414
shift direction
on cache hit: between current position and promotion position
on cache miss/fill: between insertion position and eviction position


IPV 50A
P0 0 420-0
P1 0 420-1
P2 0 420-2
P3 0 420-3
P4 0 420-4
P5 0 420-5
P6 0 420-6
P7 0 420-7
I 0 424
E 7 425
Promotion values P0-P7 (420-0 to 420-7)
Insertion Value I (424)
Eviction Value E (425)

Consider first the diagram. The 8 circles, [0...7] represent the 8 slots in an array showing the order of last use of the eight cache lines. If we call those lines A, B, C..H, then for example slot 0 might hold D as the most recently used line, and slot 7 might hold A as the least recently used line. More generically think of the ordering not as “recency of use” but as “desirability” from most to least desirable.
So two types of things may happen.
- An access may require a new line. In that case we look at the least desirable slot, slot 7, see that line A is considered least desirable, and replace line A. This is what is referred to by *Eviction point 414.*
- An access may hit an existing line, eg line F. We use some lookup (probably a few extras added to the tag of line F) to see that this corresponds to slot 5 in our array. The Finite State Diagram then says that a hit in slot 5 moves the contents of slot 5 to slot 0 (ie line F becomes the new Most Recently Used line), AND we move all the entries between slot 0 and slot 5 right by one entry, so the value of slot 0 moves to slot 1, ... the value of slot 4 moves to slot 5.
You can see that this corresponds to the algorithm for what it means to maintain an LRU cache.

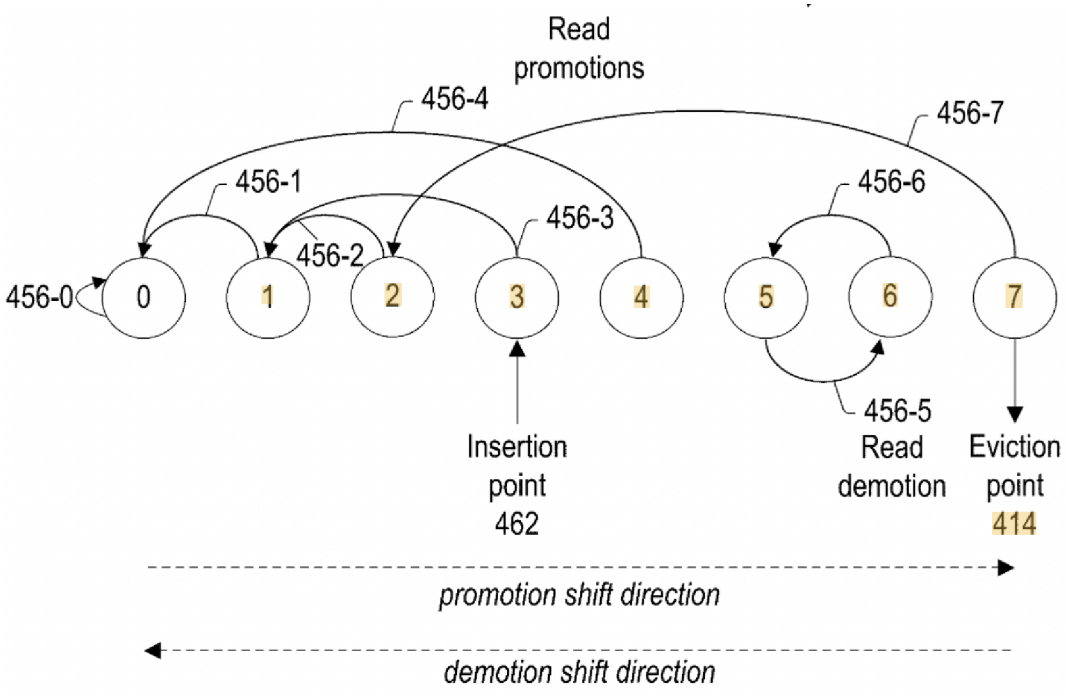
The second diagram shows how to represent this in a few bits of storage. The arrays that’s drawn is NOT the array of eight slots holding [F, A, C, ...]; it is a read-only array that describes the LRU algorithm. Each of the boxes labelled P0..P7 correspond to the slots in the Finite State Diagram. The entry in each box gives the slot to which we move a line when we access that line and it is in a particular slot.
All the values are 0 because whenever we hit a line, that line becomes MRU, so it moves up to slot 0. It’s implicit that we move all the elements between the target slot and the source slot right by one entry.

Additionally we have a ninth element, the I value. This says in which slot we insert a replacement entry. In the LRU algorithm, when I access a new line that wasn’t in the cache, I have *accessed* the line, so after it replaces the previous LRU line, this newly loaded line becomes marked as MRU, ie it goes into slot 0.

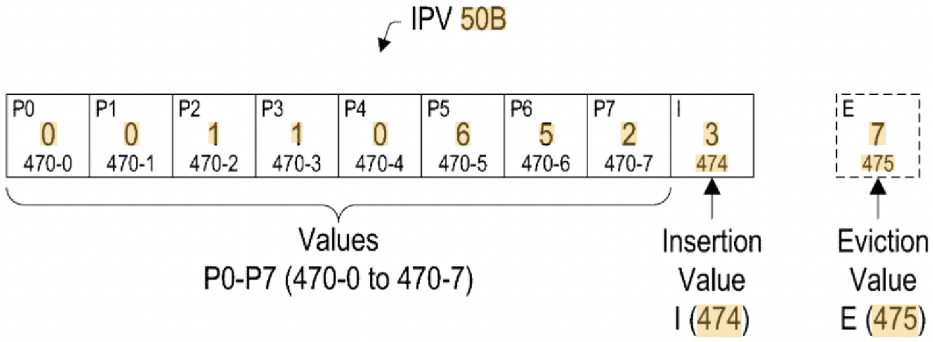
Finally we have the tenth E entry. The patent suggests that you could have a tenth entry describing the slot from which you’d evict, but the only case that realistically makes sense is having that always be 7, ie the least desirable of the ordering of line desirability. (You could in theory make it 0 if you are trying to confine linear streaming to a a single way; or you could even maybe flag it with a special value to indicate a random choice. It seems like Apple has not considered these options as of right now, but maybe in future.)

This short array describing a replacement algorithm is referred to as an IPV (Insert/Promotion Vector).

So this was a way to describe LRU. Now let’s see how this machinery can describe more interesting situations. The diagram below is not meant to show a particular algorithm, rather it’s meant to show a set of different possibilities from which, appropriately chosen, one could construct a sensible algorithm.



on cache hit: between current position and promotion/demotion position
on cache miss/fill: between insertion position and eviction position



So to put it into words, what we're saying is:
A hit of the most desirable line [0] keeps it as most desirable.
A hit of the second most desirable line [1] moves it to most desirable [0], *and* (implicitly) moves the [0] line rightward into [1].
Likewise a hit to slot [4] moves it to slot [0], with [0], [1], [2], 3 all moving rightward.
So far this is LRU.

But a hit to [2] does not move it to MRU (ie most desirable), it just moves it to slightly more desirable, to position [1]. Likewise a hit to [3] moves it to slot [1].
We could see these two transitions as elements of a scheme where a line has to prove itself via multiple recent accesses to become more desirable. (Remember even if you get one access to move from [3] to [1], if some other line gets accessed before your second access as line [1] you will be right shifted downward to [2].)

Alternatively we could have a transition like [7] to [2] where now we are saying that if an undesirable line gets accessed, it gets promoted a fair bit, but not all the way to [0] and not even all the way to [1].

Finally we can have weird transitions that go backwards like the 5/6 thing. I think this case is supremely unrealistic and locks the two lines in place with no way to ever remove them, but it's meant to show the kind of flexibility possible, that with this machinery you can either
- lock lines in place or
- demote lines to *less desirable* after they are accessed (which might be useful for the pathological case I keep referring to, namely the linear streaming case)

Then, as previously, the second diagram shows how we represent this in a short read-only memory. You can work through it to see how it corresponds to each transition. We also show that (in this crazy alternative to the LRU algorithm) we could insert a new replacement line not at the MRU slot but at, say slot [3].

The final element that turns this all into something useful is that the CPU provides a few different versions of these IPVs.
So, looking below, we have a cache (in theory it could L1I or L1D, but realistically it's probably L2, and the L1's use an approximation to LRU).
We have the cache data and cache tags (both familiar), *Replacement Data 32* which holds the order of lines , eg [F, A, B, H, ...], from most to least desirable, *Control Circuit 30* which moves the items in Repl Data 32 every time they need to be reordered (so on a cache access or insertion of a new line), and multiple IPVs, one of which determines how that reordering of the Repl Data is performed.

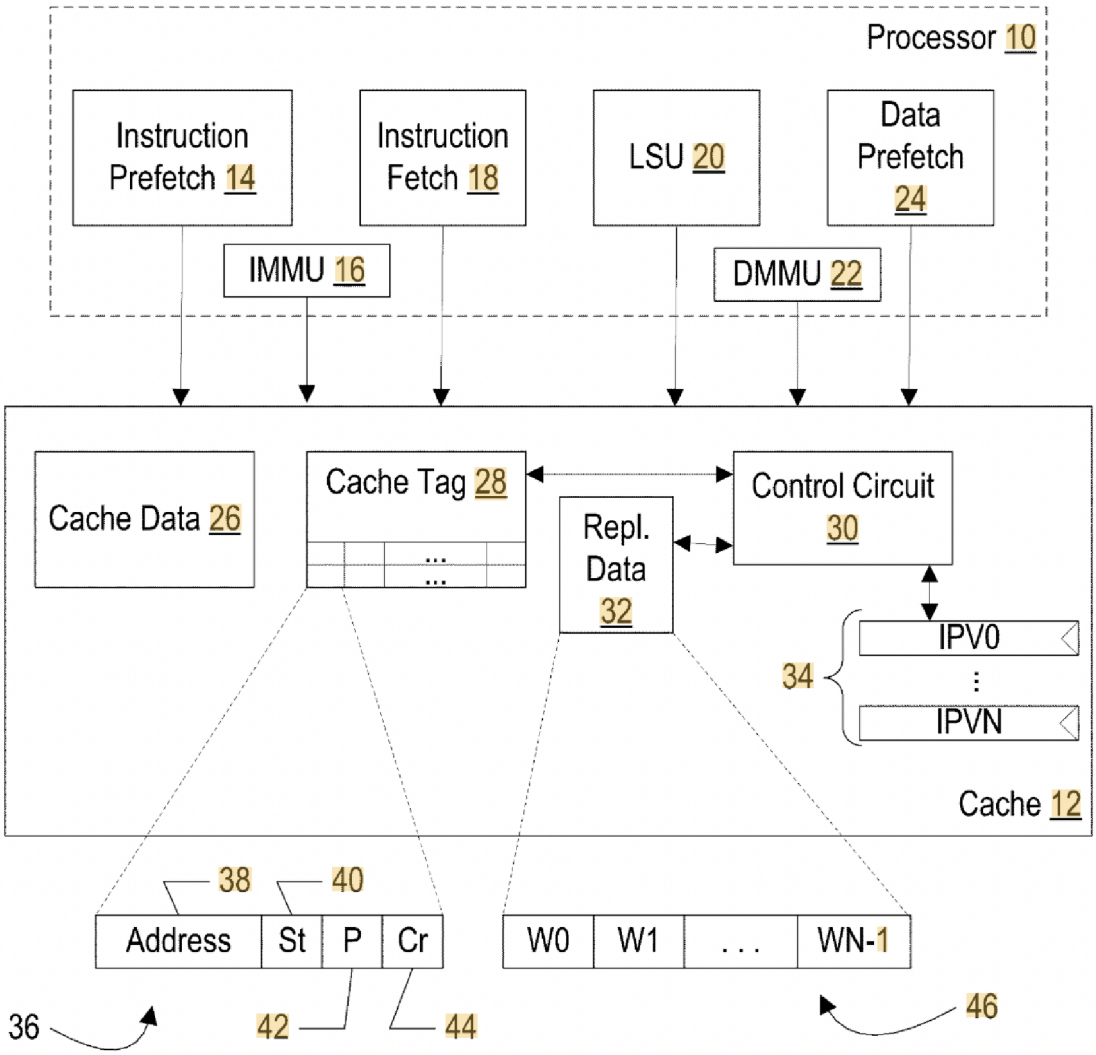

Processor 10
Instruction Prefetch 14
Instruction Fetch 18
LSU 20
Data Prefetch 24
IMMU 16
DMMU 22
Cache Data 26
Cache Tag 28
Repl. Data 32
Control Circuit 30
IPV0
IPVN
34
Cache 12
38
40
Address
St
P
Cr
W0
W1
. . .
WN-1
36
42
44
46

FIG. 1

The trick is that which IPV is selected is based on properties of the cache line. So a cache line may be marked as Prefetch'ed (the P bit 42) and/or as Critical (CR bit 44). These bits are also known when a new line is inserted into the cache, so they can also be used to select the appropriate IPV when inserting a line.

We can now imagine a few different scenarios. For example we may mark Critical lines as following something like LRU; a critical line is inserted as most desirable, and is easily moved back to most desirable. A Prefetch non-Critical line may be marked to Insert at say slot [3] rather than slot [0] so that it has to prove itself valuable fairly rapidly; if it doesn't prove itself valuable by being accessed within 4 rounds of demotion (right shifted) it was probably a mistaken prefetch and should be tossed rather than wasting valuable cache space for more cycles. Once a P line is accessed on demand, the P bit is cancelled, and from then on it is treated like normal line (which may or may not have the Cr bit set). Finally a normal line could a promotion vector that looks something like the gradual promotion we saw in our second finite state machine; eg if it's in the lower half of the vector it gets promoted to slot [3], and from slot [3] each time it's accessed it moves up one slot. This will allow it to move all the way to slot [0] if genuinely useful, but will make it rather less sticky than a Cr line.

This machinery is, unfortunately, all Apple tell us. They don't give us any examples of the entries in particular IPVs. But they do give an interesting answer to the question: how should you populate the IPVs? Their answer is basically "optimize the IPVs to maximize performance on some representative benchmark suite". The exact optimization scheme they suggest is to use a genetic algorithm to modify the set of IPVs from successive run to successive run, but obviously alternative optimization schemes could be used if someone thinks up something better.
Presumably the optimal IPVs from this optimization process, having been calculated in Cupertino, are then burned into the CPU.

You could imagine small tweaks on this process. For example a few alternative ways to categorize cache lines include MESI state (and specifically Modified vs not), or stack vs heap, or owned by the process vs owned by the OS or shared across multiple processes.
And in principle using different IPVs for each different type of line might improve outcomes...

connection to existing ideas

The state of the art in other L1 cache HW appears to the (static) RRIP algorithm, as described here:
https://en.wikipedia.org/wiki/Cache_replacement_policies#Static_RRIP_(SRRIP)
RRIP is (almost always) an improvement on LRU in that it behaves much like LRU but by treating a line differently the first time it is inserted into the cache it is also capable of handling streaming situations (where there is no data reuse after the first access, and we don't want the streamed data to wipe out the entire cache, only one way of each set). RRIP is present in ARM's 2025 high end cores, the C1-Lumex cores.

Apple's scheme seems inspired by RRIP ideas, but extended to also handle

- prefetched data (which may not be useful). RRIP could be extended to this in an obvious fashion, and may well be in ARM's core
- critical data. This is a substantial Apple difference that no-one else so far seems to be tracking.

Prefetcher deny list

As soon as you hear the title you can make a guess at what (2023) https://patents.google.com/patent/US12306762B1 *Deny list for a memory prefetcher circuit* is about! And you'd be essentially correct.

Obviously prefetching is a balancing act between pulling in too little (limiting performance) and too much (always wasting energy, and sometimes even reducing performance by replacing useful cache data with useless). One way we can perhaps improve things is to track lines that are prefetched because of the designs of our prefetchers, but which are not actually useful.

Implementing this idea requires a few elements.
- First is we need a bit in the cache (certainly L1, but it's helpful to have it also in L2) indicating that a line was brought in by a prefetcher.
- Next we need to clear that bit when the line is accessed for "valid" reasons, ie when it's actually used.
- If a line is replaced with the prefetch bit still active, we know that it's not a useful line, so we need to memorize it in some structure.
- And when prefetching we need to test each next candidate prefetch address against this structure.

You could imagine a few ways to implement this, for example a Bloom filter seems an obvious choice. But what Apple do is more like a standard lookup table – some sort of index hash (based on a few address bits) that looks up an entry with a tag of a few other address bits. This sort of scheme gives us a high probability (though not perfect) of hitting the entry we want without aliasing, while reducing the number of bits required to store the blacklisted address.
So an entry is very small: a few bits of tag (which, along with the implicit index hash together approximate the address), a valid bit, and a count. The count is presumably why a table was chosen rather than a Bloom filter. The count allows a line a few chances, maybe four, to fail being used before it's concluded that the line is truly useless. If the line *is* used before the count hits maximum, the entry is marked invalid and life goes on as normal.

The one additional thing this scheme does, and which adds some interesting value, but which is not completely obvious is it allows for some degree of prefetch rationing.
Examples of this include
- a prefetch address that matches in the Deny List, but is not yet at full count might always be moved to the end of the list of lines being prefetched. That way if latency is being rationed, the lines given highest latency (least likely to arrive at the cache in time) are probably those that aren't going to be used anyway.
- suppose we have a different situation of bandwidth rationing. For example when the CPU wakes up after being asleep (and all the caches are empty) the prefetchers are run more aggressively than usual to fill the caches ASAP. But these means that NoC bandwidth, RAM bandwidth, even the number of slots holding prefetch addresses queued up for submission, could all be flooded. An obvious solution is

simply to drop these (probably not going to be used) prefetch addresses, until bandwidth is no longer a critical factor.

Hierarchical store queue

As we know, in an aggressive OoO machine, a lot of state has to be held in a “temporary” condition until all speculative instructions ahead of it have cleared.
One part of this is stores, which have to be held in some sort of buffer until it’s safe (ie no longer speculative) to transfer the store to L1 cache. This means that the store queue is one more item that we would like to make as large as possible (to allow for more OoO), but making any structure larger has downsides. So, as always, we look at whether we can increase the effective size with smarts rather than blind upscaling.
How do we do that? By analyzing all the tasks a structure performs, and asking if they can be disaggregated.

The store queue performs three primary tasks
- storage of speculative writes
- searching for loads (of earlier, but not yet committed, stores) in the store queue
- moving stores to the L1 cache.
The second task, in particular, adds expensive circuitry.

So what Apple proposes is two separate store queues.
The first is like what we have today, and stores the oldest writes.
The second
- stores newer writes
- has no ability to interface with the L1 cache, only to move stores to the primary store queue
- still handles searching for loads, but with slower, simpler circuitry.

The assumptions are that
- much of the time only the first queue will be used
- the loads that are most important in terms of being serviced rapidly tend to be those that are oldest and access (and are serviced from) the primary store queue
- when the second queue is used, it’s frequently a situation where we’re going to land up blocked anyway once all the OoO resources fill up, so there’s no real home if it takes a few cycles more before we grind to halt anyway waiting for some DRAM access to be serviced.
Presumably simulations show that, with an appropriately sized first queue, these assumptions generally hold.

One way search of the second queue is simplified, with lower area, is that only the first queue supports full three-wide load searching; the second queue provides for only a single load search at a time.
Another difference is that the fast queue provides for multiple means of searching for addresses matching a load, including via the equivalent of an “index” [eg a table of address hashes], so that a hit against a store occurs in around 4 cycles. The slower queue provides none of this, just a fairly mechanical

comparison of addresses one block at a time, so that a hit against a store occurs in around 11 cycles.

There are a number of questions you might ask about the details of this system. Presumably both queues maintain time stamp data that indicates their beginning and ending ranges (probably in terms of instruction number). This would allow quick filtering to see whether a load should be routed to the fast, slow, or, worst case, both queues. (Load doesn't care about any stores that happened after its particular instruction number...) The patent provides a few technical details, but not this particular element.

It does say that the fast queue consists of two SRAM banks, while the slow queue is a single SRAM bank. I'm not sure quite how two banks comports with having three load pipes route to the fast queue. It may be that the (generally justified) assumption is that at least one of the loads usually falls outside the time range covered by the primary queue, so these stores are not relevant to the load and it can immediately be routed directly to the L1 cache without further store queue probing?

Another element that may make all this work better than expected is that a number of the most critical loads (recent store followed by a load) may now be captured by the various ZCL mechanisms that handle the load via register renaming, or load value prediction. Even when these require confirmation by subsequent execution of the load, most of the time it's no longer critical that the confirming load have minimal latency.

Another detail covered is that, to the maximum extent possible, "split" loads are handled from the store queue. For example
- suppose part of a load is present in the store queue but not the rest (eg store was 16b, load is 32b). This is handled by extracting as much as possible of the data from the store queue and only the remainder from the cache, rather than waiting for the store to move all the way to cache and then loading from the cache.
- likewise assume part of a load sits in the primary queue, part in the secondary queue. Again the load is assembled from the two pieces, rather than waiting for a simpler option like when the secondary queue entry moves to the primary queue.

This is all covered in (2023) https://patents.google.com/patent/US12298915B1 *Hierarchical store queue circuit.*

How aggressive is mispredict machine flushing?

Suppose we have a particular instruction, let's say a branch, that mispredicts. We first detect the misprediction at the execution of the branch. How then do we handle it?

The (relatively...) easiest solution is to tag the instruction in the ROB, and wait till this instruction reaches the head of the ROB, at which point e flush the entire machine. What makes this simple is that because the instruction has reached the head of the ROB we know there are no live instructions inside the machine still queued up or executing that conceptually come before the misprediction.

So flushing the entire is correct, and our main difficult job is ensuring we have a way to restore the register file to what it should be at the point of the mispredicted instruction.
The downside to this, of course, is that we may waste a lot of cycles waiting until that mispredicted instruction reaches the head of the ROB.

We can make this slightly more performant and energy efficient with a few simple tweaks.
One is that we can *immediately* flush the *in-order* part of the machine (instruction fetch up to Resource Allocation) at the point where we discover the misprediction. By definition of being in-order, these instructions must be after our mispredicting instruction, so they will be flushed anyway at some point. Secondly we can start redirecting Fetch to the correct address and can start to fill up the machine all the way to Resource Allocation, so that once we have flushed the out of order part of the machine we can immediately start Dispatching instructions from Allocation.

This is the standard model of handling mispredictions, and seems to be implicit in every academic paper on the subject. It's one of the reasons we assume that mispredictions are so expensive, because we have this (possibly long) wait until the mispredicting instruction reaches the head of the ROB.

But can we do better?
We can in fact imagine two levels of better.

The first level waits, after we discover the misprediction, until all *currently executing* instructions have completed. So basically we put a block at Issue for a few cycles. While that's happening we can do the same stuff as the two previous optimizations – flush the in-order part of the machine and restart Fetch.

Once the execution units have drained, then we want to essentially flush all the instructions in the various scheduling queue and buffers that are *later* than the mispredicted instruction, but not those that are earlier. The main thing this requires that's new is some sort of "global timestamp" attached to every instruction in every issue queue that gives the in-order placement of the instruction, so that we can find all the later instructions and invalidate them and only them.

Even more sophisticated is we either cancel, or allow to complete, instructions that were in progress when we detected the misprediction. Imagine, for example, that at the point where we detect a branch misprediction we also have a earlier division that's executing, and a later add that's executing. We want to preserve the result of the division, while also canceling the add.
This requires everything the previous scheme requires, along with also passing the global timestamp down every execution unit, along with a broadcast some sort of "flush" signal and a way to compare the broadcasted timestamp to the execution unit timestamp.

All this background gets us to (2018) https://patents.google.com/patent/US11422821B1 *Age tracking for independent pipelines*, which is obviously a fairly old'ish pipeline but I didn't notice it until recently. What it describes is a way to track these various timestamps so as to achieve the goals of this most aggressive form of machine flush (which is obviously most performant, in that once you detect a misprediction you can immediately start recovery, while machine execution continues with previously enqueued instructions that are prior to the mispredicted instruction).

The details are very finicky (and have almost certainly changed at some point), but the big idea at the time was to only allow one instruction issue per cycle that could possibly mispredict. Then, relative to that one instruction, earlier instructions that issue in the same cycle get marked in a bit vector as, say, 0; while later instructions that issue in the same cycle get marked in a bit vector as 1. Then if we do have a misprediction, by looking at the bit vector we know which execution units we need to flush. (There are obviously extra complications if the execution unit is multi-cycle, and obviously the scheme needs to be generalized if two or more pipelines are now allowed to mispredict in the same cycle [which is probably the case!]) But what this scheme does is allow us to track the relative temporal ordering into the execution units without the area and energy costs of actually having to tag each instruction passing down the execution unit with a timestamp.

More fusions!

It's always nice to see more fusions added to the system.
(2023) https://patents.google.com/patent/US12217060B1 *Instruction fusion* describes the latest crop. All of these are, in a sense, obvious, and I guess it's just been a question of Apple working their way down a big list ordered by frequency of use, adding new pairs every time there's some manpower available to work on the next element on the list.

The pairs added are
- integer divide followed by multiply-subtract. You can, if you prefer, call this "idiom recognition" but the goal is a pair of instructions that between them return the remainder of a integer divide. This can also be modified to return a quotient and remainder, presumably by the same mechanism that allocates and uses two destination registers for a load pair.

- an ALU operation followed by a mask (ie `AND` with a particular type of constant).
This seems like fusions we already have (ALU followed by simple logic). I think the new feature here is that a little bit of energy is saved by handling special cases like masking down to an 8bit char or 16bit short, and optimizing the data movement in that case. Also the `AND` can be handled as a simple bit extraction rather than even executing the actual `AND` operation.

- a compare followed by CSEL. The most general version of this could be something like comparing rA vs rB, and selecting one of rC vs rD. That's too many input registers to a single instruction! You could imagine a pruned down version that allows for three inputs, but that's an uncommon pattern. What Apple fuses is the common pattern where we compare rA to rB followed by a `CSEL`. This gives you `MAX` or `MIN`, and variants like pinning an index between two array bounds.
`CSEL` comes in a number of variants that can do things like negate the selected value or `not` its bits. The patent does not seem to allow for this full richness, only mentioning that the increment (only by 1, not by a short immediate or by a register) version is also supported. Presumably the other cases are less common (though I would have though the variant that allows for absolute value is common

enough). Maybe in later models?
And remember what we saw above, that the new Branch Prediction scheme also allows for predicting and short circuiting the outcome of "close to certain" `CSEL` operations. Unclear how those are handled, but presumably they are tagged as a slightly different sort of `CSEL` operation (like the branches are tagged as slightly different branches) and are not fused? Instead the `CSEL` is presumably executed as the "main" operation (a MOV, an Increment, or whatever) according to the (strongly biased) predicted branch, and the compare is left unfused to go through execution as maybe a kind of fake branch and (hopefully confirming) the predicted direction?

- construction of large immediates (ie base instruction defines the lowest bits, next instruction OR's in some higher-order bits). The fused pair acts like a `MOVI` with a larger constant, and is, like a normal `MOVI`, handled at Rename rather than at Execution.

Mislabeled (IMHO) fusions!

(2023) https://patents.google.com/patent/US12288066B1 *Operation fusion for instructions bridging execution unit types* is a very interesting patent. With an utterly terrible title!

Consider an instruction like `SCVTF` (Signed Convert to Floating-point) or `DUP` (duplicate int register across all lanes of a NEON register). Both of these seem simple enough but Dougall's M1 latency tables give a latency of around 12..13 cycles. Why?
The underlying difficulty in both cases is that the instruction requires reading an integer register then placing the result in an FP/NEON register. But the INT and FP registers are more or less two isolated worlds! How do we move items between them?
The most practical bridge is via the Load/Store unit, which has to be able to take in integer registers (eg as addresses) and write our FP/NEON registers (eg from a load).
So the way this has been handled by Apple for a long time is that an instruction like `SCVTF` is cracked into two instructions, the first of which is a fake load that moves the data through the load store unit to write it into an FP register, after which the second instruction reads the (integer) bit pattern from that register, converts it to the FP equivalent bit pattern, and writes it back to an FP register. Moreover, because this was considered a low-priority path, the minimal changes were made to the LSU, meaning that the fake load basically runs down the full load pipeline doing things like TLB lookup and checking for matches in the store queue, even though none of this is necessary.

So at some point, I guess in 2021 or whatever, someone at Apple decided enough with this nonsense, there were now enough transistors available to do the job properly. Which means the way an instruction like this is now handled is
1. there is a separate path in the LSU dedicated to Register Transfer, independent of the main LSU Memory path.
2. that would solve the immediate problem of expensive movement from the INT to the FP side. We could stop there and leave the design as above, with two cracked instructions. But Apple do a little better than that.
3. The HW that performs the conversion to an FP value, or that duplicates a value across lanes, is

moved from the FP execution unit into this Register Transfer path. Which means that a single instruction can now perform both tasks, first accept the integer value, next reformat it as appropriate to a 64b FP value or a 128b NEON value or whatever, and finally write it out to the FP/NEON register file.
4. It's still helpful, apparently, to crack the instruction at Decode time. My guess is that this helps for Rename. One instruction handles the Rename side of the integer register (looking up in Integer History File what's the appropriate physical register source) while the second instruction handles the allocation of an FP/NEON destination register.
5. But it's a shame to keep those two separate instructions since that wastes later resources, so the instruction flow is essentially
- a single instruction comes into Decode, and (now) a single ROB entry is created
- two instructions flow from Decode to Rename
- after allocation, those two instructions are re-fused to a single instruction
- which is sent to to the LSU which takes an integer register input, does the conversion/reformatting work, and writes the result to an FP/NEON destination register
So latency of these instructions should drop a lot, I'm guessing to maybe 5 cycles or so.

With all this history, you can kinda understand the name of the patent, but it doesn't reveal the most interesting aspects of the patent.

There's another interesting aspect to this.
The LSU is becoming a crowded place, and it's harder to add load or store units than it is to add ALU units. One problem is that, for two different reasons, we have both these types of instructions (Int↔FP transfer) and AMX/SME instructions also flowing down the LSU pipeline. Most AMX instructions are not "genuine" load/store and so also don't need to interact with machinery like the TLB and the store queue.
So a logical next step might be
1. to move from "Memory" path and "Register Transfer" path to "Memory" path and "Auxiliary path" where Auxiliary path handles both register transfer and moving AMX instructions around the L1 cache out to the AMX unit.
2. add a fifth port (ie scheduling queue) to the LSU handling only these "auxiliary" instructions.
Between them, this would limit the extent to which these "non-LSU instructions" are eating away at our precious and so so limited genuine LSU bandwidth.

Faster processing of barrier instructions

Think about processing an ISB (Instruction Synchronization Barrier) instruction. At the highest level, this essentially boils down to splitting instruction execution into two halves – one set of instructions executes up to the ISB, the state of the machine is recorded (ie all stores are persisted to cache, all changes to system registers, like starting SME or changing the security level are persisted) then we get the ISB, then execution of the instructions after the ISB starts using the new state of the machine.
This looks very much like the execution of the machine up to a mispredicted instruction! Once again we have execution up to the mispredicted instruction, then something that forces persistence of the old

state, and we start new instructions with new state after the mispredicted branch (or whatever).

This suggests that (with a few additional tweaks) we can use the same sort of ideas as the previous patent to accelerate the handling of ISB. (2022) https://patents.google.com/patent/US12229561B1 *Processing of data synchronization barrier instructions* works through the details.

The big picture is that the easy way to execute an ISB consists of
- start execution of ISB
- wait for the ISB to reach head of the ROB (which is equivalent to waiting for all prior instructions to complete)
- flush the pipeline and refetch instructions
- ISB is over and we can start executing the new instructions

As with misprediction, this does the job but requires long waits and excess power. How can we do better?

The first obvious step is, at the point where we *Decode* the ISB we immediately flush everything behind the ISB in the pipeline and halt Fetch. This way we won't waste power processing instructions to be flushed later. We pause Fetch until ISB 'execution" completes.

That handles the most important element of instructions *later than* the ISB. (Most importantly that they are fetched in the "correct context" of the ISB, ie after the Exception state has changed to Supervisor level, or whatever).
What about instructions earlier than the ISB? The point of waiting till the ISB hit the head of the ROB is to ensure that every such instruction has completed its context change (if any) before the new instructions are loaded into the machine. But we don't have to wait for the ISB to reach the head of the ROB, all we need is to ensure that there's no important context change pending before we allow new instructions to enter the machine...
One can imagine a few ways of doing this. The first method that springs to mind is marking some context-creation instructions (eg write special register) as significant, and somehow blocking the ISB until all such significant instructions are completed. This would work, but it's actually heavier than what's needed.
A lighter touch is achieved by doing things on both sides, by both marking that an earlier instruction has the potential to create special context, up until the instruction completes [at which point the context is by definition active] and detecting all the instructions that *consume* special context.
If consuming context happens under these conditions, then we treat the consuming instructions as *poisoned*, we flush instructions and restart Fetch again.
This seems impractical, and it is at the "user" level, but it is possible at the machine level. At the machine level, the instructions are, anyway, going to detect whether they are in SME mode or in Supervisor mode or whatever (otherwise our machine has a bug!) and that test can be anded in with whether an ISB is "active" and at least one earlier instruction has the potential to modify significant context and

has not yet completed.

The above is a fairly stringent set of conditions, so our assumption (presumably born out by simulation) is that it will rarely happen; the usual case will be that we never need this second flush, only the first early flush that happened when we decoded the ISB, and by the time new instructions have entered the machine from the I-cache, all prior instructions that are modifying page tables or security level or whatever have completed.

The case that's handled slightly differently is suppose that the ISB was preceded by a TLBI. In other words we have made changes to the page tables that might affect instructions. In that case the most energy efficient strategy is to delay fetch until the TLBI is competed. Conceptually we could use the same strategy as above of tracking poisoning, but every instruction would be poisoned (since there's a chance it could have been fetched using an invalid TLB entry) so why bother?

A similar sort of barrier is DSB (Data Synchronization Barrier) where now the goal is to segregate all load/stores on one side of the DSB from all load/stores on the other side. Once again the same sort of logic holds. The easy solution waits till the DSB is at the head of the ROB, then flushes and refetches; but we can be more aggressive, generating the flush as soon as we decode the DSB, and waiting to complete the DSB based on when pending load/stores have completed, not for all instructions to have completed, let alone the DSB has moved to head of the ROB.

In this case there is one addition tweak that you have to think about. In principle execution of a TLBI+DSB is supposed to halt the DSB completion until the TLBI has been acknowledged by all cores. The point is that suppose I send out a TLBI (so change page tables) then a DSB. The point of the DSB is to make sure that ever load/store later than the DSB sees the new page tables.
Suppose I use an aggressive scheme like described above, and, after the local DSB appears to be complete (local TLBI is completed, and local Fetch has restarted)
- I allow a later load/store to
- access a memory address that is cached on another CPU
- which has not yet completed its TLBI
?
In that case the data returned to me by the remote CPU cache may be associated with the incorrect (old) page table address.
So what happens under the traditional DSB processing is once we start the DSB executing (at head of the ROB) we don't allow it to complete until the TLBI messages have been sent to every CPU and acknowledged as completed. Meaning an even longer delay!

The scheme Apple seems to use as of this patent is to treat a load/store that required a remote cache interaction as poisoned, and flush it (or replay it) once we know all the remote TLBI's have completed. As
before this is a game of averages, we're assuming that most load/stores are local, and so most of the

time this sort of pattern (a remote load/store access during the brief window between when we started reFetching instructions and we learned all the remote TLBI's have completed) doesn't happen.

We could imagine more sophisticated solutions if necessary; for example pausing the load/store requests that require accessing a remote cache, as they're about to leave the L1, so that they aren't launched into the outside world until all the remote TLBI's have completed.

Or even, fanciest of all, tracking the most recent or two page details that have been invalidated, so that we can see if any load/store (or instruction fetch) matches the relevant address range, and if not, just proceed as normal! That would obviously be optimal, but may well be more hassle than it's worth?

There are additional elements to TLBI and DSB that involve what the remote CPU has to do when it receives the appropriate requests from the CPU that's executing the TLBI and DSB (let's call that the "host" CPU). Traditionally a remote CPU executed these commands as rapidly as possible because the host CPU could not proceed until these remote commands were completed. So a remote CPU would do something like flush all work in progress as soon as a TLBI/DSB "request" from the host was detected, (You can think of this as something like a snoop or an interrupt.)

But now that we've made the host processing of these commands more flexible, options also open up for the remote CPU. In particular Apple suggests that the remote CPU no longer needs to flush everything ASAP. Instead it can pause the pipeline at Allocate/Rename (or earlier, like Fetch) and drain the execution of all relevant instructions in the CPU, delaying the acknowledgement that the TLBI and DSB have been remote completed "enough" until all those relevant instructions have completed.
The details, as is always the case with these distributed protocols, get messy fast! But, as always, the idea is to replace an expensive complete flush with a more sophisticated tracking of the relevant affected instructions so that we can
- allow those instructions to complete (no flush) AND
- report the end of the remote TLBI/DSB and resumption of normal operations ASAP (no wait for other, irrelevant, instructions complete) AND
- overlap the resumption of normal operations ASAP (either restart Fetch ASAP, or undo the instruction pause at Fetch or at Rename ASAP).

(It's ALWAYS the replacement of
- a flush [very expensive sync] with
- a barrier [allow operations of particular kinds on one side or the other to flow through the barrier as appropriate] to, if possible
- a strand [detect the individual operations that need to be synced, eg by the specific load or store addresses, and order only those operations]!

Of course knowing the idea is easy! It's getting the details right in every case that's the hard part!!!)

TLB (translation) barrier instruction

Arm comes with a rich selection of barriers; even so they don't cover all eventualities.
(2025) https://patents.google.com/patent/US20250217207A1 *Translation Barrier Instruction*, from the prolific Jeff Gonion, suggests a new barrier type; presumably the intention is that it get added to some future ARMv9 profile.
Suppose that you make changes to the page tables (and it somewhat matters if the changes are to data

TLB or instruction TLB). Because of OoO, you have to ensure that subsequent instructions are delayed until after the TLB changes take effect. If the changes affect the I-TLB then every instruction after the change might, in principle have a changed mapping, so all instructions after the change have to be re-fetched. Likewise if the changes affect the D-TLB, then in principle any load or store after the change might in principle have its mapping affected.

Right now the way this is handled is with heavyweight barriers that flush everything after the page table changes; but the patent points out that this is more aggressive than necessary.

If such a barrier is provided (so that the CPU understands what needs to be done) multiple optimizations are possible.
For example, on the instruction side, on decoding an I-Translation barrier, the decoder could freeze at this point, waiting for all earlier instructions to complete. At the point of completion, it could then resteer I-fetch (which will pick up the changes to the I-page tables). This would save power and restart things at the absolute earliest time feasible.
More ambitious might be to mark the I-instructions after the barrier as speculative but allow them to proceed as far as possible. At the point where the prior (translation modifying instructions) are complete, it can be validated whether the translation changes affected any of the instructions in flight (the usual case will of course be no) and if necessary the speculative content can be flushed.
Something similar is possible on the D-side. We can allow all non-load/store instructions to flow through the barrier as the first implementation attempt. More ambitious might also allow load and stores to flow through the barrier all the way to the load-store queue (but no further!), and sit there pending until we know that they, likewise, are not affected by the earlier translation modification.
All this basically builds on the branch/memory aliasing speculation machinery already in place, so it's not too much extra work.

However right now the patent is more a stake in the ground than anything else. Presumably something will be negotiated with ARM as to the precise semantics, at which point implementation patents will start to implement some of the ideas above.

TLB

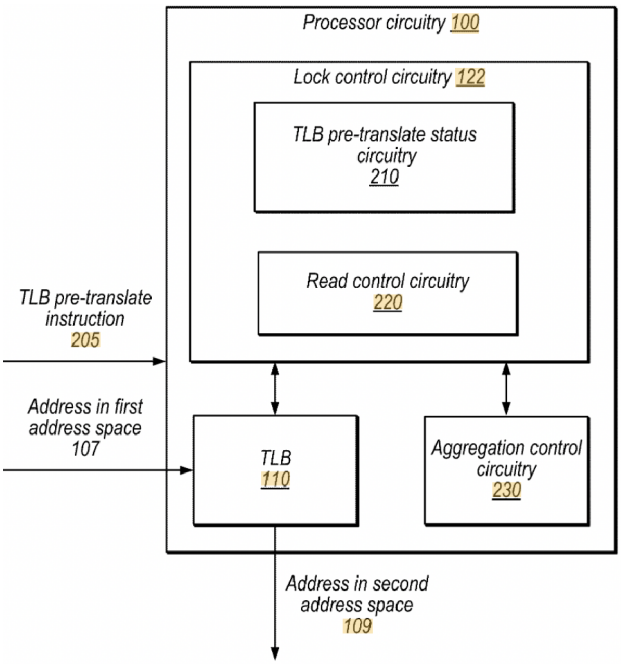
The great TLB mystery continues! We continue to see patents that suggest all sorts of TLB strangeness, but no obvious indicators in the real world (eg performance, APIs, ...) of these changes.
The latest such is (2023) https://patents.google.com/patent/US20250094355A1 *Translation Lookaside Buffer Entry Locking*. The idea is we add a "pre-translate" instruction to the (presumably ARM) instruction set. Nominally this instructs the TLB to look up and cache a translation, without accessing the address (either by fetching an instruction or performing a load-store), but the more important thing done by the instruction is that the value is then locked into the TLB.

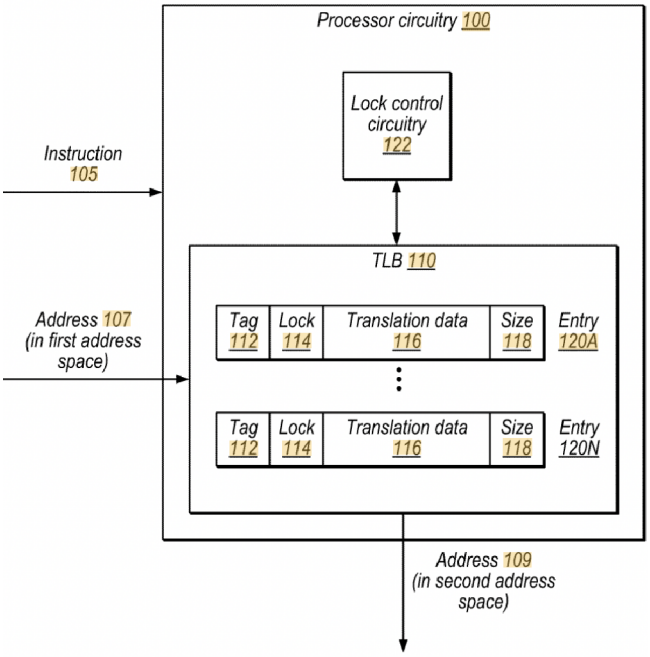
Now in principle you could imagine latency sensitive code that cares about having a locked TLB entry, but honestly this is a stretch and unlikely ever to be worth the hassle. Or does it make sense for Vision

Pro?

Here's another suggestion for what's really going on here.
The more important point is that the TLB (my guess is this is the L2 TLB, not the L1) is split into two parts. One part handles traditional fixed size page lookup, the second part referred to as a "sidecar" holds TLB entries of variable size. (As is fairly common, larger lookups may be translated into single page lookups for the purposes of storing the lookup in the L1 TLB.)
So obviously variable-sized TLB entries can be valuable, either for fixed-size large pages (like Intel uses them) or for a few variable-sized large pages (like PowerPC and POWER BATs [Block Address Translation], which could be used for things like a single entry covering the address range of a display's backing RAM). But where do these variable-sized TLB entries come from?

This appears to be the goal of this pre-translate instruction! The TLB has aggregation functionality so that when a new entry is added, it is compared to existing entries and, if appropriate, tacked onto the beginning or end of an existing entry, increasing the size of the entry.


Processor circuitry 100
Lock control circuitry 122
TLB pre-translate status circuitry 210
Read control circuitry 220
TLB pre-translate instruction 205
Address in first address space 107
TLB 110
Aggregation control circuitry 230
Address in second address space 109



In the diagrams above note the “size” field in a TLB entry.

We have previously suggested that the Apple TLB might be using coalesced page table entries, but normal COLT ties together a small (4 or 8 or so) page table entries and because of the small fixed size can use slightly different details (eg maintain a bitmap tracking state of each page in the coalesced entry).

This seems to be rather different, more like it’s trying to achieve the best of what BATs and large pages can handle. So what I think might happen is

- the point of the pre-translate instruction is to be used in a loop.
- the loop wants to do something like create a single BAT entry for a display. So the OS loops over the relevant virtual address range sending a pre-translate instruction for each page, which also locks the entry in the TLB, and each successive page meets coalescence criteria [contiguous virtual and physical address spaces, same permissions, etc] so each successive page gets accumulated till we have built up the BAT entry.

- the same sort of scheme could be used in other situations, for example when an app is launched we could map various common libraries (if we know they are never paged out...) each into a single variable-sized entry. Or if we add an API to provide large pages we could use that API as a hint to allocate say 1MB worth of contiguous physical address space which we then aggregate this way into a single large-page TLB entry.
After this large entry construction, we can leave the TLB entry locked (for some purposes like, the display or low-level libraries this might be a a good idea), for others we can unlock and leave the entry to its fate; if it's used enough it will stay

The overall scheme (if implemented!) is surprisingly ambitious. For example there's also a mechanism to detect an "invalidate TLB entry" instruction and use that not to invalidate the entire variable-sized entry but just remove a page from the beginning or the end. So we have a scheme that (at least for some purposes) almost starts to look like x86 segments, with their variable sizing and even the potential for dynamic sizing. Or to put it differently, a way to represent variable- and dynamically-sized memory regions without much TLB overhead.

This is admittedly a clever way to add what's effectively the functionality of BATs and large pages to the ARM ISA without having to modify the ISA (much...).

Here's a different variant on speeding up TLBs (though just as mysterious in the extent to which it is implemented). The primary caches in a system (CPU or elsewhere) are hardware managed, meaning that messages implementing MESI are automatically generated and transmitted between caches without SW being involved, and the advantages of this are obvious.
However not all caches, in particular the TLB, are handled this way. For various reasons, some good, some simply historical inertia, the T:B is software managed. So, imagine that page mappings have been updated (for example a page is swapped out). The OS needs to update every TLB in the system with the information that the previously existing mapping from a particular ASID and virtual page to a particular physical page no longer exists. This is easily enough done for the core on which this piece of OS code is executing, by executing some sort of TBLIE (TLB Invalidate Entry) instruction, but the same invalidation has to occur on every other CPU core (and in fact every IP block that might have a TLB). The traditional way to do this involves sending a software interrupt to each other core, in response to which the core will run the relevant interrupt handler and execute the equivalent of a local TBLIE.
There are various ways this can (and has) been made more efficient over the years. You could, for example, imagine something like a broadcast of the request, so that every IP block sees the same request, rather than looping sending the same software interrupt to every other core. You could also imagine more or less efficient ways of having the OS wait to be sure that the TLB change has been handled everywhere.

ARMv8 has a fairly nice abstraction for all this, in that the OS mainly has to say TLBI+DSB SY to solve the problem. The TLBI is responsible for telling "the system" to invalidate entries, the DSB waits until all entries have been invalidated. By operating at this level of abstraction, an implementation can choose

what it wants to do, whether that's a loop over interrupts, some sort of broadcast, or whatever.

So that's our starting point, and presumably Apple does something efficient in terms of getting the message out to every relevant IP block. But then what? As said, traditionally this would be treated as the receiving CPU noticing a SW interrupt and running an interrupt handler, which is expensive.
But is that necessary? Why not simply have the TLB itself see the bus request to invalidate the entry and do the work, without ever bothering the CPU? Now we're getting about as close to hardware-managed MESI as is feasible for handling the TLBs!

Now let's consider a different situation. Suppose we are executing a JIT. We may have a page of code that was just constructed by the JIT a few seconds ago. If the JIT determines that this is hot code, it may pause execution of the translated code, recompile that code taking more time, to make it faster, and then resume execution of the translated code. This means that the newly constructed code was in the L1I cache (being executed) then needs to be transferred to the L1D cache (to be modified by the JIT compiler) then needs to be transferred back to the L1I cache to be executed again.
The issue, on ARM, is that (unlike eg x86) the ARM L1I cache is not required to be HW-coherent with the L1D cache. So we have a situation similar to the TLB case where we have SW-managed coherence, only now the processor needs to execute an instruction to invalidate an address range within the L1I before constructing the new version of the code.
And once again we can have a version of this idea where data in a remote cache (if for example our interpreted code is ambitious and is running code on multiple threads) needs to be invalidated, ideally without the requirement for a software interrupt.

Finally we can situations that combine both of these, for example in we invalidate a TLB entry, we may wish to also invalidate code or data corresponding to the page being invalidated.
I don't *think* this is necessary for a peer CPU because once the TLB is invalidated, subsequent access by the peer CPU to the relevant lines in cache will be impossible, the lines will not have an address that can be generated by the CPU (since all accesses must go through the TLB). The lines will persist in the cache (taking up space, and so being a slight inefficiency) until they are replaced, but that's OK.
However other IP blocks like the GPU or ANE are somewhat looser in exactly when they force coherence (of the cache) or synching up to the physical address space (hence requiring an access to pass through the TLB).
Thus we can have cases where a "full purge" requires not just invalidating a TLB entry but also invalidating everything in cache corresponding to the virtual and physical address ranges covered by the TLB entry.
Once again the ideal situation would be to have the CPU emit some sort of abstract "invalidate cache range" instruction which gets broadcast in the most optimal fashion, and each relevant cache then handles the invalidation autonomously, without requiring execution of an interrupt handler on the target IP block.

All this is described in (2024) https://patents.google.com/patent/US20250103492A1 *Remote Cache Invalidation*. The patent actually describes everything in terms of a "primary processor" and a "coprocessor" meaning other IP blocks (eg ANE, GPU, etc), and talks primarily about invalidating the cache,

not the TLB.
But my guess (given how we've seen other elements of Apple's designs evolve) is that, as much as possible, more of the ideas I've described above (HW invalidation of TLBs, propagation of these ideas to CPUs) will be considered next.

(Slightly more) secure pointers

One possible security attack is based on APIs that, somehow, allow the provision of an index into an array. That index is multiplied by some object size and added to a base pointer. If you provide a very large or negative index, you can maybe generate a net result (base+index*size) that points to somewhere attackable on the stack or in the heap.
Obviously in a perfect world we use languages or APIs that are hardened against this sort of thing, but right now we have to accept that older languages, non-hardened APIs, and bugs exist. So can we improve things slightly?
The idea is to essentially define that a pointer is not 64 bits long but 60 bits long with the 4 high bits forced to 0. This is obviously a subset of the various other pointer taggings that are part of, or being suggested, for ARM, things like MTE.
If you do this, then during the construction of the ultimate pointer that will be used by the load/store instruction, there will be an addition of the base with an index*size that is "too large", probably even negative, and this will result in changing at least one of the tag bits during the generation of the ultimate load/store address. At which point the machine can fault in some way.
Obviously there are many precise details to be worked out, presumably in conjunction with ARM Ltd, to make this a full part of the ecosystem, though Apple could probably go ahead on its own before standardization.

Small processors

As we know, Apple has at least three processor lines, the P- and E-cores, and the Chinook class companion cores used to control the GPU, ANE, and similar such items. (There may also be even smaller cores, the equivalent of an ARM M0 or so, who knows?)
The companion cores are 64-bit ARM and, for the most part track the same ISA as the visible cores, though they may drop some elements that are not required. (They surely don't have AMX/SME! And they appear not to have FP/NEON.)

Now suppose you're creating a small core where the most important elements are area and power, not performance. What are your options? Generally you'll start with an in-order 1-wide non-speculative non-pipelined core. Then, as appropriate, probably in this order, you'll add pipelining then 2-wide then maybe a limited form of branch prediction or limited out of order. There have been a few ideas suggested for how to provide limited out of order execution without having to pay the costs of the full high performance package (register rename, ROB, and all the rest). (2023) https://patents.google.com/patent/US20250094173A1 *Processor with out-of-order completion* suggests one more possible scheme.

I honestly don't see what's patentably new in their design, but we don't care about the legal issues, just what is revealed about the new core. The (presumably new Chinook-level) core seems to be some-

where between a Pentium and a PPC 601 in sophistication. There appears to be an FP unit (though perhaps without NEON capabilities) and 2-way issue.
The particular element they seem to have chosen for handling complexity is to move all this into a hazard detector at Issue time. So the hazard detector looks at both the one or two next instructions and compares them against what's presumably a scoreboard of previous instructions to decide whether one or both (in order) are allowed to proceed. Constraints include that registers to be read must not be busy, registers to be written must not be busy by the time the register will be written, etc.
The main focus of the patent, which is hardly new, is that if the hazard detection also checks when instructions will write their results (which is known for most instructions, though maybe not for, eg a load instruction) then we can avoid issuing an instruction B which will land up writing back in the same cycle as previously issues instruction A. This allows us to avoid various buffers or temporary storage that would otherwise be required to handle such a collision.

AMX/SME

"Fusion Buffer"

(2024) https://patents.google.com/patent/US20250103338A1 *Processor Operand Management Using Fusion Buffer* hints at something rather different from what it is; it's a power patent not a performance patent.

The standard execution of AMX/SME instructions, especially the vector instructions, follows the pattern of load a register or two from storage, perform the operation, then store the result back in register storage. However some instructions (especially those added once SME replaced AMX) require intermediate storage. Consider permute-style instructions (this can be things like vector interleave ["zip"] or table lookup). These can't really be executed in the "main" AMX outer product array of multiply-adders; they require some auxiliary hardware and, crucially, they require some temporary storage in that permute hardware to hold the result of the permuted vector(s).
This means that after we construct a result in this temporary storage, we pay some energy cost to move it back to register storage, then again to move it out to a subsequent operation. This is not ideal in terms of power, and is the problem we wish to solve.
The idea is to capture a sequence of operations that begin with a permute-style operation, followed by a compute-style operation using the permuted vector, so that we can dispatch the fused operation as a single unit which only needs to moved data from the permute unit to the "main" compute unit bypassing the larger transfer to and from register storage.

This gives the basic idea. Once you have this in place, you can add a few additional optimizations. For example given a sequence like

```
permute R1, R2→R3
compute R3, R4→R3
```

R3 is overwritten by the compute operand, meaning that there's no need to ever write back the result of the permute, it can be dropped.

You can also see this as providing another (very limited) sort of OoO mechanism insofar as instructions can be fused that are not back-to-back, as long as various constraints are met.
Additionally the amount of temporary storage available in the permute unit appears to be larger than just a single output vector, and the fusion mechanism is smart enough to be able to exploit this where feasible.

The companion patent (2024) https://patents.google.com/patent/US20250103551A1 *Interleave Execution Circuit* describes how (part of) an appropriate permute unit might be designed.

interaction of SME/SSVE with NEON

An unfortunate aspect of how ARM has defined SME arises from the following:
- SVE defines new Z registers whose lowest 128b are aliased with NEON registers. And some assembly hackers have even started writing code that exploits this fact, for example doing some work in NEON, then switching to SVE for one particular SVE instruction. So far this seems harmless.
- SSVE (by definition) uses those same SVE registers, but if you're following Apple's implementation model, the Z registers are now in a separate co-processor isolated from each core by 15 cycles or so of latency
- and SME likewise uses those same Z registers.

So consider what happens when I want to use the Z registers, either for SSVE or for SME. Do I have to transfer the NEON registers to the SME co-processor? Do I have to ensure they stay in sync when I modify the NEON registers?

Fortunately Apple has given us a very clear answer to this in
(2025) https://developer.apple.com/download/apple-silicon-cpu-optimization-guide/ in Table 4.4 (of the 2025 version). The answer is that
1. you cannot access the Z registers outside streaming mode
2. starting up streaming mode (SSVE or SME) makes the Z registers available and CLEARS the NEON registers
3. accessing the NEON register while streaming mode is active is illegal and will result in a crash

This is very clear and means there is no real problem in enforcing synchronization between the Z and NEON registers; there simply does not exist any such aliasing on Apple platforms regardless of what the ARM spec says. In fact Apple is simply repeating the ARM spec, which says that whenever you toggle Streaming Mode (either SVE or SME) on or off, you have to go through the above steps.

So now look at this from another angle.
Overlapping SVE and NEON registers seems reasonable.
The REAL sin, the thing that makes this so dumb, is insisting that the registers used by SVE (the Z registers) are the *same* registers used by SSVE and SME. Look at the three above statements. There is *no way* for the SSVE/SME registers to affect the SVE registers and vice versa. So why pretend they are

the same registers?

What ARM should have done is call the SSVE/SME registers some new name, like the Y registers (likewise for predicate registers, maybe the J registers) and stop the pretence above. This would
1. avoid a whole lot of confusion
2. avoid time/energy wasted wiping out the NEON/SVE registers when toggling SSVE/SME
3. allow NEON (at least) to operate simultaneously with SSVE/SME. (If you also want SVE to operate simultaneously, and why not, then you also need to duplicate a few status registers like the SVE length register, but that's pretty minor).

The whole thing feels like a cocked up, adversarial negotiation between Apple and ARM, where ARM wanted SVE to be the next big thing; Apple wanted AMX/SME; and eventually in frustration Apple accepted this idiotic and unnecessary pretence that SSVE/SME is a variant of SVE rather than something completely different.
A real example of ARM cutting off their nose to spite their face, making the entire ecosystem worse just so they can look like they "won" in some argument with Apple that no-one outside ARM cares about!

ANE

ANE Lookup Tables

Obviously an activation (ie a non-linear function) is required between each layer of a neural network, otherwise the layer could be coalesced with the previous layer in a single matrix multiply .The foundational ANE patents thus refer to multiple places in the data flow where data can route through an activation function. More details than this were never given, and my assumption was that these were just a choice of a few "easy" activation functions, like ReLU.

Maybe that was the case in the first round or two of ANE? However it seems like, at least in more recent ANE designs, the system allows for generic activation functions.
(2020) https://patents.google.com/patent/US12189599B2 *Lookup table activation functions for neural networks* describes how these are implemented.
First we indicate if the desired functions has one of a few obvious symmetries [eg $f(x) = -f(-x)$ or $f(x) = 1 - f(-x)$], which allow us to shrink the size of the table encoding the function. Next we encode the function as a set of tables, each table giving a piecewise linear approximation to the value of the function over some range. The details are of less interest than the fact that at least recent ANE's can handle more or less arbitrary activation functions.

Lower power ANE?

This one is very strange (2023) https://patents.google.com/patent/US20250060939A1 *In-memory computing devices for multiply accumulate*.

First point to note is that at least one of the inventors is associated with other ANE patents, another seems to be associated with the CoreML team, and another seems to be associated with SRAM details. So this seems to be associated with the legit ANE product, not blue sky optimism.

Second point is that this is not just another PiM patent of dubious ultimate real world significance; the difference is this is not about computing on the DRAM die (most PiM) or on/near the flash die (Samsung's version of PiM). Rather this is about computing "in" SRAM. This is a strange idea that I've never seen discussed elsewhere, but I assume it makes some sort of sense in terms of area, energy, and performance.

The simplified version is that we associate with the SRAM a very simple multiplier that can essentially multiply an integer by 2 bits. Outside the SRAM we have some shifters and integer adders.
The end result is that we can perform a dot product by multiplying one vector by two bits of another vector, shifting out the result and adding it across these partial sums, then multiplying the vector by the next two bits of the other vector, shifting and adding, etc etc.

So what does this get us?
As far as I can tell on the negative side, we give up floating point (at least via this path; presumably we will still have FP in other parts of the design). This is integer only.
But on the positive side, it becomes both very flexible (by slightly tweaking the sequencing of operations we can handle all sorts of combinations like three-level (+1,0,-1) weights with 8-bit activations, or 6-bit weights with 16-bit activations, or whatever. We simply take an additional cycle per 2 bits of multiply.
It also (pretty much) does the arithmetic as part of data movement. We never actually move the large items (vectors) out of the SRAM, all we move out a short distance is the partial products, and then over a longer distance the ultimate single value dot product.
It may also cost almost nothing in term of area. Remember all these scare stories we've been hearing for a while about how SRAM scaling has been flat since N5? SRAM can't shrink because of wire density, but maybe with clever routing you can lay down some logic interleaved with the SRAM, and if the wiring does not clash, not take up much more SRAM area?

We may even be able to get the multi-cycle dot product (loop over the input bit pairs, and maybe a loop over the dot product sum) to happen in parallel with data movement into the SRAM, just so long as we stage things carefully so that maybe one bank is computing while a second SRAM bank is loading new data?

This looks like it could be very very interesting performance boost for ANE while reducing power even further. And amusingly, in true Apple ANE fashion, it takes Apple even further away from the nVidia path (dedicated tensor units, FP8 and FP4) and all that implies.

GPU

More efficient GPU atomics

Recall the idea behind atomics.
Suppose we wish to update a counter of “events” shared across multiple threads. The naive sequence of operations will be something like
- load counter value, from the address of the counter (which may be a load from DRAM or some cache), into a register
- increment the register
- store the register to that same address.

The problem with this three step operation is that we can have things happen like two threads, both wanting to update the counter, in more or less the same cycle perform the first step (read), then each independently increment their register, then store, with the later store (whichever that may be) overwriting the earlier store. Bottom line is only one of the two increments gets recorded, and what went wrong was in the first step, where both threads were able to read [with intent to increment] the same value of the counter.

There are multiple possible solutions to this problem and you should know many of them (“lock prefix” on x86, Compare-and-swap instructions, load-linked + store-conditional pairs, etc).
But all these standard solutions assume that the increment ultimately happens in the CPU.
One of the more interesting alterative solutions is to perform the increment not in the CPU but in a cache, a so-called *remote-atomic*. Instead of loading the data from the address of counter into a register, and all the subsequent steps, we simply send an increment message to the address of counter and rely on the cache to perform the increment.
This has all sorts of nice features (eg lower energy because less data is moving around, lower latency [only one transaction, out to the cache], and more or less “naturally” the queuing of transactions present at the cache will ensure that multiple increments arriving at the cache will be processed successively, so that no increment is lost).
There are also features you have to be aware of. Firstly this works best if you rarely want to know the counter value; maybe you only want to know that at the end of all processing. If you want to know all the intermediate counter values then you can’t just leave the work in the cache, you will have to load the value back into the CPU. Making sure you get the “correct” value will require some sort of synchronization barrier which will be expensive. Secondly I’ve used the term “cache” here somewhat loosely. Do we want to perform this operation in the L2 (in which case it only efficiently covers the cores of one cluster), or in the SLC?

Well, those are concerns for the CPU! That’s all background because now I want you to consider the same sort of problem in the context of the GPU. The main thing that differs for the GPU is that a single

"core" can be executing up to 128 lanes (4 SIMDs each 32-wide) all of which might execute the same atomic increment in the same cycle. Imagine, for example, that we are classifying triangles as "big" vs "small" and atomically incrementing two counters, smallCtr and bigCtr. In any given cycle many lanes of one SIMD, or all four SIMDs in a core, or even multiple SIMDs of multiple cores, might want to increment smallCtr, then a cycle or two later other lanes will increment bigCtr. (Or alternatively, maybe the lanes have set an address register to smallCtr vs bigCtr, so there's a single atomic increment targeting different addresses in different lanes.)
How can we make this fast?

It should be fairly clear that this is a situation where we don't care about the intermediate counter values till the end of the operation, so a remote atomics solution seems optimal. And it seems clear that we'd like to enqueue the atomic operations in some sort of buffer so that we can fire and forget, continue executing right after the atomics without having to wait for the bus transactions. But can we do any better? For example we'd really like to do as much work as we can within the L1 cache, but if multiple cores are involved, don't we have to do the increments in L2?

These are the concerns of (2024) https://patents.google.com/patent/US20250086116A1 *Atomic Smashing*.
As you might imagine, the details become horribly complex, but the basic idea is that
- most atomics don't care about the order of the operations. (Obvious for increment, but just as true for eg atomic add or subtract, or and/or, or even max/min.) The only case where you might have a problem is in the abomination that is floating point atomics (which probably don't get to join in this fun).
- so we gather the atomics locally at the appropriate addresses in local core storage. (Nominally in "the L1 cache" but conceptually a separate table that's tracking, for a few addresses, the type of atomic that's being performed, the cumulative value and a few other details.) So if a SIMD generates 32 atomic increments, these will be queued in L1 and will successively increment the appropriate [counter, address] pair in this table.
- when certain qualifying events occur, the accumulated value will be sent out to the L2 where all these increments will be added as a single value to the counter value that's accumulating in L2. Qualifying events include things like certain cache events [eg the relevant cache line is being evicted], a change to the counter value that doesn't follow the previous pattern [eg we switch from atomic increment to atomic and], or some sort of synchronization event that would occur before an attempt to read the counter value.

This is pretty clever stuff! It's obviously helpful to the GPU, but one could imagine something similar possibly being implemented on the CPU side, especially as CPU count grows larger, if the mythical M4 or M5Extreme based on four or more chiplets ever ship.

More aggressive GPU compression

Consider the idea of "texture" compression. One can imagine this implemented more or less aggressively.
Least aggressive is off-line compression of a texture, which is decompressed by HW, perhaps as it passes through the SLC, perhaps as it enters the L2 cache.

More aggressive is to perform the decompression as it enters the L1 cache. Now we don't have to waste space in SLC or L2 on decompressed texture.
But this still only handles pre-compressed assets. How about we treat the generated image as texture and allow that to be compressed (possibly always lossless, possibly allowing lossy if other priorities, like using no more than a particular memory bandwidth, are higher priority). Again we have options. Simplest probably is to perform the compression as the data leaves L2, as part of a final "dump" to the rest of the system of the newly generated image. That gets us a lot, but only covers the final generated image.
Most ambitious is to imagine something like:
we allow all "textures" to be compressed (some losslessly, some lossy). Recall that many intermediate graphics structures (normal maps, bump maps, procedurally generated imagery, etc) are textures.
We may even have in mind compression of non-texture data (Apple suggests, for example, compression of ML model weights, even as the model is being trained, so that weights are loaded for this round of training, decompressed on their way into the GPU, modified and then written out, to be recompressed.)

This all sounds great, but comes with obvious technical challenges.
One is accumulating enough intermediate data to cover the block size used for compression. This fact of life (whatever the data is, it may be modified in a somewhat random access fashion) means that we may well have to write out data to the L1 cache corresponding to modified, uncompressed, partial elements of the texture or whatever.

This leads to the second complication which has to do with data consistency and coherency. Conceptually data is tracked between caches by its address; we ensure that data in cache A matches data in cache B by ensuring that if a line shared between the two caches [ie same address] is written to, the line is first acquired as exclusive, and all the rest of cache protocol behavior.

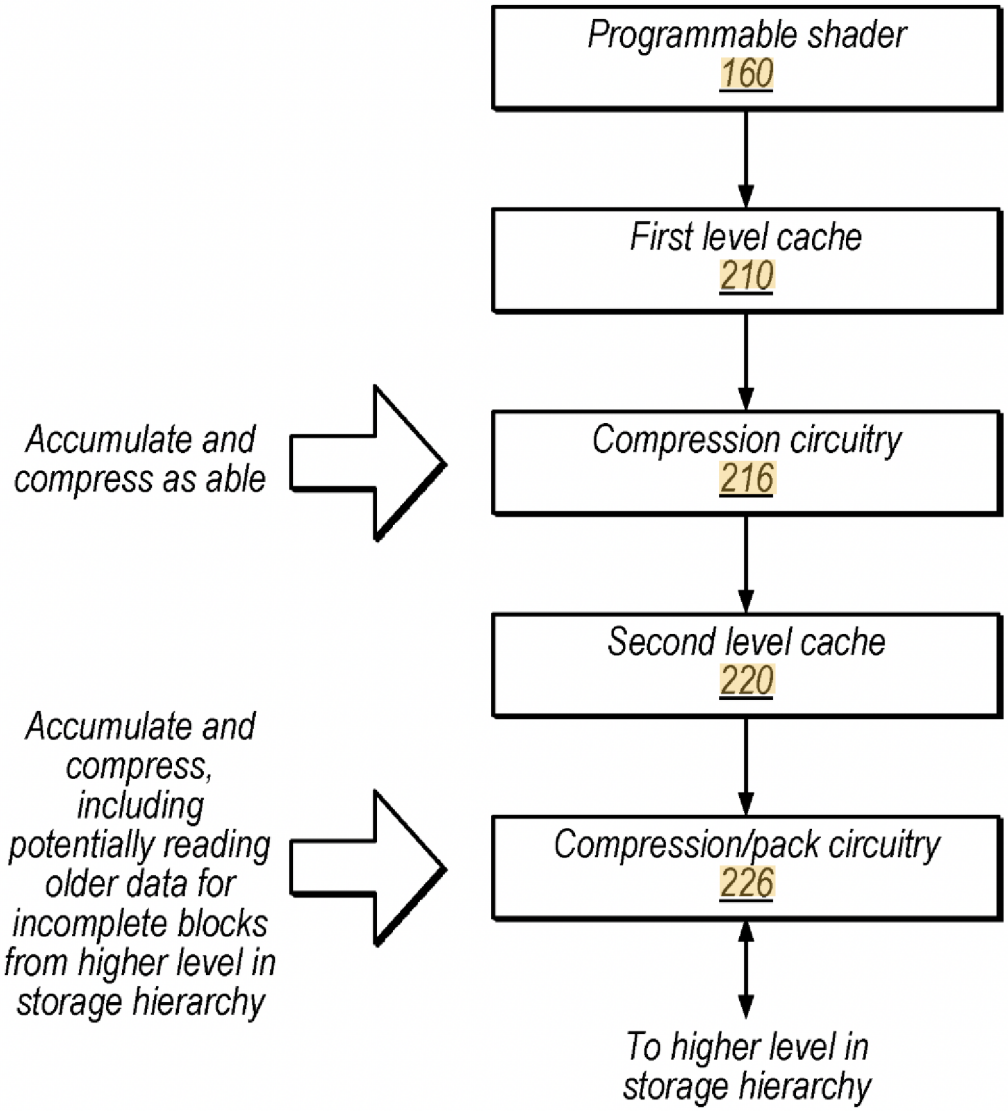
So let's think through this all a little more deeply.
The Apple Texture Compression scheme, as far as I can tell, takes in units the size of a cache line (so let's assume 128B), in Morton ordering which, recall, is a locality-preserving linearization of pixels in a 2D or 3D (or *n*-D) array, and compresses these either 2:1 or 4:1, either losslessly or lossy. This unit of a cache line size is called a subblock. These will then pack into larger units which fit into a cache line (eg two sub blocks for 2:1, four subblocks for 4:1) so let's call these macroblocks. In fact for various reasons Apple's macroblocks may sometimes be larger, perhaps two or four cachelines, depending on the codec.
This gives us a natural reason to track things at both the subblock level and the macroblock level.

Next suppose we decompress a subblock from a macroblock on its way from L2 into L1. Where (in terms of address space, or in terms of physical space) do we store the decompressed block? You can imagine many ways to look at this, but supposed we want *transparent* compression. In a sense, then, what we want is that the GPU uses an abstract addressing ("give me the pixel at [texture ID; *X*; *Y*]") not give me *N* bytes at address *A*. If the GPU does use such an abstract addressing, then we can define something like two address ranges, conceptually, for example, a compressed range at address A, and a

non-compressed range at address A+(some large 64b offset). As long as we allocate address space correctly, this will work out and, by looking at an address we can easily figure out if it's compressed or uncompressed. Either way we can do the correct thing, either load from the uncompressed address range [which hopefully hits in L1] or load from the compressed range [which may hit in L2, and then we decompress it on the way into L1].
The texture unit or cache is responsible for tracking how texture ID maps to address A, and transparently converting [texture ID; *X*; *Y*] to the appropriate uncompressed address.

This sort of scheme might work if we're only dealing with texture reads. What if we also want to compress outgoing textures? Apple has a patent for this originally dated 2019 https://patents.google.com/patent/US11488350B2/ *Compression techniques and hierarchical caching*. Superficially this looks like what we're after, but this patent (as far as I can tell) deals with a rather simpler situation. In particular the 2019 patent seems to be primarily about display compression and similar limited situations. It seems to consider situations where either we generate an image/texture from scratch, or we have a very constrained situation where a compressed image/texture is loaded, modified in some trivial way, then sent back to memory.
The point about both these cases is that there's never a question of *which* version of a texture the GPU core should be reading. Either there is no prior version of the texture (the display case) or the prior version is read in, modified, written out. What doesn't happen is a situation where both the old compressed and the newly generated texture can both be present in cache (so no sort of end of frame/kernel flush has occurred) *and* code may want to access the texture.

Before we get to this, let's look at how this read (zero or once) then write situation works.


Programmable shader
160
First level cache
210
Accumulate and compress as able
Compression circuitry
216
Second level cache
220
Accumulate and compress, including potentially reading older data for incomplete blocks from higher level in storage hierarchy
Compression/pack circuitry
226
To higher level in storage hierarchy

The important elements make sense if you remember what we said about macroblocks and subblocks. The Compression Circuitry that sits between the per-GPU-core L1 and the shared GPU L2 compresses a cache line (a subblock) by 2 or 4x. This can be stored in some fragment of an L2 cacheline.
When the cacheline is in turn cast out of L2, various cases may happen. Ideally the full set of compressed subblocks is present in an L2 line, and so the line can be transferred directly to SLC. This presumably happens when generating an image. But if we're modifying an existing texture, our modification may only have touched part of the existing texture, and so we have some L2 lines that only have part of the L2 line containing valid subblock. We thus need to load in the appropriate material (maybe from SLC, maybe it's still present in L2) and pack it into the "holes" in the line that we're about to cast out.
This obviously may take a few cycles, which means that if we defer this all the way till the end of a set of kernels, when an L2 flush occurs, we'll have a bottleneck. So we also have a timer attached to these lines so that, at a fairly steady rate, the lines are packed and written out while the later parts of the frame are constructed.

Once you think through these elements of the problem, you can see why we actually need this more complicated setup:

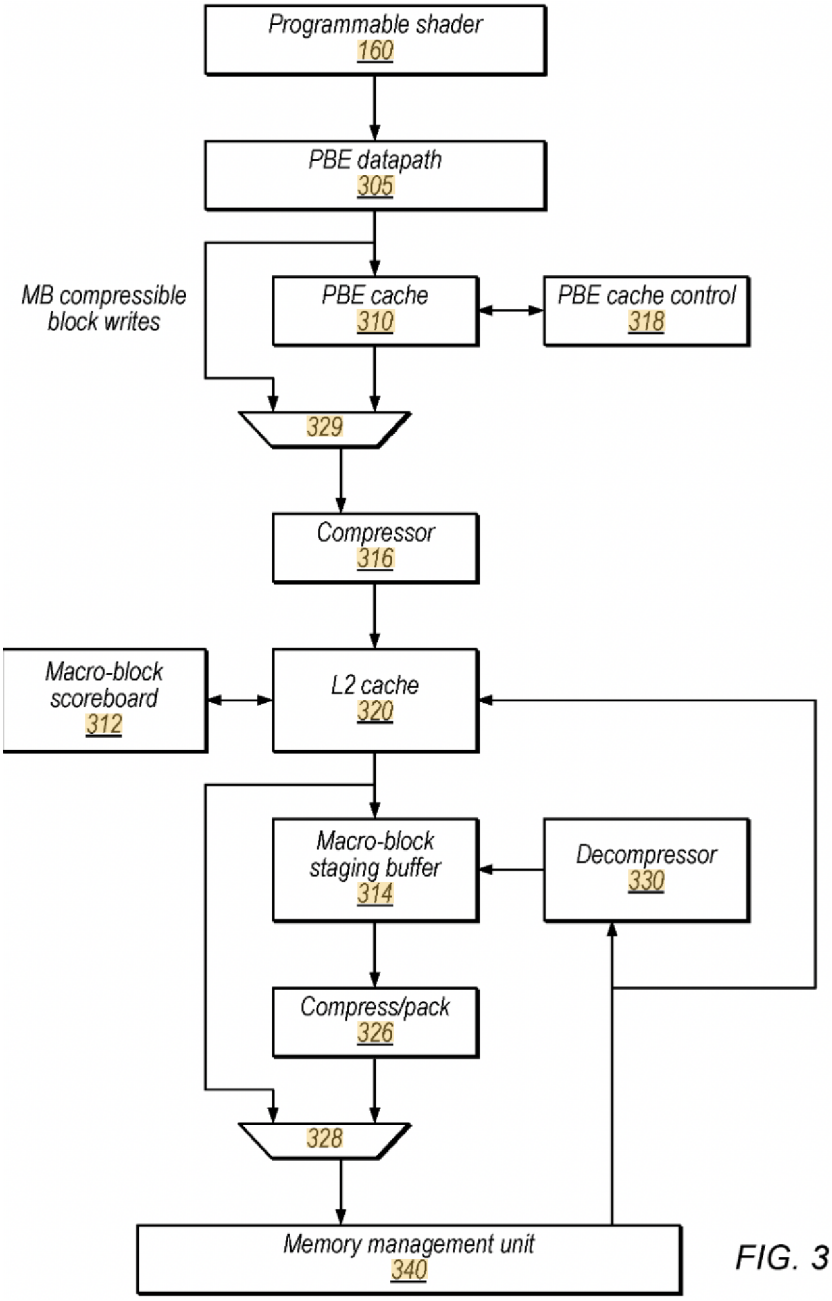

Programmable shader
160
PBE datapath
305
MB compressible block writes
PBE cache
310
PBE cache control
318
329
Compressor
316
Macro-block scoreboard
312
L2 cache
320
Macro-block staging buffer
314
Decompressor
330
Compress/pack
326
328
Memory management unit
340

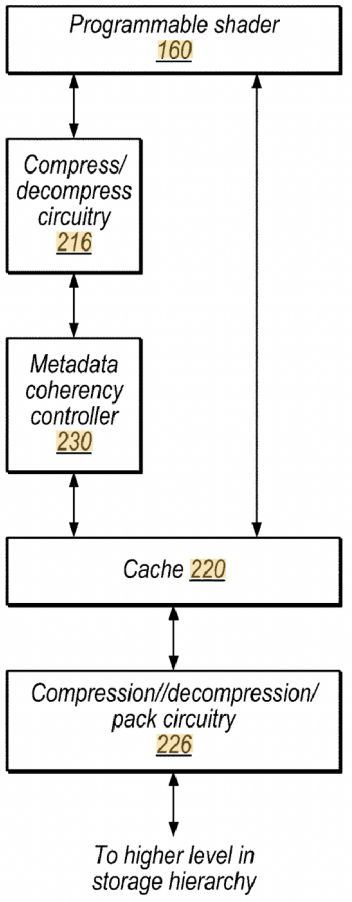
FIG. 3

Now the L1 cache is expanded to “PBE” cache (pixel back end) cache and cache control.
More interestingly we have Macroblock scoreboard 312, which is tracking the properties of each macroblock in the L2 cache. For example it’s tracking which subblocks are valid and which are not, also the timer associated with trying to consolidate the macroblock before writing it out (but not leaving this till L2 flush), also the compression algorithm used, and similar sorts of metadata.
(Memory Management unit 340 just refers to the SLC or something similar, ie memory outside the GPU).
So as blocks are transferred out of L1 and through Compressor 316, the appropriate metadata for the relevant macroblock is updated in the scoreboard 312.

As we’ve said, the scheme adds some value to a system that only *reads* compressed textures (for example it can save substantial bandwidth on writing out display frames) but it’s limited in what it can do to reduce bandwidth for all the intermediate textures, bump maps, normal maps, etc that are utilized while generating a frame. The scheme falls apart once compressed textures can be modified because we hit a “consistency” problem.
In these earlier cases, it didn’t matter if we weren’t too precise in how we tracked compressed vs uncompressed because either way we’d get the correct data. But if we’re now modifying data that we may later read, the problem changes. In fact, and this is the best way to think of it, it becomes like a cache coherency problem: I have data in two caches (compressed in L2, uncompressed in L1) and now I modify the data. I want to ensure that all subsequent references only see the latest (ie the modified) data…
The advantage of this viewpoint is that it makes us consider a similar type of solution. We “solve” caches by having a cache directory (ie a set of tags, one per cache line) storing the line state, and an invalidation protocol that, as appropriate, either toggles the states of these cache lines or invalidates other cache lines.

This is, more or less, the solution Apple adopts, but there are a few details that mean we can’t exactly use the existing cache scheme. These include
- we still need two address ranges for our compressed and uncompressed data (so the directory has to know when two addresses refer to the same texture)
- we need to operate at the sub-cache-line level because a compressed subblock both corresponds to a full uncompressed line and half, or a quarter, of a compressed line.
(Also Apple refers to the addressing as “coarse” vs “regular” hash. I’m interpreting that as essentially meaning the two address ranges I described above, though one could implement these two address ranges in various different ways.)

We already saw something like this with the macroblock scoreboard above, but our new directory also needs to be present in the L1 cache. Our new setup looks like



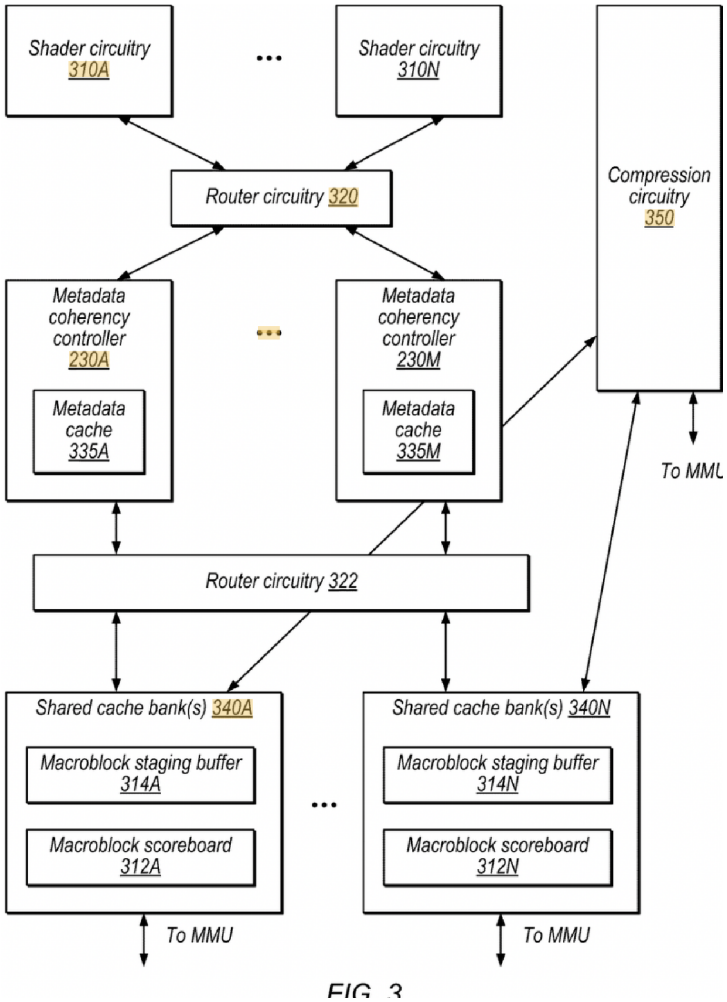


FIG. 3

In the above left 220 is the L2 cache, and we don't see the L1 cache, but we see that there's now a new element Metadata coherency controller 230, that sits between the L1 and L2. (You can think of this as

being present in L2, so that it's relevant to the entire GPU, but some of the data in the metadata controller are cached for performance in any relevant L1).

The above right shows the full horror. The important elements are that the L2 cache is shown as multiple banks (for bandwidth, so that ideally any particular GPU core is hitting one bank in a cycle while other GPU cores are hitting other L2 banks). Likewise the macroblock control areas (staging buffers and scoreboards, as we've already seen) are replicated across each bank. The metadata coherence controllers (MDCC) are also replicated but possibly not at the granularity of one per L2 bank.

So the flow now is that elements of the MDCC that refer to blocks currently in a particular L1 are stored in that L1. An access to a particular address can flow through this cached MDCC (kinda like it could flow through a TLB) to see which of the two address spaces (compressed or uncompressed) holds the data. That solves part of our problem. This ensures that the later of compressed or uncompressed data is read.
The second part of the solution is that an entry in the MDCC can be locked. This is, in a sense, like a cache like being marked as modified. This lock also propagates from the local L1 cache of a subset of the MDCC up to the full MDCC and thus to the other local MDCCs, somewhat like cache snooping.
Once a line is locked, only the L1 cache with the MDCC that holds that line and that lock can now modify the line. This solves a second problem, namely ensuring that two GPU cores don't try to modify the same area of a texture at the same time, and that they are informed of when their cached area of texture has become invalid.
Since this scheme is something of a hybrid between a traditional locking scheme and a cache protocol, it's every bit as complicated as you might imagine, and the rest of the patent is state transition diagrams, timing diagrams, and the usual paraphernalia of these sorts of schemes.

The last thing to say is that it seems to my eyes (though I could be missing something...) that, as the patent suggests, there's nothing here that explicitly ties this to textures. If you used any sort of compression scheme with the same sort of structure (eg that single cache lines are compressed down to a half or a quarter line, so that we can keep track of things fairly easily by tracking the subblock and macroblock level) the scheme should work for such data. Including, as Apple suggests, AI data, not just read-only weights, but also temporary data while training. The scheme also doesn't care much about lossy vs lossless compression; textures can use either, and AI weights are similarly somewhat flexible.

Even more so, there seems no obvious reason that this couldn't even be transferred to the CPU, giving us a scheme with uncompressed data in the L1, a mix of compressed and uncompressed in L2, and almost data transferred to/from the SLC compressed (in this case always losslessly). But of course the CPU is more sensitive to latencies, and that may mean the scheme (in particular routing all requests through the MDCC) may be impractical for at least one reason?

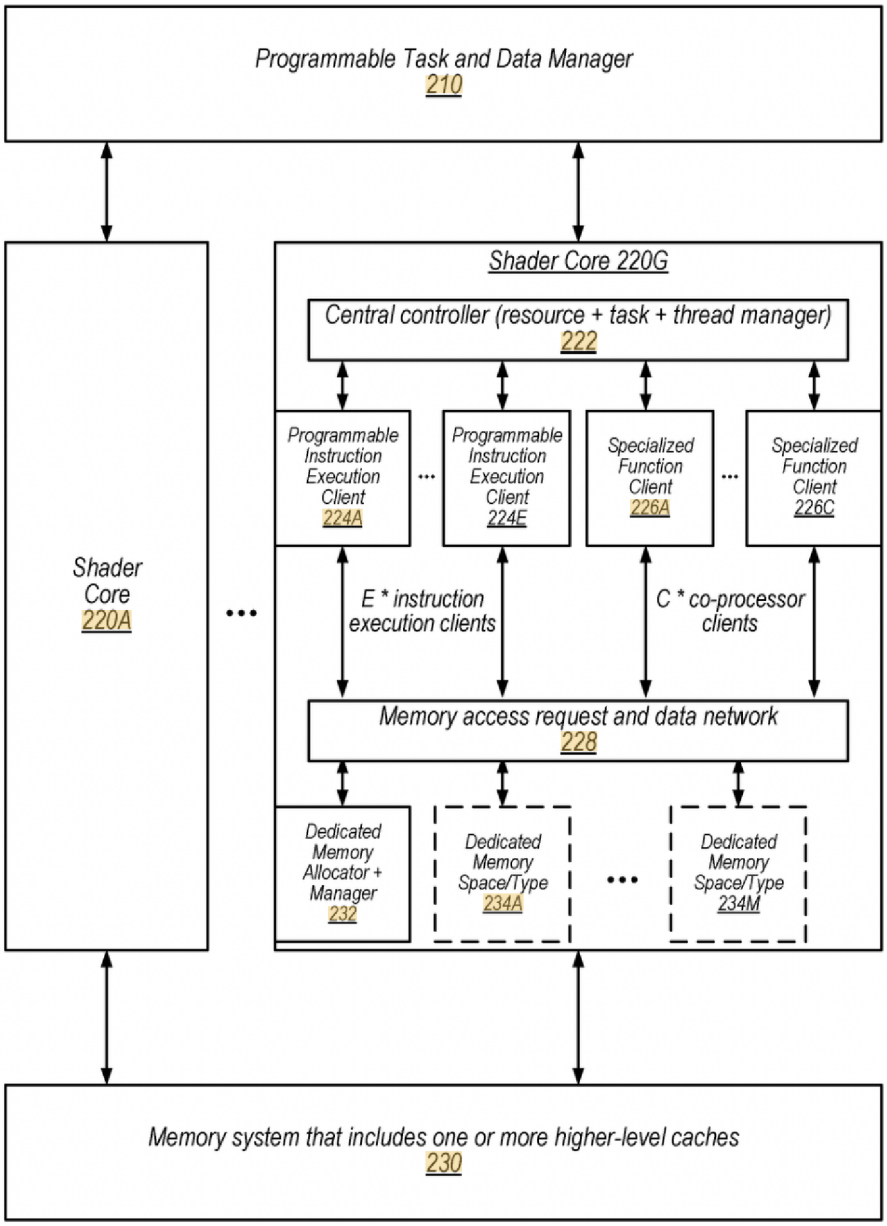
The A19 presentation mentioned on a slide something called "Unified Image Compression" but gave no details. My guess is that it refers to what I've described here.

One more aside. You might want to take a look at the dissertation (2018) https://uu.diva-portal.org/smash/get/diva2:1257495/FULLTEXT01.pdf *Understanding Task Parallelism*.

Much of this may not interest you; the summary is a set of tools were developed to understand data reuse between different tasks. However starting at pg 35 these tools are used to analyze real-world rendering of complex scenes (like eg the standard Manhattan benchmark scene).
The dissertation shows just how many rendering passes and intermediate buffers are used in constructing these scenes (which shows the value of being able to compress these intermediate buffers...)
It also discusses the amount of data reuse within a frame and between frames, which should give you some real world feeling for both why Apple structured much of Metal as it did (explicit buffers holding resources, so that it's reasonably easy to track resource use within and across frames) and the value of being able to cache *appropriately* these resources, justifying Apple's fine-grained control of the SLC. For example he suggests (just as a quick experiment, obviously a production system should be much more sophisticated) that you get most of the value of caching by tracking the ten most re-used resources and caching those, while not even bothering to cache resources beyond these ten most re-used (these just take up space while providing no benefit).

Structure of the GPU L1 cache

(2024) https://patents.google.com/patent/US20250068564A1 *Graphics Processor Cache for Data from Multiple Memory Spaces* doesn't say too much new, but confirms some things we've suspected/pieced together about how the GPU works.
Let's start with this diagram of how things used to be:


Programmable Task and Data Manager
210
Shader Core 220G
Central controller (resource + task + thread manager)
222
Programmable Instruction Execution Client 224A
Programmable Instruction Execution Client 224E
Specialized Function Client 226A
Specialized Function Client 226C
Shader Core 220A
E * instruction execution clients
C * co-processor clients
Memory access request and data network
228
Dedicated Memory Allocator + Manager 232
Dedicated Memory Space/Type 234A
Dedicated Memory Space/Type 234M
Memory system that includes one or more higher-level caches
230

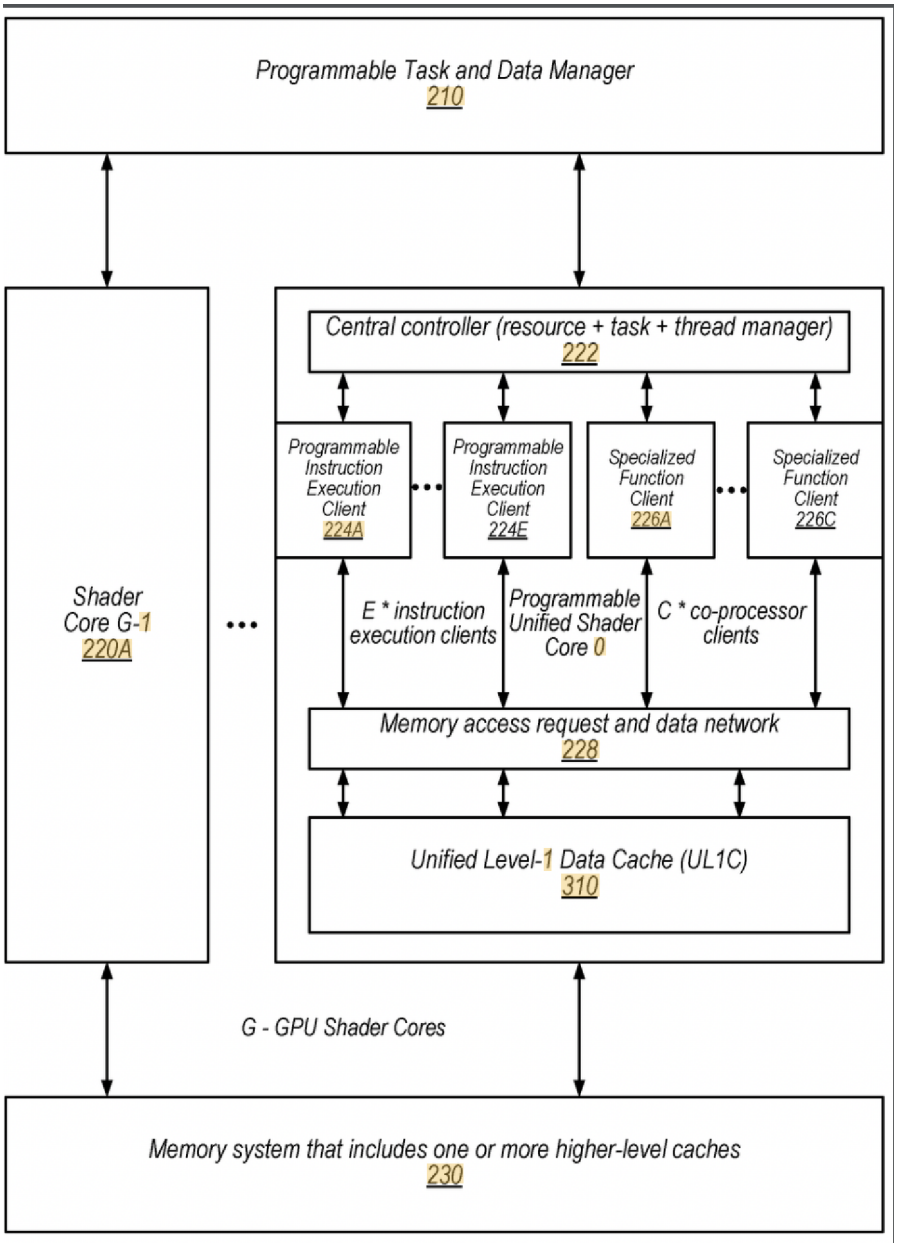
So as expected we have a the Manager at the top (we assume a Chinook-style lightweight ARM core) submitting tasks to the GPU cores. And we have different pools of SRAM (234A..234M) which hold things like registers, the threadgroup scratchpad, ray tracing storage, and the L1.

Now obviously the next step is going to consolidate these, but before we go there, let's take a slight detour to the CPU. The CPU utilizes a VIPT (Virtually Indexed Physically Tagged) cache, and the part of this that we care about is that all lines in the cache are *physically* addressed. If a system is to run any form of Unix (and so is built upon `fork()` as the process-creation mechanism, along with copy-on-write and mem-mapping) it pretty much has to be able to address the same physical addresses via different virtual addresses associated with different address spaces (ie different processes), and that pretty much mandates a physically addressed cache. This, as a downside, in turn means that every L1 access has to first pass through the TLB, with implications for an additional cycle or so of latency, and some additional energy spent.

Now, suppose that we insisted that our new CPU was going to run a custom OS that does not care about these memory mapping games. In that case we could use a virtually addressed cache. What would be the consequences?
- we could avoid TLB lookups (latency and power savings)
- we would need a TLB lookup when making an L2 cache lookup, but L2 lookup is much more rare
- we need to have the virtual address stored in the cache tags for when we check if a hash-matched line matches the address we're looking up. But to which address space does this virtual address correspond?

The most basic way to do this would just be to have the address space implicitly be the current address space. This however would also mean that on every context switch we would have to flush the cache, which is obviously not ideal. Slightly more sophisticated would be to give each process an address space ID, which is stored alongside the virtual address in the tag for each cache line. This now allows the cache to hold lines from a few different address spaces simultaneously, with the lines of the different spaces being cast out (or not) depending on load/store and context-switch activity.

Which now has an obvious analogy with the new GPU unified SRAM design...


Programmable Task and Data Manager
210
Central controller (resource + task + thread manager)
222
Programmable Instruction Execution Client
224A
Programmable Instruction Execution Client
224E
Specialized Function Client
226A
Specialized Function Client
226C
Shader Core G-1
220A
E * instruction execution clients
Programmable Unified Shader Core 0
C * co-processor clients
Memory access request and data network
228
Unified Level-1 Data Cache (UL1C)
310
G - GPU Shader Cores
Memory system that includes one or more higher-level caches
230

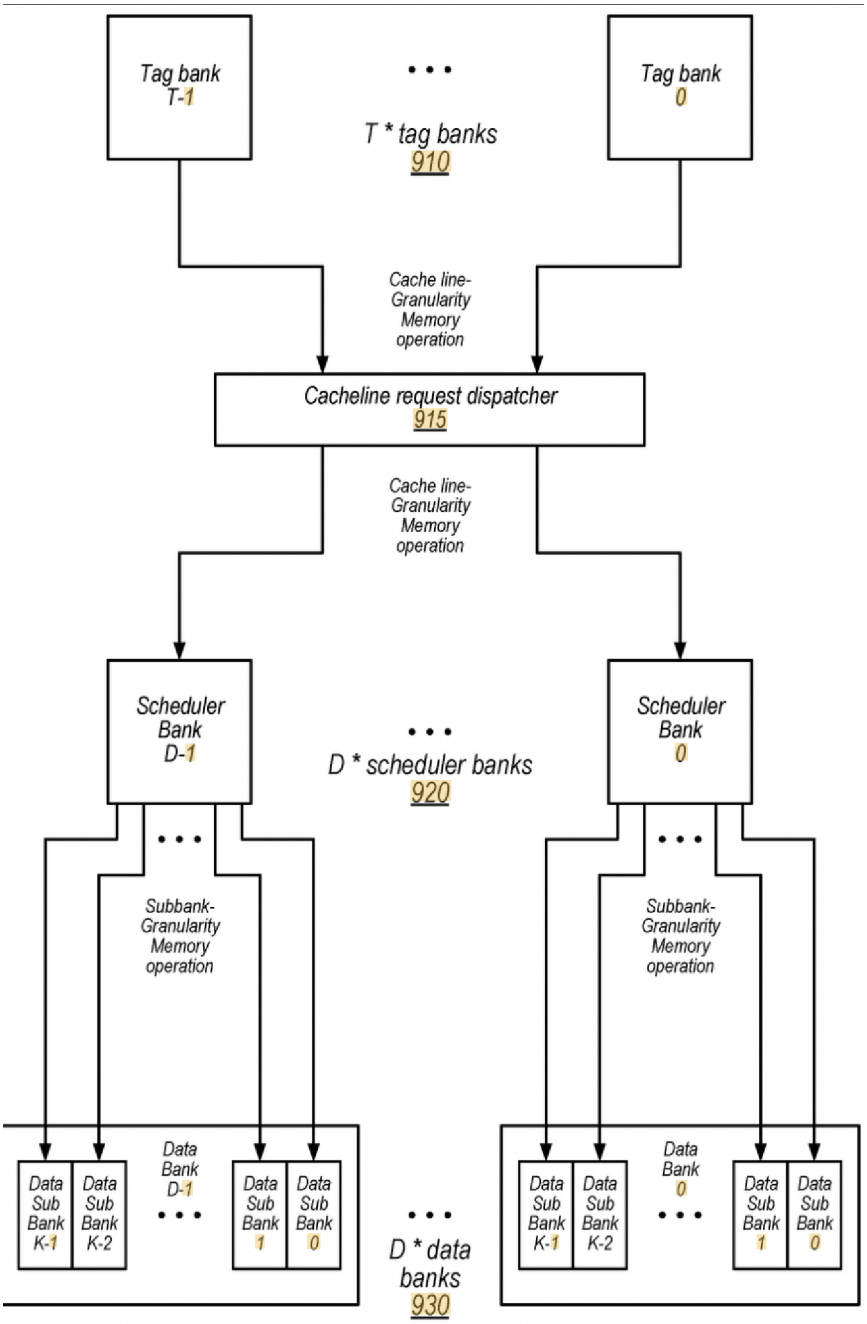
We still have (conceptually) multiple different address spaces. A register reference implicitly refers to an address in “register” address space, likewise a scratchpad load/store instruction lives in “scratchpad” address space, or a ray tracing instruction. But all these address spaces can live together in a single L1 cache, as long as

- by the time the load/store reaches the L1 cache it has been tagged with the appropriate address space ID
- all communication from the L1 cache up to the L2 cache goes through the equivalent of a TLB, mapping all these address spaces to a common “pseudo-physical” address space. (It’s unclear to me quite what Apple chooses. They could map to the genuine common physical address space. Or they could use an intermediate space that is mapped to real physical space when moving from the GPU L2 to the SLC. There are minor advantages to either choice.)

This in turn allows currently unused registers or scratchpad or whatever to be moved from L1 to L2, and then loaded back to L1 on-demand.

This provides the same advantages as we saw in the CPU case – most cache accesses will hit in the L1 cache and we can avoid the cost of the TLB lookup that’s mapping each address space into the common “pseudo-physical” address space.

This consolidation has obvious advantages in terms of more flexibly handling code with different demands, whether for more registers, or more scratchpad space, or more L1 space. On the other hand it also comes with some downsides. For example the consolidated cache needs to be able to provide the same bandwidth as all the previous caches (which might in principle have been referenced simultaneously by different parts of the GPU core). This requires aggressive banking of both the tags and the data, and unusual hashing, to ensure that as frequently as possible simultaneous requests are spread out across different tag and data banks. Thus we have a design like so:

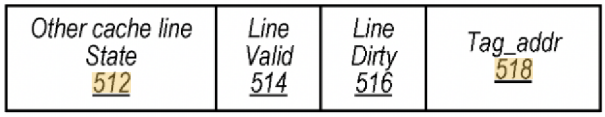

Tag bank T-1
Tag bank 0
T * tag banks 910
Cache line-Granularity Memory operation
Cacheline request dispatcher 915
Cache line-Granularity Memory operation
Scheduler Bank D-1
Scheduler Bank 0
D * scheduler banks 920
Subbank-Granularity Memory operation
Subbank-Granularity Memory operation
Data Bank D-1
Data Sub Bank K-1
Data Sub Bank K-2
Data Sub Bank 1
Data Sub Bank 0
Data Bank 0
Data Sub Bank K-1
Data Sub Bank K-2
Data Sub Bank 1
Data Sub Bank 0
D * data banks 930

allowing for multiple tag accesses (tag bank based on a hash of the address and the address space ID). The tag banks generate requests for the Scheduler banks, and the Scheduler banks (like a simplified DRAM scheduler) arbitrate and route requests to the data sub-banks based on availability. This somewhat unusual utilization of three levels of parallelization (multiple tags routing to multiple schedulers which fan out to multiple banks) gives us the desired degree of bandwidth together with the desired degree of "randomization" across banks.

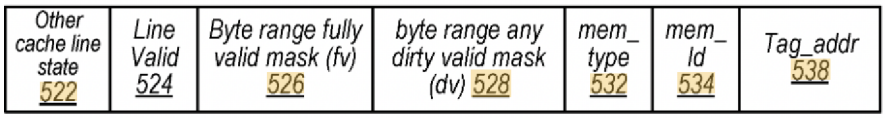
There's also some temporary storage (the equivalent of CPU MSHRs) to hold both cast out lines (modified being moved up to the L2) and to record load requests from the L2 that are in progress. Obviously in both these cases, you'd like to move such requests to the side so that the scheduler can get on with other requests while we wait for the L2. There's also some logic to deal with in-cache atomics, as we've already discussed in the *Atomic Smashing* patent.

We see additional unusual elements if we look at a cache tag.

Example traditional cache line tag state 510



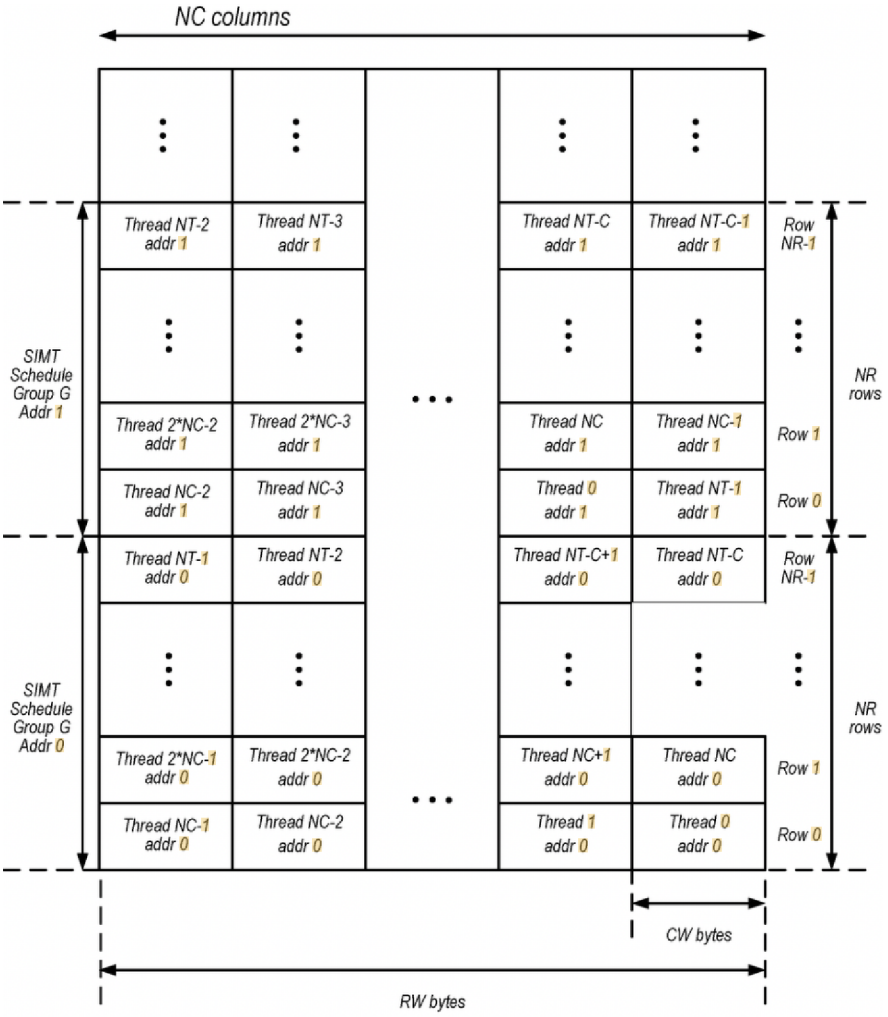
Example multi-memory-type sparse-access cacheline tag state 520



The mem_type and mem_id together are equivalent to what I've been calling an address space ID. But the more unusual element is that we have bitmaps that indicate which sub-elements of a line are valid and modified. The granularity of these bit maps varies depending on the memory type, for example register address space is tracked on a 4B granularity, whereas "general" L1 address space is tracked on an 8B granularity. You could imagine a few different uses for these, but the most obvious use case is in reducing bandwidth when writing back to the L2, though that's not discussed so I'm not sure quite how it's handled. You could also imagine, for example, if a register or scratchpad location is read that is marked as not valid (ie uninitialized) that might be treated as a error resulting in a GPU fault?

There are a few other higher level control features. These include some degree of quotas, so that no memory type can claim too much of the cache; and to flash invalidate all the memory corresponding to a particular address space ID. We have also seen some of this already, with the patent for locking cache lines corresponding to registers of kernels that are currently active.

One of the wilder elements is that hashing may try to distribute some elements by columns rather than by rows. This vertical hashing is another reason for the existence of the valid/modified bitmaps, in that a particular "unit", however you define it, may be valid but only along a column.
The example below shows a complicated 2D tiling of this sort, where registers are spread horizontally and vertically according to the threadID and the registerID. (Note how, for example, registers 0 and 1 for thread 0, likely to be accessed together, land up spread across both different columns and different rows.) The patent includes another such example of the same idea, this time for scratchpad memory.



Legend

RW = row width (bytes)
CW = column width (bytes)
NC = # columns per row = RW/CW
NT = # threads per SIMT per SIMT-schedule-group
NR = # rows per SIMT-schedule-group = NT/NC

Notes on example settings:

1. Set CLW = cacheline width (bytes) = RW
2. Set NBK = number of data banks = multiple of NR for SIMT-schedule-group parallel column access
3. Rotate columns for consecutive addresses for parallel row access (memory type dependent)

Of course there are a few fairly common non-sparse/non-random access patterns; like all the threads of a SIMD accessing the same register, or accessing successive L1 cache memory locations. To optimize these common cases, the cache also includes “data transposers” that can rearrange data on the way in or way out, optimizing cache bandwidth for these common cases. We see these below, also how the “upper” half of the cache concerns itself with addresses, while the “lower” half concerns itself with data:

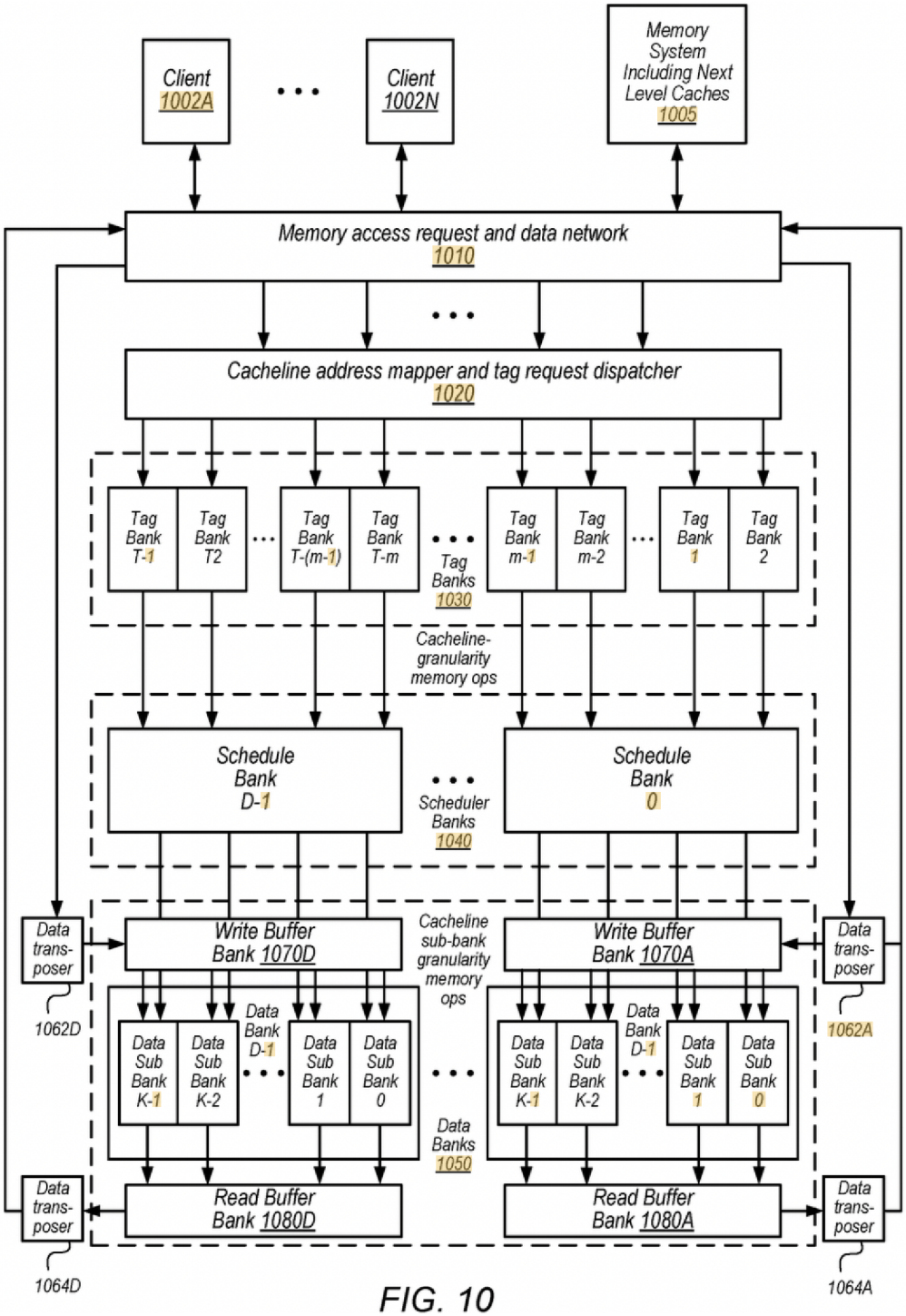
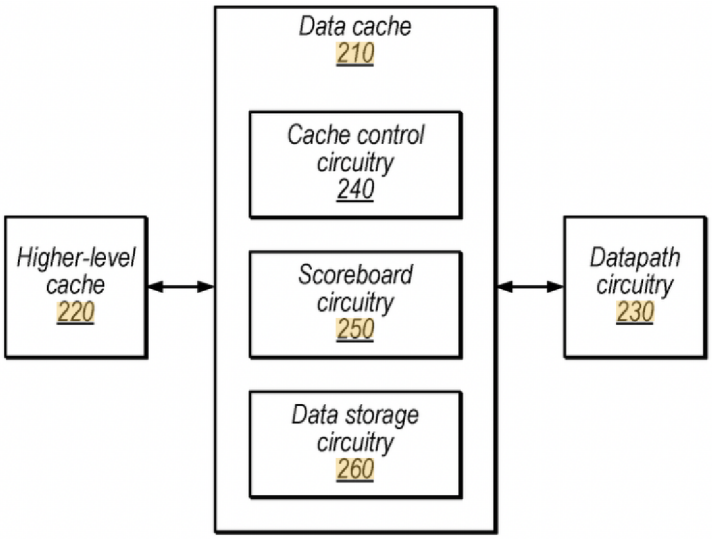

Client 1002A
Client 1002N
Memory System Including Next Level Caches 1005
Memory access request and data network 1010
Cacheline address mapper and tag request dispatcher 1020
Tag Bank T-1
Tag Bank T2
Tag Bank T-(m-1)
Tag Bank T-m
Tag Bank m-1
Tag Bank m-2
Tag Bank 1
Tag Bank 2
Tag Banks 1030
Cacheline-granularity memory ops
Schedule Bank D-1
Schedule Bank 0
Scheduler Banks 1040
Data trans-poser
1062D
Write Buffer Bank 1070D
Cacheline sub-bank granularity memory ops
Write Buffer Bank 1070A
Data trans-poser
1062A
Data Bank D-1
Data Sub Bank K-1
Data Sub Bank K-2
Data Sub Bank 1
Data Sub Bank 0
Data Banks 1050
Data trans-poser
1064D
Read Buffer Bank 1080D
Read Buffer Bank 1080A
Data trans-poser
1064A

FIG. 10

Structure of the register system

All this gets us to (2024) https://patents.google.com/patent/US20250103493A1 *Scoreboard for Register Data Cache*. Consider the flow of a dataflow instruction. The instruction specifies a register. Ideally the register is in the register cache (so the instruction can issue immediately); or it might be in the L1 cache (slight delay while instructions issue from the the other mini-front-ends); or it might even be in the L2 (maybe the relevant warp is put to sleep and a new one swapped into the mini-front-end).
That overall design is understandable, but how do we know the state of the registers at any time so that we can make this decision (issue vs pause vs swap out the warp). The answer is a structure called the Register Scoreboard that holds this information. Remember that although there are a lot of registers (32 actual storage locations for each "register" of a warp, for the most part we only need a single data item to indicate the status of all 32 registers. That is, either all 32 registers of the warp are present in the register cache or they are not. Or, if we're waiting for them to be loaded from the L2, the only state that matters is that they haven't all been loaded yet, ie a single bit covering 32 registers. This means that the register scoreboard can be fairly small even though your initial thought is that it's covering so so many registers. The register scoreboard storage is, in fact, flip flops (ie "fastest" storage) for the currently active warps, with "slower" SRAM storage for the state of the currently paused warps.

The system looks like



where the “Data Cache” is the register cache, associated with the datapath, and the Higher-level cache is the L1. Datapath instructions will hit the scoreboard circuitry described a register by [warpID, registerID], this will be looked up in the Scoreboard circuitry 250, and hopefully the request is fulfilled from the Data storage circuitry 260. If not, the request will be transformed into an L1-cache-relevant request (so the appropriate addressSpaceID will be constructed along with the appropriate address hash) and the request thrown out to 220.

I *think* Load instructions mostly bypass this. My understanding, but I could be wrong, is that they will set flags in the scoreboard, but the result, loaded from a cache or memory, will be stored in the L1 SRAM. I think we don’t execute load instructions directly into the register cache (ie storage 260) because that storage is limited, and we can’t afford to reserve a slot there indefinitely while a load possibly misses in cache and goes out to SLC or DRAM; it’s easier to reserve space in the L1, and transfer the loaded data into the L1. Then on first use of that loaded value, it will be transferred into storage 260 with a guaranteed short delay.

The fields of a scoreboard entry look something
like

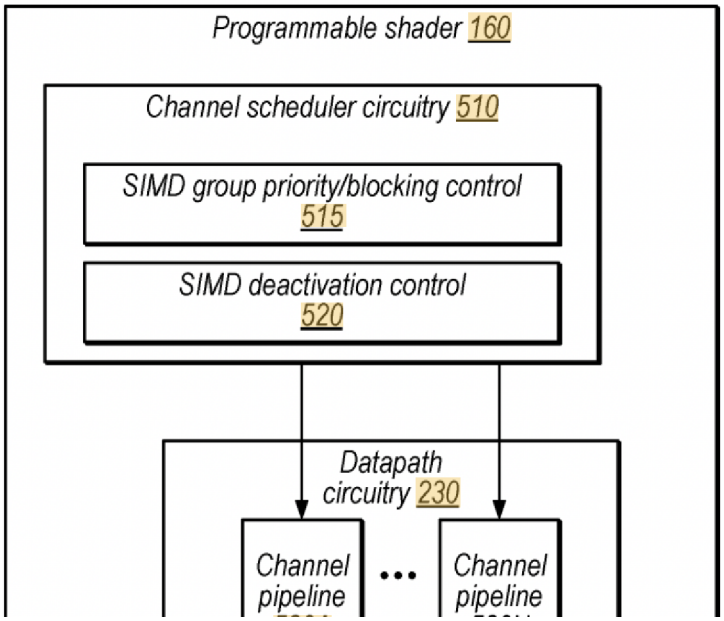
Map state 310	Initialized? 320	Pointer to storage entry 330	Long-latency operation pending? 340	Line-fill data return pending? 350	Evict pending? 360	Locked? 370

Map state means that a slot in the Register Cache has been allocated for the register, and Pointer 330 tells us where that allocation sits.
Initialized means that what's in the allocation is valid (eg there may be a window between mapping and data being loaded from the L1 during which the mapping is valid but not the data in the Register Cache).
Line-fill 350 seems to be about indicating that a register is about to be moved into the Register Cache from L1

Long Latency Operation 340 seems very similar, and I can't tell quite what the difference is. The obvious assumption is that it's the same idea, but slow operations in the Special Function Unit or in the Texture Unit. However the patent description is exceedingly confused on this point so??? Eviction pending, as you'd expect from the name, indicates the data is about to be written out to L1. (Maybe this is necessary to avoid race conditions, if at "the same time" an instruction is chosen that wants to make use of this register?)

Finally Locked is a performance mechanism to allow parts of the system to prevent eviction of a register from the operand cache. For example we've seen how GPU instructions can preload registers into an operand cache, and the locked bit could be applied to such registers. Alternatively you could imagine the compiler inserting such a lock instruction into the stream if it sees something about the upcoming set of instructions that might make this worth doing. This lock control is described in more detail in the patent below.

The rest of the patent is basically a reminder of the new GPU two level scheduling paradigm:



So from a large pool of executable warps, ie SIMD groups, (unclear what the number currently is for M4 generation, maybe 24..32?), at any given time we expect most of these warps to be “sleeping”, that is in a state of waiting for something slow, generally waiting for memory, but sometimes waiting for some ordering event (barrier or whatever) to happen.
So from this set of executable warps at any given time there are some number (say 6 to 8?) that are potentially executable, ie not waiting on a long latency event.
Block 510 handles this level of scheduling, including deciding which (say three or so from the 6..8 currently executable) will be running in the mini-front-ends (the “channels”) of this quadrant of the GPU core. The decision of whom to send to a channel might be made based on age, but also on knowing something about which type of execution block (eg datapath, load/store, texture unit, ray, etc) the code will dominate.

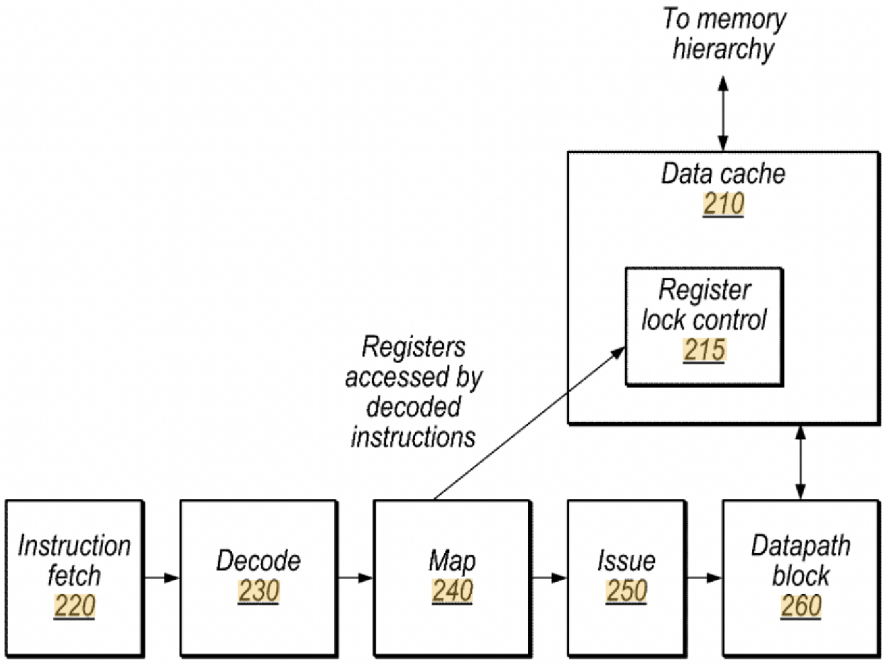
Then block 540 decides which instruction gets dispatched from the various pending channels into one of the execution pipelines, trying to avoid short delays (things like L0 instruction cache misses, registers not present in the operand cache, or instruction dependencies).

The relevance of this to the Register Scoreboard is that the scoreboard provides two levels of storage describing registers. I assume this is actually used more or less like a two level cache, with the fast level of Register Scoreboard storage describing the active warps (those present in channels, able to issue an instruction at any moment) and the slow level of Register Scoreboard storage describing the “pending” warps, the ones that are not waiting for a long-delay event, and so could be scheduled to a channel when that becomes necessary.
A natural way to structure this would be something like the number of entries in the fast storage of the Register Scoreboard more or less matches the number of entries available in the Operand Cache, but this is not explicitly stated. What is stated is that the fast Scoreboard entries are grouped into blocks allocated to each channel, along with (again per channel) entries that are allocated to warps that have just been deactivated from the channel (ie so that we can immediately switch to a new warp and get going, without having to wait for all the state associated with the previous warp to be cleaned up).
How about the size of the slower Scoreboard storage? That appears to be sized large enough to cover the registers present in L1. One interesting consequence of all this is that we may occasionally have registers that are present in the Scoreboard but mapped out to L2 rather than L1, and on register access the system should be able to send the appropriate request to L2 bypassing L1 and saving a few cycles.
However it’s hard to be certain of these sorts of details because different patents use different words to refer to the same piece of storage!

The Operand Cache is treated like a cache. In other words, suppose that Warp A is deactivated from a channel and Warp B is activated. In the background we might copy out the modified registers of Warp A, so that we can rapidly reallocate that storage if required, but otherwise we might keep Warp A’s registers present in the cache, just marked unlocked (and with the Scoreboard entry moved from the fast storage of the Register Scoreboard to the slower storage).

If we do need to allocate a new entry in the Operand Cache, it will be Warp A's registers that are first chosen to be overwritten, but there is some structure to the choice. In particular the highest numbered registers are chosen first. The compiler tries to use lowest numbered registers as much as possible, so hopefully this convention allows for retention of the most useful registers in the cache until Warp A returns to the channel.

(2024) https://patents.google.com/patent/US20250094357A1 *Cache Control to Preserve Register Data* describes the register cache locking scheme in more detail.



The diagram shows what we would expect. The Register Lock Control is accessed, like pretty much all the Register Scoreboard access, right after Decode.

The rest is policy details.
The current specifics include

- locking happens automatically, when a register is accessed
- bulk unlocking happens automatically when too many locked lines are present in the operand cache, or when the client of a particular set of registers has been stalled too long. (I assume the cache does the bulk unlocking internally when too many lines are present, and the core counts stall cycles for each warp and indicates to the cache when a warp has been stalled too long.)
- at least two lock bits are available for each scoreboard entry, so you can do things at various "transition points" like switching to a new active warp. For example at a transition point you can send a "bulk *un*lock" message to the cache, then start locking new registers from the new warp (using the alternate lock bit) while the bulk unlock, and related behavior (maybe moving modified registers out to the L1 storage) proceeds over the next few cycles.

Local result caching for FMA

Suppose you have a scalar CPU that's calculating a long dot product. Essentially the code looks like a loop of C=A×B+C, perhaps with A and B cycling through register 0, 1, 2, 3, ... and some other code sequentially loading these registers 0, 1, 2, 3.
So a consequence of this design is that we're continually reading register C from the register file, adding it into the multiply, then storing C back in the register file. That's not ideal! Now in any decent modern CPU, we'll in fact pull C off the bypass bus as it is being written back to the register file, so we can at least avoid the latency of re-reading from the register file. (It's still a shame we keep writing back the intermediate C result which is immediately overwritten, but that's a problem for later).

Switch to the GPU, and imagine we're similarly calculating a whole lot of long dot products in parallel. We'd have a similar code structure of C=A×B+C, only with A and B each 32 elements long and being pointwise multiplied, then accumulated into a 32-wide register C, to calculate 32 dot products in parallel.
Once again we have a situation where we waste time and energy moving the register C back and forth between the execution unit and the operand cache. Can we avoid this?
Suppose we provided a single register of storage attached to each lane of each execution unit (maybe the FP16, FP32 and INT pipelines all have their own separate storage per lane, maybe they share a common 32bit storage, that's a detail). Now we can define a new instruction that looks something like local=A×B+local, and avoid all that back and forth movement of C.

This is essentially what (2024) https://patents.google.com/patent/US20250103292A1 *Matrix Multiplier Caching* is about.
The more common task we care about is matrix multiplication, not exactly dot products, and you should recall that we the Apple GPU provides a matrix multiply instruction that handles the multiplication of something like two side by side 4×4 submatrices with a 4×4 submatrix to give two 4×4 results, achieved by packing these matrices in appropriate order into the 32 lanes of A and B registers, and then shuffling the values around as we pass the registers repeatedly through the FMAC execution units.
This is nice if we want to calculate a 4×4 matrix product, but usually we have larger matrices, so we have to repeat this operation. At which point (you can work through the algebra if you like, or can just imagine it in your head as operating like dot products, only with the "basic units" being 4×4 matrices

that are multiplied together and accumulated) we are back to our earlier dot product explanation. As long as we have one register of local storage available to each lane, we can keep accumulating our matrix multiplication while avoiding the latency and energy costs of frequent movement of the C values.

Interestingly this also avoids use of the third track of the operand cache into the execution units, which in turn means that we don't have to expand the bandwidth of the operand cache too much to get interesting consequences. Suppose the operand cache today can supply three operands per cycle, and suppose we boosted it to supply four operands per cycle. This would in turn allow us to supply two operands to each of two pipelines, meaning that, for example, we could now run two large matrix multiplies in parallel without as much expansion of the hardware as earlier seemed necessary... Of course this still requires two of the appropriate execution pipelines, but it mostly solves what looked like the hardest part of the problem of making GPU execution wider.

We see a related patent in (2024) https://patents.google.com/patent/US20250110734A1 *Operand Selection Circuitry*.

We have seen three different mechanisms available permuting data between the 32 lanes of a SIMD. From the very first days of an A-series GPU we've been able to move data between the four lanes that form a quad (ie lanes 0..3, lanes 4..7, and so on). This was present as part of the ability to calculate local gradients.

Then we saw that the addition of matrix multiplication instructions (performed via the existing GPU FMA units, not a separate tensor core) required a specific pattern of lane shuffling, as described in (2019) https://patents.google.com/patent/US11256518B2 *Datapath circuitry for math operations using SIMD pipelines*.

At the same time we also saw (2019) https://patents.google.com/patent/US11126439B2 *SIMD operand permutation with selection from among multiple registers*. This describes a different set of fixed permutes, as would be relevant if you want to slide a window a few pixels over relative to existing content, so the fundamental operation is essentially a left or right shift, treating the 32 lanes as a large SIMD register.

There are a few variants on this that handle dealing with edge cases, or "sub-rotations" (ie rotate within the two 16 lane halves of the full 32 lanes).

In terms of API we have that Metal 2.2 (2019) provides `simd_shuffle`() which *looks* like a generic permute, and is described as such in the MSL documents, but it is NOT a generic permute. If we look at Dougall's M1 GPU reference, https://dougallj.github.io/applegpu/docs.html, we see that `simd_shuffle`() only shuffles within a quad.

Metal 2.4 (2021) provides for `shuffle_and_fill_down`() and variants, which provide the cross-lane shifts I describe above, and which are present as of the A15 (released Sept 2021).

So to summarize:, in spite of misleading documents (which confused me) the actual permutes available on Apple GPU hardware are

- general within a quad
- variants on left and right cross-lane shifting
- what's necessary for the original matrix multiply.

Why do I stress all this? A general purpose permute network is very expensive in area. So if you can get away with simpler variants, you will do so! I wrote my earlier analysis of the matrix multiply under the

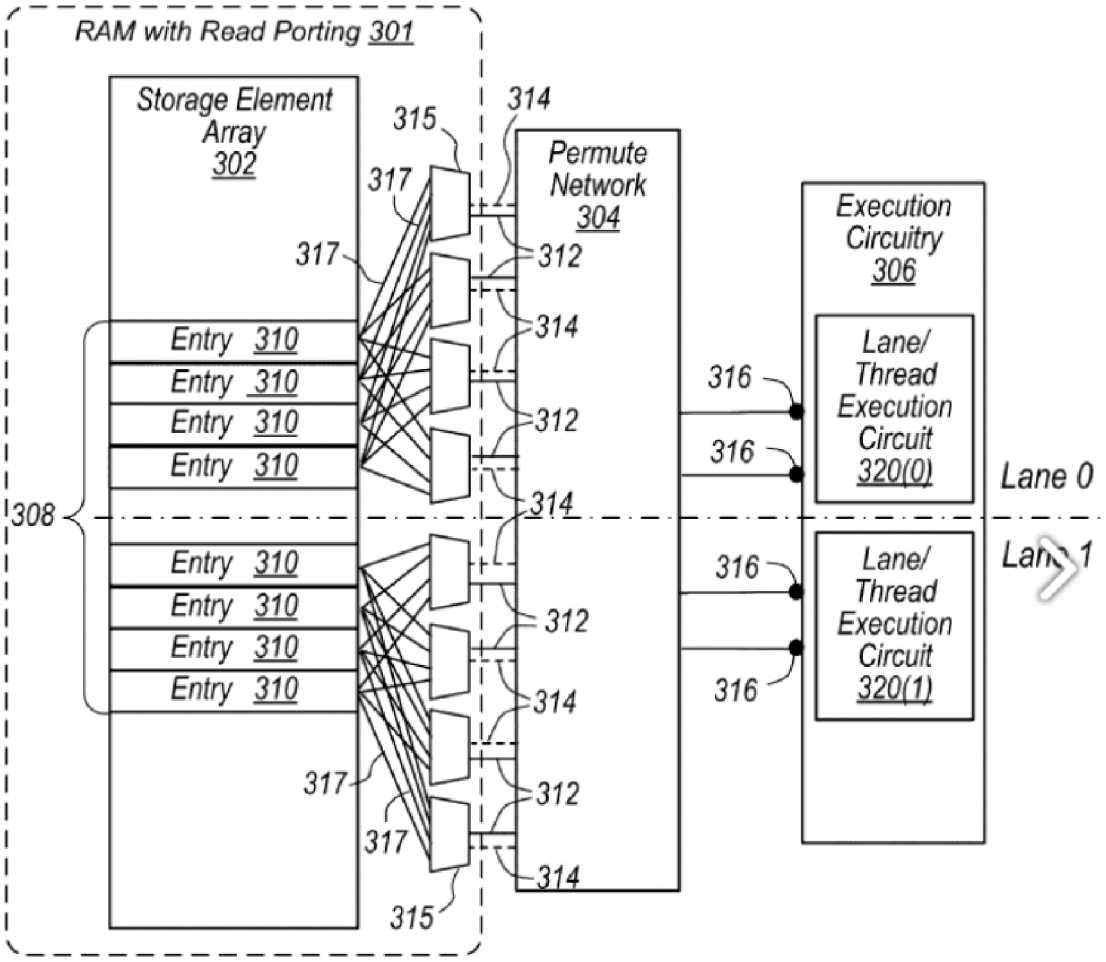
impression that a general purpose shuffle network existed (in which case it would make sense to fuse general permutes with the subsequent instruction) but in fact no such general permutes yet exist; what we have is a few specific permutes, implemented in different ways to solve different problems.

OK, that sucks, but that's life. Given this situation, suppose we wanted to grow the size of our basic multiplication unit. As I mentioned above, as best I can tell for the 2019 multiplication patent the basic unit is a pair of 4×4 matrices (filling one register) multiplied against a single 4×4 (filling half a register) to produce a pair of 4×4's (filling one register). This does the job and (I think) fully utilizes the FMA's but does not make optimal use of register cache bandwidth, which is our most restricted resource.

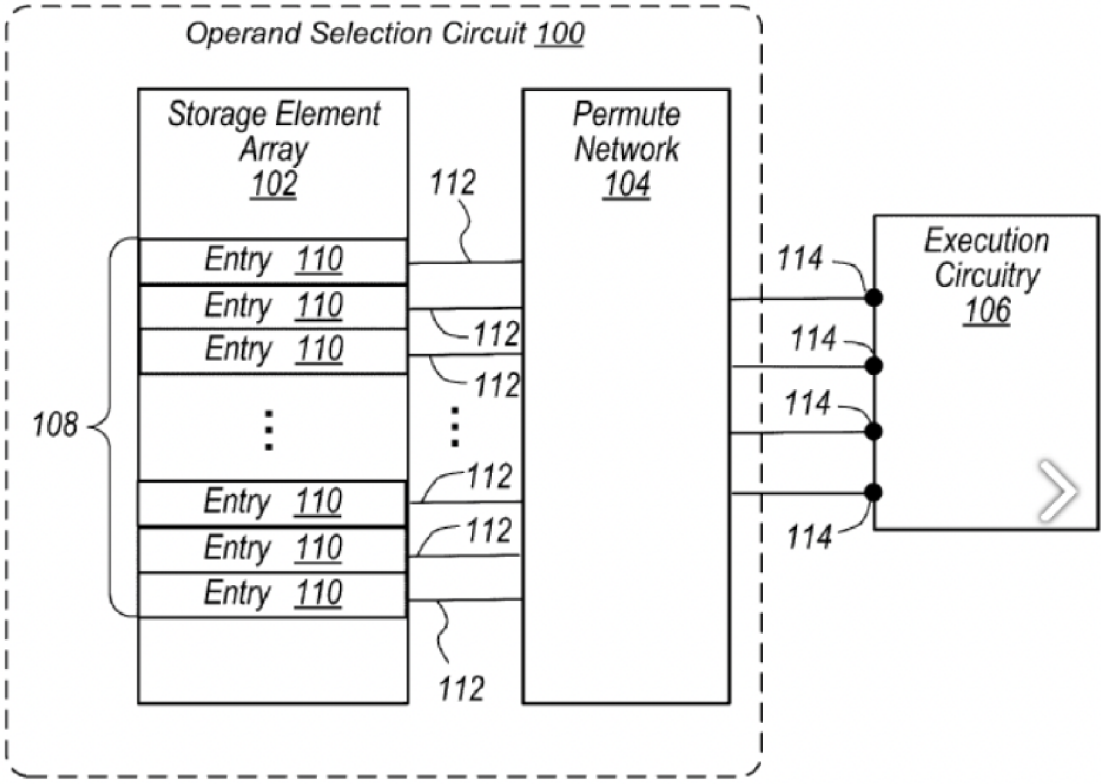
BTW the *Datapath circuitry for math operations using SIMD pipelines* patent pushes this a little further by exploiting the fact that a 32 lane 32bit register has FP16 high and low halves, so a single such register can in fact hold the 64 values of an 8×8 matrix. In that case we can still use our limited shuffle resources to create an appropriate layout within two single registers (one for each source operand) and again use a sequence of FP16 FMA's (16 in this case) to create a single 8×8 FP16 output matrix. But I think it's easiest to think of this in terms of FP32 registers and their constraints, understanding that we can probably double *something* when using FP16 to get twice the effective storage or throughput.

Suppose we grow the fundamental unit to multiplying an 8×8 matrix (64 elements, spread over two vectors) against an 8×8 matrix (again two vectors) to create an 8×8 matrix (again two output vectors). This gives us better register cache bandwidth utilization, especially if we can repeatedly gather intermediate results in local storage. But it requires a different way of packing the matrix values into the registers, and thus a different shuffle network. I *think* I'm correct in saying that all of the work required in handling the 4×4 matrix multiplication was some combination of (in successive cycles) shifting lanes by 4 units and perhaps shuffling within a quad. So building on existing shuffle hardware.
But we need a little more if we want to manipulate rows or columns that span 8 lanes.

What this new *Operand Selection* patent notices is that, first, we can still perform the required permutes using a restricted permute network, by slightly augmenting the existing permute network, and secondly, that we can move part of the permute network into the SRAM storage sitting between the actual SRAM (the operand cache) and where we read the SRAM results. The basic idea is below:
We allow some (say four) storage elements to have four separate hardwired paths to a mux (the SRAM read port for this register), and in that mux select which of the four values we actually want (as the value "read" from the SRAM into the permute network.


RAM with Read Porting 301
Storage Element Array 302
Entry 310
Entry 310
Entry 310
Entry 310
308
Entry 310
Entry 310
Entry 310
Entry 310
315
317
317
317
317
315
314
Permute Network 304
312
314
312
314
312
314
312
314
316
316
316
316
Execution Circuitry 306
Lane/ Thread Execution Circuit 320(0)
Lane/ Thread Execution Circuit 320(1)
Lane 0
Lane 1

Now there is some strangeness possible at this point!
What are the “storage elements” that we hook up to the muxes that feed into the Permute Network?
Look at the basic version that does not use a fancy SRAM:
In this design presumably 32 lanes go in along each channel 112, get permuted, and come out as 32 lanes 114. So the entries 110 are the 32 lane values for his register.


Operand Selection Circuit 100
Storage Element Array 102
Permute Network 104
Execution Circuitry 106
Entry 110
Entry 110
Entry 110
Entry 110
Entry 110
Entry 110
108
112
112
112
112
112
112
114
114
114
114

The simplest version of our new design would have that the first four entries are lanes 0..3, so the shuffling that happens within the mux gives intra-quad shuffling, we send a shuffled version of lanes 0..3 into Permute Network 304, and the permute network rearranges as it wish, but having avoided the work of the within-quad shuffle.
That might save us a little energy or area, but it doesn't give us anything fundamentally new.
However if the first four entries in the SRAM are, say, 0, 4, 8, 12, now we have random permutation available within these, followed by the previous shuffle network. So we have all the permute capability we had before, plus *some* degree of random mixing between four adjacent quads.
(The reason for the two lines 312 and 314 into the permute network 304 is that we may want to read a value twice from this particular mux of the Storage Array Element. This is more than just a shuffle, which never replicates values, but less than a permute, which allows possible total replication of a single value.)

So this is the basic big idea – provide some limited shuffling ability within the Operand Cache SRAM, which can be coupled to the limited shuffling ability outside the Operand Cache SRAM, to give us a slightly more powerful set of data re-arrangements (including a limited set of permutes that are not just shuffles).
It's unclear quite how far Apple is pushing this idea in the first implementation. The examples given in the patent do not seem to suggest support for a larger basic block for the purposes of matrix multiplication, but that may have been to keep the diagrams simple, giving only 4×4 examples.

In principle, you can do even crazier things with this design, though Apple does not mention these.
For example what if you laid out the operand storage so that two registers had interleaved entries in this storage element array?
Now using this same basic idea, you could start to do things like zips, that interleave or otherwise (in some restricted way) shuffle the values of two registers together. This in turn is a useful primitive for the transpose of large arrays.
Of course the extent to which this is sensible depends on how cheap it is to lay down these extra lines from each source storage element to each read port...

Finally let's get back to the *Matrix Multiplier Caching* patent; there's one interesting side thing they mention. The patent includes this diagram

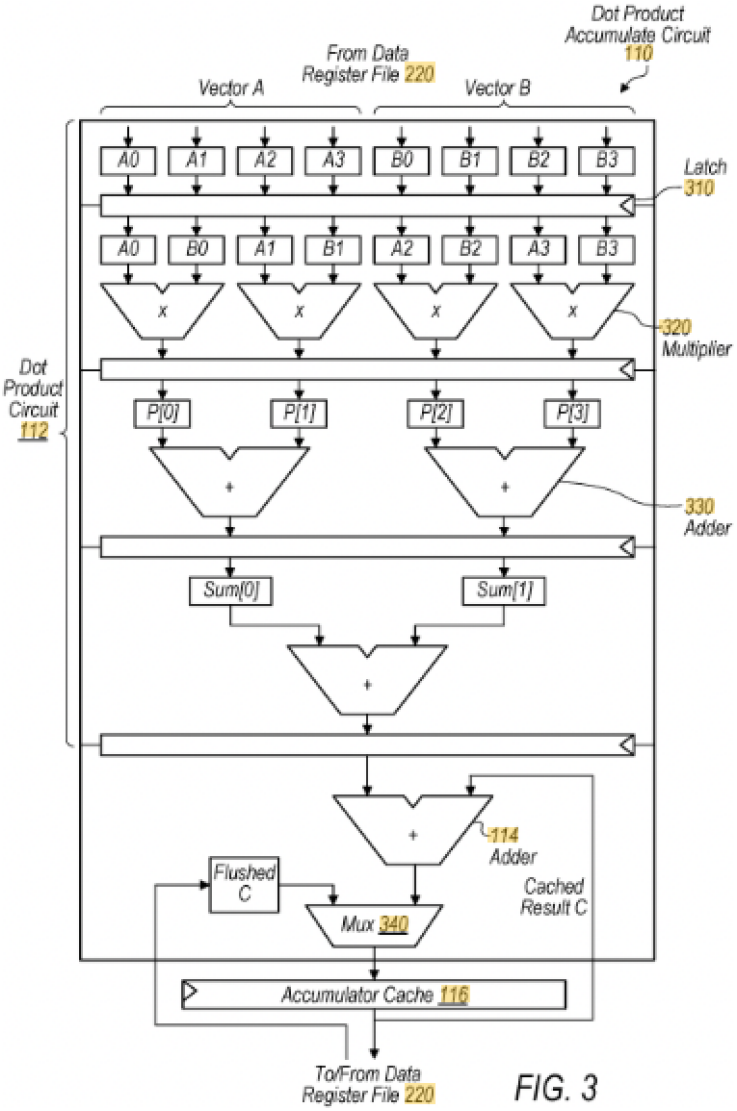

Dot Product Accumulate Circuit 110
From Data Register File 220
Vector A
Vector B
A0
A1
A2
A3
B0
B1
B2
B3
Latch 310
A0
B0
A1
B1
A2
B2
A3
B3
x
320 Multiplier
Dot Product Circuit 112
P[0]
P[1]
P[2]
P[3]
+
330 Adder
Sum[0]
Sum[1]
114 Adder
Flushed C
Cached Result C
Mux 340
Accumulator Cache 116
To/From Data Register File 220

FIG. 3

This is not how you'd calculate a dot product using normal FMA's, because the basic units are not FMA's, they are distinct Adders and Multipliers, and the operation is performed "width-wise" across four lanes, rather than lane by lane. So what's going on? Here's my best guess.

First note that if we stick to matrix multiplication, it doesn't help because for that we need actual FMA's, not separate multipliers and adders.
Second this design does give us "single cycle" (throughput, not latency) dot product for 4-long vectors, but at the cost of "disaggregated" FMAs. So, for long dot products, or many 4-wide dot products, it's no faster than an FMA design, but for short dot products we do get the results faster.

So maybe the real point is that
- the FMAs *were* (in some sense) disaggregated
- some minor cross-lane data movement was provided
so that 4 FMA units can ALSO be used as four ("multiply, then add") units following the pattern of the diagram?

Adding additional adders to the existing FMAs seems a terrible idea for such a niche use case, but maybe adding some additional wiring +buffering (and given that GPU timing is not very aggressive) was quite feasible? So you get a faster dot product for the cases that care (presumably a lot of graphics) at close to free?

By disaggregation I mean that
- a traditional FMA unit is not just a multiplier and adder close to each other, it's also a particular data flow between the two so that you can achieve the result (A×B+C) at lower latency (the multiple results flow into the adder) and at higher precision (the intermediate results don't have to be reduced to the format of FP16, FP32 or FP64 to be stored in an intermediate register). But if we're not obsesses with the high frequencies of a CPU, we can add some buffering between the multiplier and the adder so that we can do some alternate timing or routing of the results, and if we're not obsessed with IEEE accuracy, we can even also throw away a few of intermediate precision bits to reduce the size of these buffers and routing wires.

I could imagine a thinking that starts with "let's add a register to each FMA lane to avoid data movement costs" then, after doing this realizes "hell we could also add a few buffers with very minimal area and no cycle hit. hmm?" which flows into "hey, with a few extra buffers we could rearrange data flow between our adders and multipliers to speed up dot product!"

There is an alternative analysis that claims these patents suggest not what I am describing (gradual modifications to existing FMA pipes and multiply machinery) but substantial modification in the form of a separate "tensor core" added to the GPU. I guess we'll see over the next year or two. Meanwhile you, dear reader, can look at the patents yourself and decide!

Wider execution

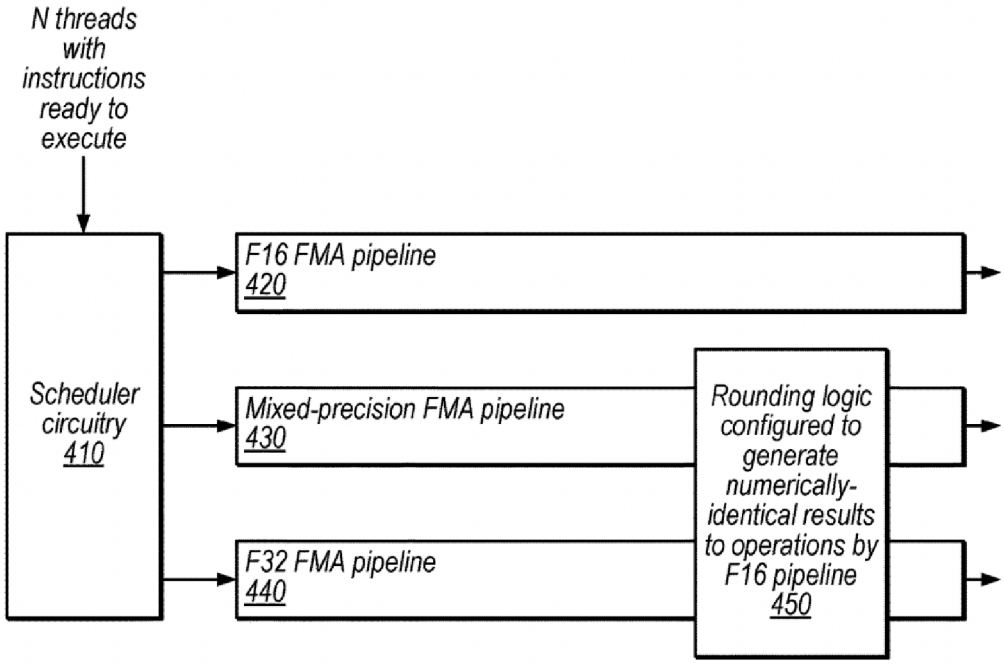
Suppose you want to increase your GPU compute, but not to spend much area, so you can't increase the number of cores. What are your options? The most obvious choice, as we have already discussed, is

to go superscalar to 2-wide, so that more of your execution units can be active every cycle. Apple's first pass at this, presumably just getting infrastructure in place, had a disappointing performance boost because while three execution pipes are available (FP32, FP16, and INT) INT is rarely used, so its dual issue doesn't help much, and it's rare to have either a single warp that uses both FP16 and FP32, or two independent warps that use these two different types simultaneously.
Other GPUs have either modified their pipelines so that the same pipeline can handle both FP16 and FP32, or even more ambitiously, so that the same pipeline can handle FP32 and dual FP16 operation. Both of these make sense. An FP32 unit has to have a multiplier and an adder/shifter, and running those slightly less wide gives you essentially FP16; while you need to add a little more logic to get dual FP16 (ie an FP16 pair) but you can transport the data into/out of the execution unit along the existing 32-bit wide path to the operand cache, so you're making better use of the operand cache, which is one of your tightest resources.

Apple has so far avoided this path, saying in patents that to save power they wanted the FP16 and FP32 pipes to be maximally energy efficient with no extra overhead.
This changes with (2022) https://patents.google.com/patent/US12405803B1 *Superscalar execution using pipelines that support different precisions*, which appears to be one of the features we know is present in the A19, and presumably coming in the M5.

The patent discusses two main features.
The first is how the precise rounding to FP16 is performed. This seems pretty obvious and it's unclear why they mention it; I'll get to that a little later.
The second feature is that the instruction scheduling is modified to preferentially steer instructions to their optimal pipeline, specifically steering FP16 to the FP16 pipeline where possible, so limiting the excess power draw.

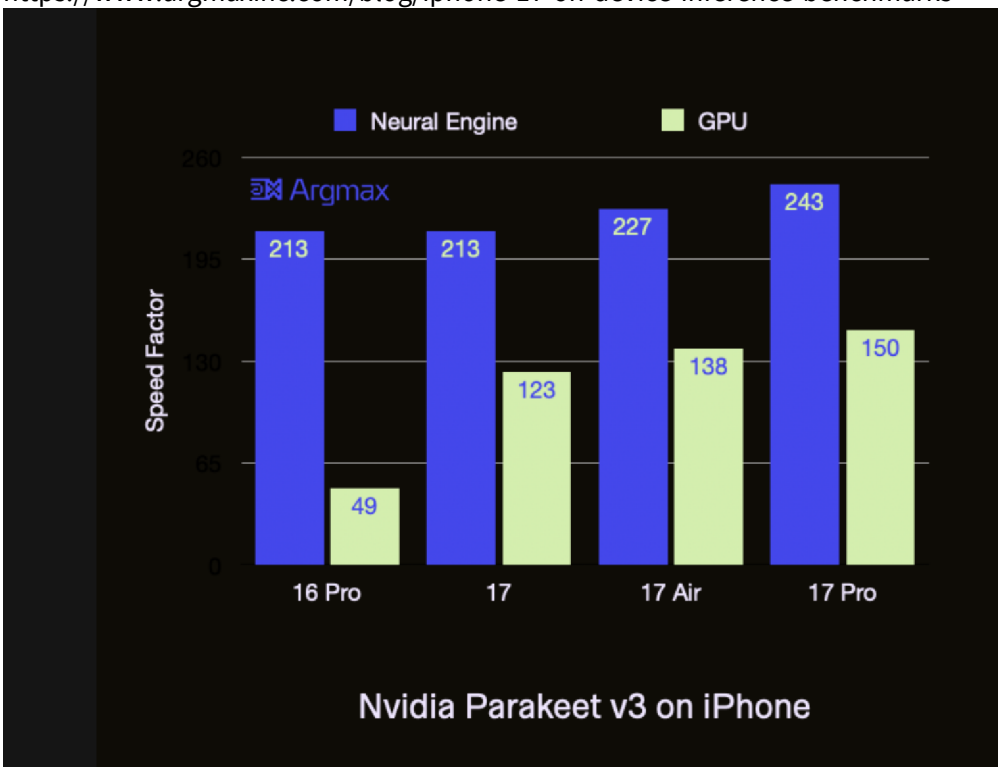
So the above are, in a sense the obvious elements of the patent. There's one less obvious element. The patent speaks of each SIMD possessing not two but three pipelines, for FP16, FP32, and "mixed precision" meaning FP16 and FP32. Essentially we seem to have pipelines specialized for
- FP16 only,
- FP32 only, and
- FP32=FP16+FP16*FP16 and FP32=FP32+FP16*FP16 (so FP16 multiplies, but accumulating to FP32)



So it seems that in fact
- the amount of logic has increased (the additional, mixed, pipeline added)
- under different circumstances (wider instruction issue, or specialized instructions) not two but **three** FP16 operations might be possible per cycle.

The A19 presentation suggested that, for LLM purposes, the A19 is 3× as fast as the A18 in executing mostly matrix multiples in FP16. This seems feasible if we go from having one FP16 pipeline available to three pipelines available; and while generic FP16 code cannot access all three simultaneously, the specific matrix multiply instructions (making use of the local per FMA register we discussed in the

previous patent) can use all pipelines simultaneously? If, for example, the register cache can deliver six operands per cycle, this could work out as usually two groups of three operands (A×B+C) per cycle, but three groups of two operands (A×B+local) for matrix multiply.
Tests of FP16 AI code on the GPU bear out this 3x improvement:
https://www.argmaxinc.com/blog/iphone-17-on-device-inference-benchmarks



Neural Accelerators

The A19 presentation also talked about “Neural Accelerators” added to the GPU, and these providing 4x the “OPs” (whatever that might refer to) per cycle. To understand exactly what I think Apple has

done, let's start with some history.

history of nVidia's tensor cores

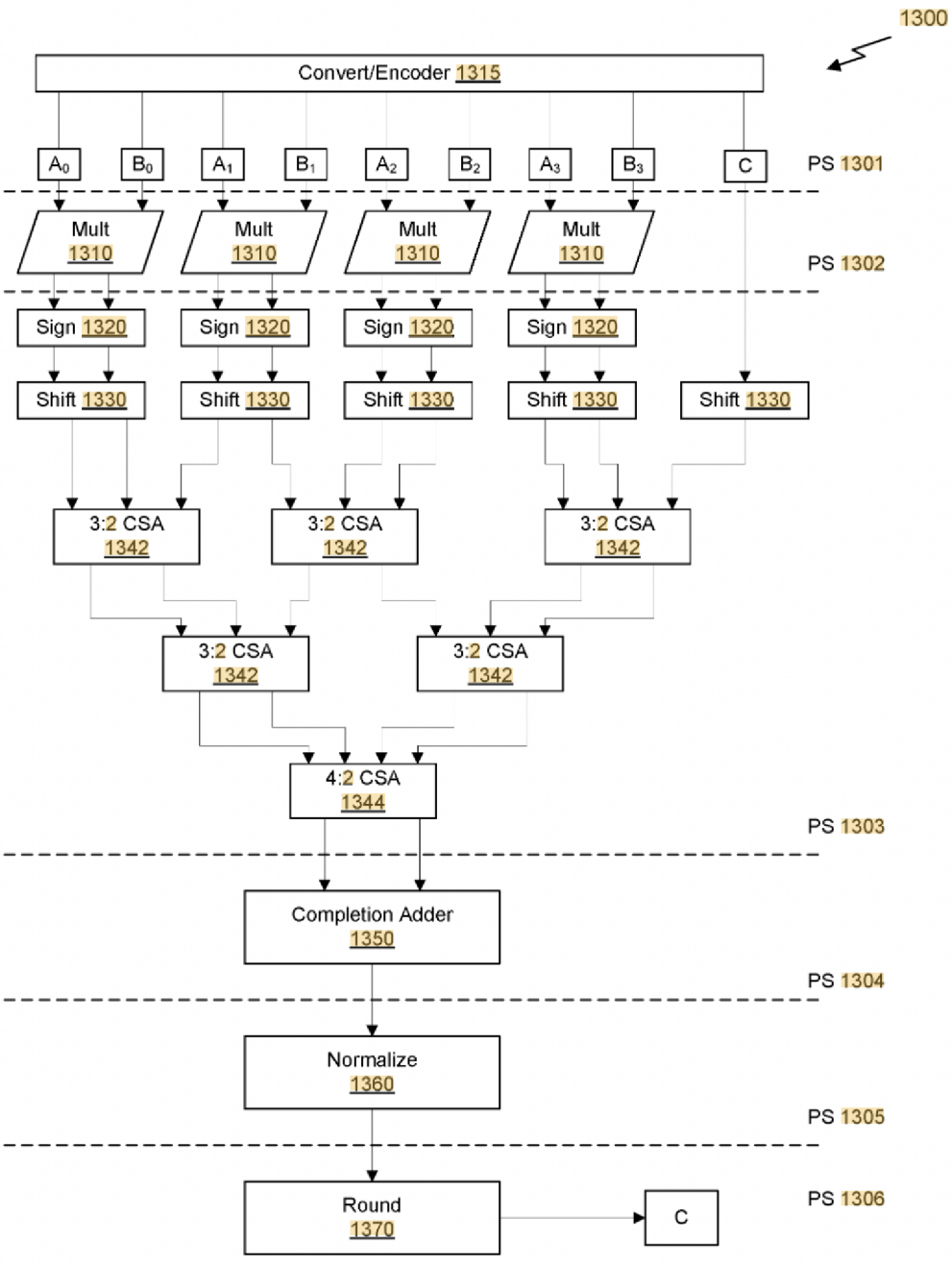
Recall that Apple's "standard" matrix multiply setup takes in two 4×4 matrices (ie 32 elements). Each element of a resultant 4×4 matrix consists of a 4-element dot product, so calculating the two output 4×4 matrices requires a four cycle dot product, with the same input registers each cycle, but permuted each cycle. This is our baseline. In principle (as described above, and if other elements, like the permute network, allow it) this could be three times as fast on the A19, for FP16.

Now let's compare with nVidia.
We start at the beginning, with Volta in 2017.
To quote from https://semianalysis.com/2025/06/23/nvidia-tensor-core-evolution-from-volta-to-blackwell/
"An SM of a Tesla V100 GPU contains 8 Tensor Cores, grouped in partitions of two. Each Tensor Core is capable of computing an equivalent of 4x4x4 matrix multiplication per cycle".
Let's go through this.
Each quad of an SM (conceptually a 32 wide SIMD) has a pair of 4x4 matrix multipliers, which matches 32 FMAs, and matches Apple.
And requires two pairs of inputs each 4*4=16 values wide = two 32 wide input registers.
So far so good, again matches Apple.
BUT
if they were performing this via say dot product, they'd need to iterate this over 4 cycles, repermuting the data each cycle. That's what Apple does.

But nV claim they do not need this, they can execute the whole thing in one cycle. More precisely they deliver a pair of 4×4 multiplied matrices (at this stage only FP16 matrices) every cycle, not every four cycles.

How do we resolve this?
An nV tensor core is NOT just the existing nVidia FMA hardware (although it may be based on it).
Rather it's something like four sets of this HW (specialized to only do the FMA part, dropping everything else) and some internal register latches.
One way you could imagine this is exactly four sets of this HW, so every cycle two 32 wide input registers corresponding to two 4×4 input matrices get delivered to the engine, but get latched by a rotating modulo 4 version of the FMA HW, and over the four cycles until the next inputs are latched, the 4×4×4 matrix multiply is performed.
This is not quite right. The patent (2017) https://patents.google.com/patent/US10338919B2 *Generalized acceleration of matrix multiply accumulate operations* shows how they did it back then.
A single 4×4 matrix multiplier consists of 16 of these things below, each of which calculates one of the 16 dot products that go into the output. So to first approximation it is 4× the hardware as we suggested above, but rearranged slightly so that some hardware (like Normalize and Round) doesn't need to be

replicated as much.


1300
Convert/Encoder 1315
A0
B0
A1
B1
A2
B2
A3
B3
C
PS 1301
Mult 1310
Mult 1310
Mult 1310
Mult 1310
PS 1302
Sign 1320
Sign 1320
Sign 1320
Sign 1320
Shift 1330
Shift 1330
Shift 1330
Shift 1330
Shift 1330
3:2 CSA 1342
3:2 CSA 1342
3:2 CSA 1342
3:2 CSA 1342
3:2 CSA 1342
4:2 CSA 1344
PS 1303
Completion Adder 1350
PS 1304
Normalize 1360
PS 1305
Round 1370
C
PS 1306

Later it becomes unclear (and unimportant) whether nV retain this basic design. They could, in principle, switch to something like an outer product based design if that saved a little energy or a little area; what really matters is the count of multipliers.

This is, in a sense, an astonishing claim – with Volta nV have, in each tensor core, 4x the FP HW of their mainline data path! It's presumably only viable because *everything else* in the tensor core has been dropped -- all the instruction control flow, various SRAMs, FP32 and integer support, special function unit HW, etc etc.
Which alternately suggests that the actually FP16 pipeline was not *that* large a component of an SM quad, small enough that adding a quadruple of that pure datapath HW was feasible.

So at this stage, nV Volta has about 4× the "IPC" advantage of Apple A18, in some handwaving sense, given that each SM quadrant has 4× the hardware thrown at FP16 matrix multiply.
Compared to Apple A19, this 2017 Volta still has about $\frac{4}{3}$ the "IPC" advantage (maybe?)

After Volta we then have
- with Turing, 2018, we add INT4 and INT8 support to the "tensor core". I think this is feasible and "easy" given my model for what nV has done. The constraint is the connection to the outside world, the delivery every cycle of those two input registers packing 32 values.
If those values are now, say, INT8, the obvious presentation is something like a baseline 8×8×8 matrix multiply. If we treat each FP16 FMA as also incorporating (shared HW or side by side) two INT8×INT8 multipliers, this means we could handle the matrix multiplication over 8 cycles (not 4), so INT8 runs both at 2x the FP16 multiply rate, but also at 1/2 the absolute max INT8 rate possible if hardware density were no object (and we provided 4 INT8 multipliers, not 2).
Likewise we seem to provide four INT4×INT4 multipliers, given a further 2x boost over INT8, but also likewise running at 1/4th the rate if HW density were no limit.

- before Ampere, 2020, data had to flow from the memory system through the register file to shared memory, then from shared memory to the register file to the tensor core. This obviously costs some energy, and it also suggests to that, regardless of other constraints, you can't get much useful simultaneous work of the tensor core and the rest of the GPU because the rest of the system is each cycle (more or less) engaged in all this moving of data to where the tensor core can access it.
With Ampere some of this is fixed, so we get a path straight from memory to shared memory bypassing some caching and the register files.
But the flow per cycle is still shared memory to register file to tensor core.

Ampere also adds structured sparsity support (nV's scheme for allowing half the values in each input matrix to be zeros, saving bandwidth and compute). However the general consensus as of 2025 seems to be that structured sparsity is a dud, even for inference.
This may be a general issue with LLMs. While ANE offers more flexible sparsity than NV, and this sparsity seems to work well for CNNs (like older vision models), it appears not to be used by Apple's 2026 Foundation Models, their large (by the standards of phones!) language+vision models targeting ANE.

It seems like everyone has concluded that quantization to 4b, for example, is more flexible and gives better results, than use of 8b and some sort of sparsity.
However this is an interesting paper, (2025) https://openreview.net/pdf?id=6lC3MPFbVg *WhisperKit: On-device Real-time ASR with Billion-Scale Transformers* which suggests a possible path forward. The essence of their scheme is to split weights into "common" weights, which are quantized to eg 4bits, and outliers, which are compressed using Apple's sparsity scheme (bitmap of zero vs nonzero). This seems to give almost all of what one might want, though given that the sparsity map is now only ~1% non-zero bits, I still think Apple should offer a run length compression option to make the bitmap a whole lot smaller...

- with Hopper, 2022, the tensor core can now read data directly from shared memory!
(In many ways this looks like the tensor core picking up functionality from the ANE, in that it can autonomously loop over n-D data structures and perform simple data movement, without requiring the help of HW that's better used doing more sophisticated data movement).
This at least opens up the hope of doing other useful work simultaneously with matrix multiply.
We also add FP8 support.
We also add the "Transformer Engine", which stripping out the implementation details, is basically support for block FP8, which can be thought of treating a block of data as FP8 along with a common multiplier that scales up (or down) the value of each element in the block.
Just FYI, also somewhere around Hopper nV also shrink the accumulator in the tensor cores for FP16×FP16 multiple down to accumulating in FP24 rather than FP32.

- with Blackwell, 2024, we replace the shared memory path with a new block of SRAM called tensor memory that only supports specialized access patterns appropriate to the various matrix multiplies, not a generic SRAM that can be accessed arbitrarily. We also drop the ability to use GPU registers as input to the matrix multipliers (still present in Hopper). So now the tensor core really feels like an autonomous engine that can be pointed at some data and unleashed, while the rest of the GPU can do whatever vector work it likes until it needs to synchronize with the Tensor Core either to read values or provide new input data.
We also get microscaling (block FP acting on smaller blocks) and MXFP4 and MXFP6 which are 4 and 6b FP formats.

The big summary is
- the Tensor Core is real, *separate* HW within each nV SM that, each upgrade becomes more autonomous (starts to look more like ANE in various ways) along with picking up more HW (eg INT8 and INT4 then ever more numerical formats)

- At the Volta vs A18 level, nV had about a 4x IPC advantage
- At the Volta vs A19 level, nV has about a 1.33x IPC advantage
- At the Turing level nV picks up an additional 2x advantage if you care about INT8, or 4x if you care about INT4, which later translates to a 2x FP8 IPC advantage

- at the Hopper level, nV picks up an additional “autonomous execution” advantage in that the rest of the GPU can do quite a lot while the tensor core grinds away, using its HW to pull data into its SRAM, largely decoupled from the rest of the GPU.

Apple’s response

To this gets us to the Neural Accelerators. What are they?
Look at (2024) https://patents.google.com/patent/US12405786B1 *Hardware support for conversion between integer and floating-point data*. At first sight this looks like a not-especially interesting, very technical patent.

Two elements are described
1. hardware (maybe added to each FP pipeline, maybe added only to the new “mixed” FP pipeline) that takes in integers (eg 8 bit, but quantized, so 6 or 4b are also feasible), and converts them to FP, along with optional dequantization and zero shift (in others words fpOut=intIn×fpSxale+fpOffset); and that can do the reverse, to transform FP into int subject to a quantizer and a zero offset. So this looks somewhat like what nVidia calls the Transformer engine and microscaling, a block scaled number format matching what we see on the ANE, and in Apple’s case unlike nV, structured around INT8s or INT6s or INT4s carrying the data, rather than nVidia’s FP8, FP6, and FP4.

2. an **integer** matrix multiplier that takes values from the register file and outputs integers! This is the big discovery, but it’s barely mentioned in the patent (presumably because, by itself, it’s hardly patentable!)

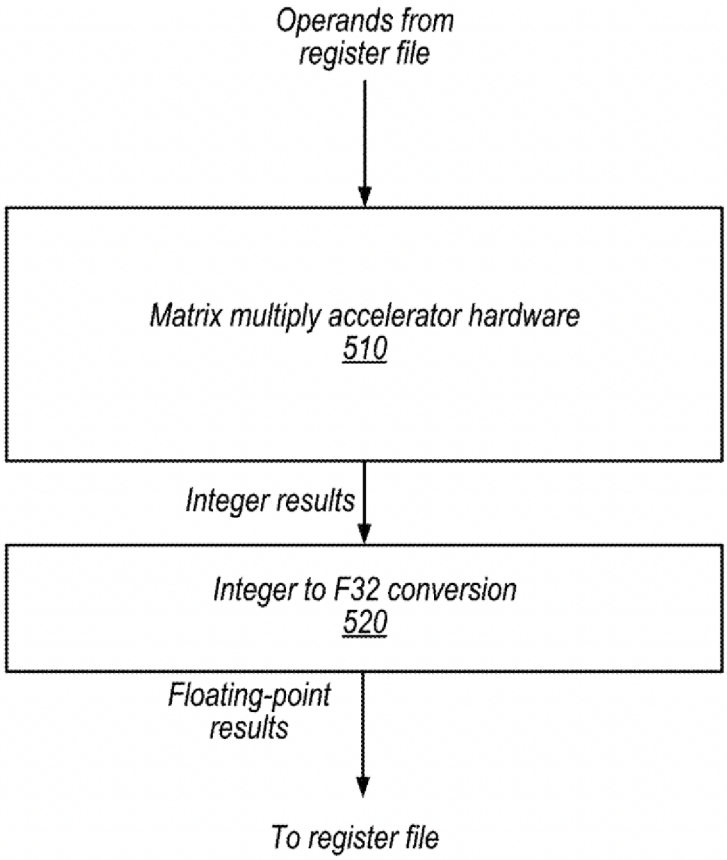

Operands from register file
Matrix multiply accelerator hardware 510
Integer results
Integer to F32 conversion 520
Floating-point results
To register file

FIG. 5

Putting these two together, the essence seems to be recreation of something like a Turing (ie 2nd gen) nV tensor core, a matrix multiplication optimized piece of hardware that's distinct from the rest of the GPU datapath.
Differences from Turing include
- only integer support (so substantially smaller!)
- block-based scaling and offset are already part of the system (so in this respect they're matching what nV has as of around Hopper)
- some additional HW (looks like it's in an FP pipeline, maybe that new "mixed" pipeline) to allow rapid conversion between FP data and this block scaled integer data.

- presumably if Apple like this direction they can head down the nV path fairly rapidly, including
- adding more HW (so one engine can perform more INT8 or INT4 multiply-add's per cycle) AND,
- the big next leap forward, making the engine more autonomous from the rest of the GPU, so that it can load/store data and loop over large matrices, in parallel with the rest of GPU compute.
It seems feasible that they could get to this point, matching much of Blackwell, as early as next year if that's a priority.

This, together with the earlier patent, may look somewhat chaotic! They're providing both extra FP16 AND extra INT8/INT4 (even as they run Apple's own AFM models on the ANE, not even using the GPU)! My best guess is that when the HW was designed (and remember, this is say 3 to 4 yrs ago) they were operating on ignorance of the future, and on rumors about nV (so guesses about what was in Hopper). GPT3 was known, but ChatGPT, its "consumer-facing" version wasn't yet released until mid 2022. Given this uncertainty the safest path was probably to provide what they hoped might be useful along every dimension!
- retain ANE for its advantages (a lot of compute in a small area at low power – but maybe it would not be flexible enough for future AI models?)
- provide extra FP16 (generally useful for a GPU, even if ML inference goes down a path that doesn't require FP16, which seems to the case right now? Everyone seems to be inferencing at 8b or lower.)
- provide a decent amount of INT8/INT4 support in the GPU just in case it turns out that really is where LLM inference converges (as does seem to be the case).
- and all this in the GPU. Apple wants to be covered just in case on-device training takes off (?); or if the next big thing, whatever that is, is a poor fit to ANE; or if developers want to run their own models on Apple GPU (as a target that somewhat matches CUDA) rather than on ANE?

All in all Apple's bets seem to have paid off well.
- They have been able to cram AFM onto the ANE, so most o the time they get the ANE low power advantages. - Developers and researchers who want to experiment running experiments and open source models have a familiar target in the A19 GPU, with reasonable performance (a lot better than A18)
- The additional FP16 HW will benefit all A19 users regardless of what they do.
- The Neural Accelerators (ie INT8 acceleration in the GPU), who knows? But they are there for external

developers, and may still pay off, especially if some new model gets wrapped up in a very convenient, very useful app.

Ray tracing

Since LLMs hit the big time, we've almost forgotten that the big GPU differentiation point a few years ago was ray tracing! Regardless of AI, Apple seems to still be plugging away at ray tracing.

Let's start with (2023) https://patents.google.com/patent/US20250095264A1 *Ray Cache with Ray Transform Support for Ray Tracing*.

Recall the basic model for ray tracing. A large number of rays are launched from "the camera" and for each ray we need to find the precise piece of geometry (think something like a specific triangle) that the ray intersects. Once we know which geometry the ray intersects, the shader then makes a decision based on the properties of that geometry. The geometry may be opaque (so we give the ray a color based on texture, material properties, lighting, whatever). Or the geometry may be reflective, so we launch a bunch of reflected

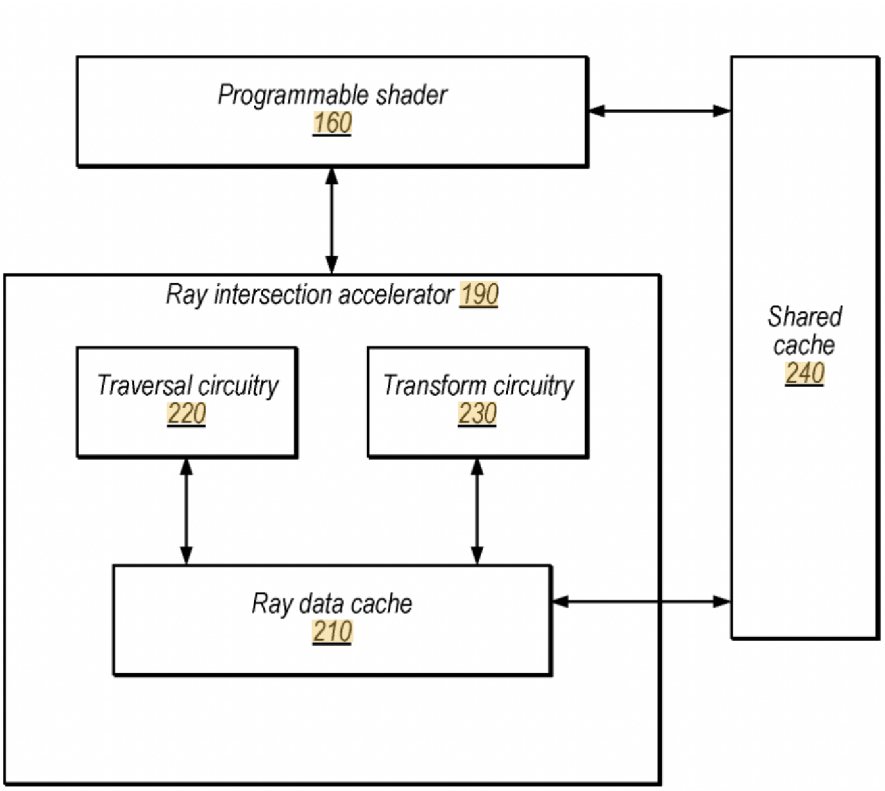
rays which then repeat the process. Or the geometry may be refractive, so we launch of bunch of "bent" rays which repeat the process. Or the geometry may be translucent or semi-transparent (think maybe marble) in which case we do some combination of all the above.

Whatever the details, the most expensive part of this process is, for each ray (initial ray, and then recursively generated rays on hitting a reflective or translucent surface) figuring out which element of geometry the ray hits. This is done by walking a large tree called an ADS (Acceleration Data Structure) and testing, for each node, which of the subnodes the ray might intersect. Hardware offload for ray tracing means providing hardware that does all this work (lots of memory access to walk the tree, along with some ray shuffling to try to get as many ray tests as possible against any particular node of the tree that has just been loaded.

So we see this basic model in the diagram below. For now ignore 230.

The Programmable Shader communicates with the Ray Intersection Accelerator by writing the details of some initial rays in the Ray Data Cache 210 which is perhaps better thought as "Ray Storage".

The Traversal circuitry walks the ADS tree, loading nodes from Shared cache 240 (ie the L2, SLC and up to DRAM) and compares that against the rays recorded in 210.



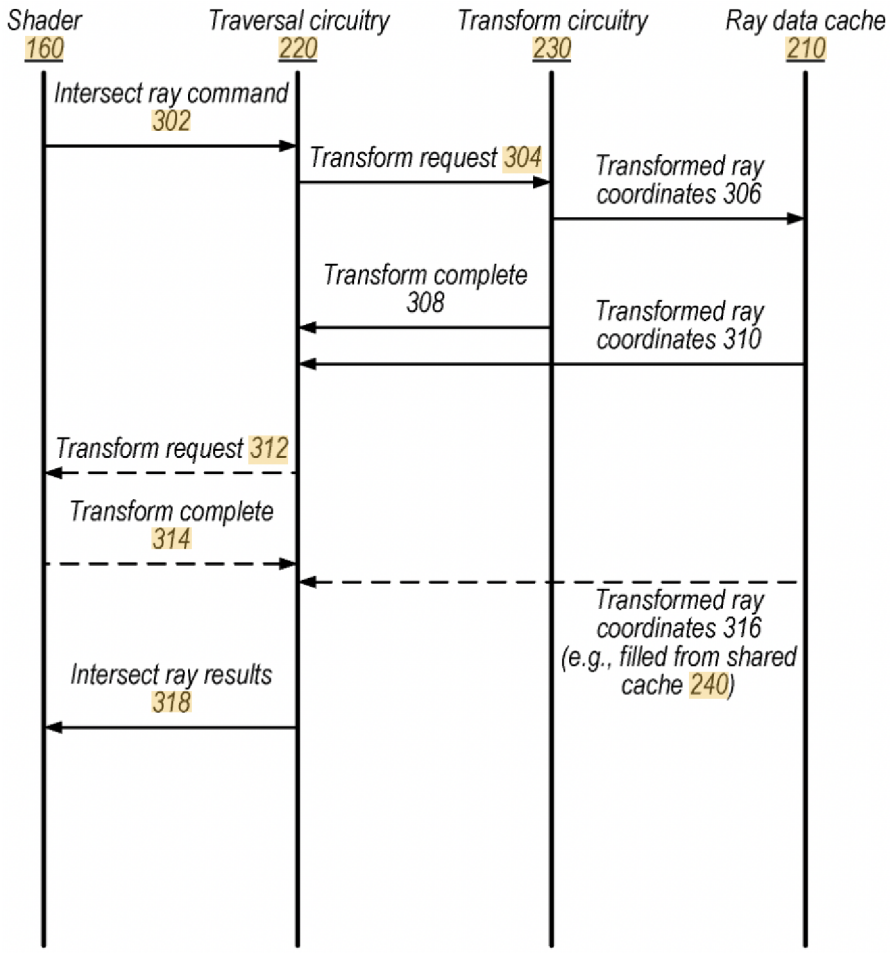
Some significant elements of the design at this point are:
- the ray parameters are written by the Programmable shader, the Ray intersection accelerator never creates rays, it simply finds an intersection point for the rays it has been given
- terminal nodes of the ADS tree will have a flag in them that essentially says “return this node and the intersection details to the shader”. The shader then inspects the node and behaves appropriately. As we have seen this may simply mean giving the ray a color, or it may mean the launching of a whole new set of rays (again by writing those ray parameters in 210).

- given the unified SRAM model of the most recent GPUs, the Ray data cache storage lives in that unified SRAM with (of course) its own address space ID.

OK, so that's the basic model. Now let's think about ray tracing a realistic large scene.
Such scenes frequently make use of "instancing" where the scene consists of a piece of geometry that is replicated many times (for example a pacific island might replicate a palm tree many times). The basic palm tree is one piece of geometry, but each time it is instanced its placed differently, rotated differently, maybe stretched a little along one axis and shrunk along another, so that even geometrically it doesn't obviously look like a clone. (The shader can then add even more variety by using some randomness in generating the colors of the rays that hit the geometry depending on the instance.)
Instancing allows the description of the geometry to be a lot smaller; wherever we have a tree, instead of the full description of the tree, we instead have an indirection that says "at this point refer to the tree geometry, but apply this transform to that geometry model".

This idea of instancing has proved very powerful for traditional rendering, but how do we implement it when ray tracing?
The most naive idea is to flatten the instancing by moving every instance into the ADS. This is only feasible for simple scenes, otherwise the ADS just gets too large!
The next idea is to set a flag within the ADS that indicates that, for this node, we need to break out of the ADS traversal back to the ray shader. The ray shader code will then transform the ray (shift both its angle and its spatial location) to move the ray to some side location where we have stashed the geometry of the instance, and we now walk that (single) side instance of the geometry using the new location/angle of the ray, which were returned to the ray accelerator by the shader. This works (of course you have to be careful with the details!) but means we lose performance every time we cross this instancing boundary because we have to communicate between the ray shader and the accelerator.

At which point you can now understand the value of the Transform circuitry 230! The new feature we have is that we can perform this geometry transform within the ray accelerator and continue testing the ray against the instance, as encoded to the side of the ADS, without having to communicate with the shader. You can see the basic idea here.
The model we have described above is essentially the dashed lines 312 and 314.
The new model is lines 304→306→308, bypassing any dashed lines.


Shader
160
Traversal circuitry
220
Transform circuitry
230
Ray data cache
210
Intersect ray command
302
Transform request 304
Transformed ray
coordinates 306
Transform complete
308
Transformed ray
coordinates 310
Transform request 312
Transform complete
314
Transformed ray
coordinates 316
(e.g., filled from shared
cache 240)
Intersect ray results
318

The actual patent has to do with details of how the new ray geometry and old ray geometry are stored in
the ray data cache 210 (and when/how they can be accessed by the ray shader if that is necessary).

Why would the ray shader want to access this transformed ray geometry?

One limitation of this scheme, as described and as implemented, is that it can only handle one level of instancing. You can obviously imagine, in principle, that our instanced palm tree has multiple leaves, and we could represent those as multiple transformed instances of a single leaf geometry. If you want to do this sort of multi-level instancing, then the second level of instancing has to be performed by the shader (as we earlier described) so the relevant ADS nodes in the tree (the nodes that refer to a leaf instance) need a flag of the old type that says "hand over the ray to the shader at this point". And the shader needs to know to use not the *original* ray geometry (in island space) but the *transformed* version of the ray geometry, the ray geometry in tree space, so as to transform it into the geometry of leaf space.

Obviously a recursive transformation scheme would be more ideal than this one-level transformation scheme. I'm not sure if this simpler scheme is just a limitation of time, to be fixed soon; or if it's legally required (maybe someone else has patented recursive HW transformations?) Doing recursive transformations is also trickier because you need to think about where to store the additional (varying length...) data associated with the ray as it passes through successive transformations.

Given this discussion of instancing, we understand that we don't want to include the full geometry of every instance within our full ADS tree. On the other hand, the transform circuitry can become a bottleneck in this design. Even if the actual volume of our palm tree is fairly small (so it's unlikely that a ray will hit it) every ray that hits a transform node has to be transformed before it can be tested against the instance ADS-subtree…

We can split the difference between these two models by also incorporating say the first level of the instances ADS-subtree, transformed from instance space to model space, along with the instance transform node. This way we can do a quick preliminary test (does the ray actually everything in the instance?) and if not, bypass the transform altogether.

How many levels down the ADS subtree should we go? Maybe two levels is better than one level? This probably depends on a whole host of factors like how "spread out" vs "compact" your instance geometry is, and how much instancing you are using relative to non-instanced geometry. Apple doesn't know, so they allow you to use whatever number of levels you want (no incorporation of ADS-subtrees from the instance subtree, only the first level, the first two levels of the instance sub-tree, etc).

This is described in (2024) https://patents.google.com/patent/US20250095272A1 *Bounding Volume Hierarchy with Bounding Volumes in Prior Space corresponding to Subset of Transform Sub-Tree Bounds*.

Another item we want to keep track of is how many recursive rays (ie rays generated by the shader because of reflection, translucency or whatever) are we sending to the ray accelerator. The ray accelerator has limited storage in the common L1 SRAM, and you don't want a situation where the ray accelerator is thrashing, constantly moving ray data from the L1 SRAM to L2 then back to the L1 SRAM.

The solution adopted introduces a new hint into the instruction stream that basically says "I'm going to

start sending new rays to the accelerator soon". The higher level schedulers (in principle both the level 1 scheduler that decides which warps to send to the channels; and the individual channel schedulers) can then reference how many rays are currently being tracked by the ray accelerator and, if appropriate, can decide to pause the ray shader for a while. Obviously the ideal situation you want is that most of the time the accelerator is doing its thing, walking the ADS, while simultaneously the ray shader is doing its thing (deciding what color to shade rays, calculating angles for refracted rays, etc); and hopefully when appropriate we can pause ray shader warps that are about to launch new rays, and replace them with warps (ray shader or traditional shader) that are more engaged in compute activity. This is described in (2023) https://patents.google.com/patent/US20250095098A1 *Hint for Scheduling Graphics Ray Tracing Work*.

Finally, consider the memory model of the GPU. The basic idea, as we understand it, is that resources live in per-resource address spaces as far as the GPU core and L1 SRAM are concerned, these are mapped into a per-process virtual address space as far as the L2 is concerned, and that per-process virtual address space is mapped into the common physical address space as far as the SLC, DRAM and the rest of the chip is concerned.
Now conceptually this all makes sense, and it's also easy to understand for many types of GPU program. The exact resources required (number of registers, amount of scratchpad storage, etc) are predetermined and the relevant virtual address space and physical pages can be specified in the binary and allocated at launch time.
But suppose our GPU code is recursive, of which the most obvious form is ray tracing (one ray can hit a reflective surface which spawns multiple rays which each hit a reflective surface which spawns more multiple rays etc etc). Now it's not clear quite how much virtual address space we need to make available (in which to map the conceptually unbounded ray address space) and how much physical address space to provide to back the virtual address space. Virtual address space is cheap, we can ignore that problem; but physical address space is not cheap, each physical page we allocate to the GPU while ray tracing is a physical page we can't use elsewhere.

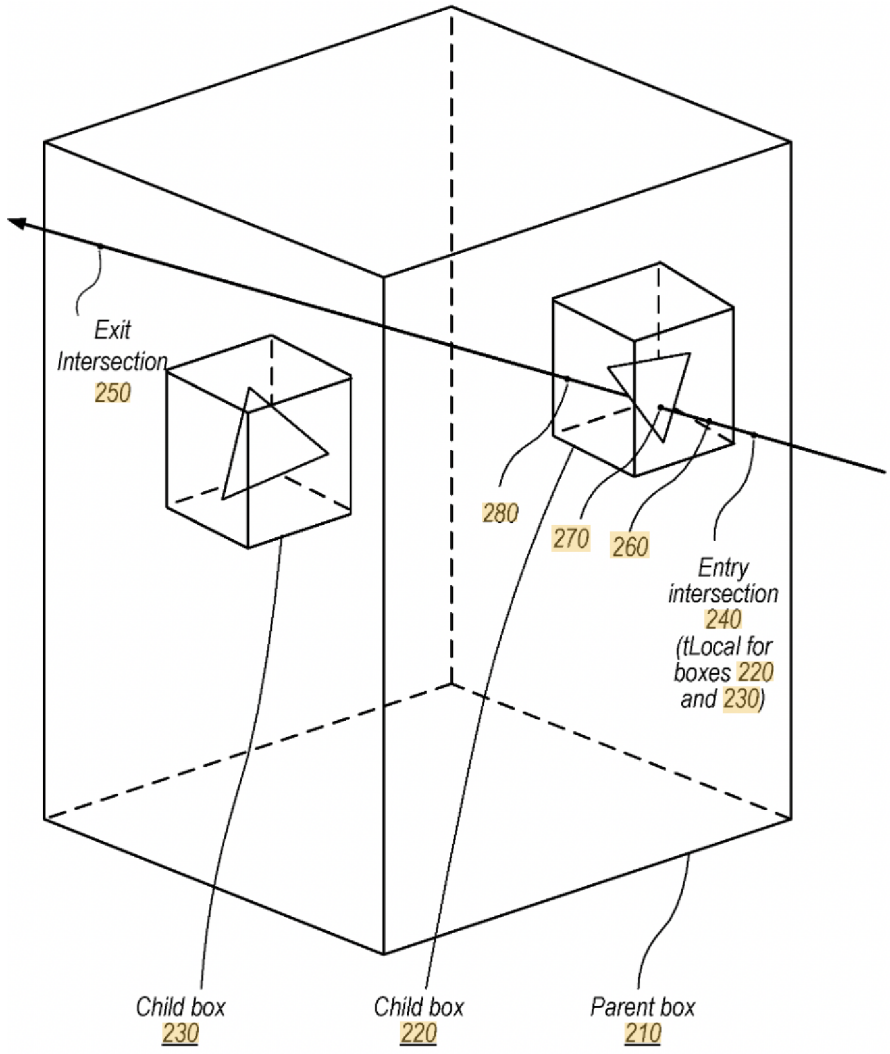
Now you can imagine a few "solutions" to this.
One is to put a hard cap (somehow) on the number of active rays ever allowed, and to force new rays to reuse the ray space allocations of earlier rays. We could maybe do this with the previously described hint mechanism, though that mechanism is designed to maximize performance, not exactly to prevent new rays being launched even when there is absolutely nothing else for the GPU to do instead of launching new rays.
We could allocate "hopefully enough" physical pages, just accept the waste, and accept that an exceptionally aggressive ray tracing program might crash the machine.
Or we could do what CPU programs do, and dynamically allocate new pages when required, but not until then. This might be difficult in a PC environment where the GPU has limited communication and control with the OS and the CPU, but it's somewhat easier for Apple with its control of the entire system.

One final ray tracing patent in this round is (2023) https://patents.google.com/patent/US20250095274A1 *Pruning Ray Tracing Traversal Operations based on Local Ray Parameter Value*.

We've been a little sloppy in talking about what happens when a ray finds an intersecting triangle. Imagine a ray as a straight line beginning at the camera and going out to "infinity". When we find an intersecting triangle, that's not the end of the story. Yes we have an intersection, but there may be another, closer triangle, that also intersects (in other words what we have discovered is a hidden surface). So we have to keep iterating the search even after finding an intersection, until we find the closest intersection.

However can, at least sometimes, we make use of an intersection to prune some of the later parts of the ADS tree that we might otherwise need to search? The idea is explained below:

We have a large bounding box which indicates an intersection, so we step into that box node (which contains two smaller boxes). If we get an intersection in the first box, then without looking at the details we can calculate that even a possible intersection in the second box is not interesting because the intersection of the first box is closer to the camera.


Exit
Intersection
250
280
270
260
Entry
intersection
240
(tLocal for
boxes 220
and 230)
Child box
230
Child box
220
Parent box
210

The bulk of the patent is how you can make use of this idea, and involves a few different elements including details of the exact order in which you walk the AD tree, and some additional intermediate state that is associated with each ray.

Mesh support

I know even less about Geometry (in the sense of GPUs) than I know about the rest of the GPU pipeline. Even so, there have been some interesting developments here, so let's overview them.

The original geometry stage of the GPU pipeline takes in a scene (essentially a large collection of triangles placed in 3D) and converts them into a set of triangles placed in 2D. The essential operations are a transformation of each corner of each triangle (offset + rotation+scaling, to match the plane of the camera) and hidden surface removal (ie ensure that only the frontmost surface is visible). Generally it's not optimal to do all of hidden surface removal in Geometry processing, some is left till later, but it makes sense to do what you can at this stage.

OK, so how do we make this fancier?
An obvious next step is to add instancing, just like we described in ray tracing above.
More sophisticated is adding the ability to display curved surfaces, which goes by the generic name of tessellation. The idea is to represent a curved surface not as a set of triangles but as an algebraic expression and, at execution time, we dynamically tesselate the surface (ie convert it into a set of triangles that match the curved geometry to required precision). This is dynamic in the sense that the number of triangles, and even their orientation, changes depending on whether we are near the surface (many small triangles required) or far away (just a few triangles is good enough).

Now the constrained form of tesselation (ie the sort of thing you usually get in the first generation of new GPU hardware) solves the exact problem mentioned above, of converting a *particular representation of a curved surface* into triangles. But looking at the problem more generally, from the point of view of thinking like a shader, ie like a programmer, what we're doing is taking a block of data and converting it into a set of triangles. And that block of data could be anything! The more generic version of this sort of thing is called a mesh shader, taking in a block of "data" and generating a so-called meshlet, which is a small fragment of geometry
Why might this be useful?
One possibility is the procedural generation of geometry. So rather than statically represent fire or water or whatever as some curves, run a program that (either by doing genuine physics, or by doing pseudo-physics that looks correct) evolves the geometry of the fire or water as appropriate. Or clouds, or landscapes, or whatever.

However there's another possibility, which I haven't seen much discussion of outside Apple, and this is *geometry compression*. We somehow shrink the geometry (maybe losslessly, maybe lossy) and then extract it from the compressed representation at runtime.
I assume ultimately this is going to hit the big time on Vision Pro as ever fancier, dynamically generated 3D content is used to augment reality. Apple have a number of patents that suggest this is the sort of thing they have in mind.

We start with (2023) https://patents.google.com/patent/US12198389B2 *Three-dimensional mesh compression using a video encoder* which is exactly what it says (and builds on work from 2020).

We start with a point cloud (in other words the data we get from a camera but with associated 3D info – possibly derived from lidar, possibly derived from a stereo camera, possibly derived from a neural network as in Apple's Portrait mode photos; we could even have a synthetically generated point cloud, eg a scientific visualization), and use the video compressor to convert this into multiple shrunk patches of geometry and texture. You can do this either with a one-time representation of a 3D object/scene, or with a sequence of scenes (ie a "3D movie" or a scientific visualization that's changing in time, and which you can rotate and examine from every angle).
There are a whole lot of additional associated patents that get into various technical details of the specific geometry compression scheme.

This is accompanied by (2022) https://patents.google.com/patent/US12256098B1 *Real time simplification of meshes*.
The previous compression scheme, while lossy, still preserves the original content in full detail. Which means a lot of detail to process by the GPU, often more than is required given the particular way in which we are viewing the content (maybe far away, so we only need a reduced level of detail). This patent describes what one might call *lossy decompression* (as opposed to lossy compression) to extract as much geometry as is appropriate for the desired rendering and no more.

Finally we have (2024) https://patents.google.com/patent/US20250095268A1 *Mesh Shader Work Distribution*. Now the problem is that when executing a mesh shader you have an amplification of geometry, in other words a small amount of data goes in and a variable amount of geometry comes out. (The same problem already exists with Tesselation, but in that case it's somewhat more controlled because that problem is more constrained.)
We first have to be able to allocate enough storage as required on demand, and we secondly need to throttle the mesh shader relative to the rest of the graphics pipeline to keep geometry generation matched with vertex transformation, culling, and subsequent pixel shading; not too fast, not too slow. The patent describes techniques to enforce this.

Power management

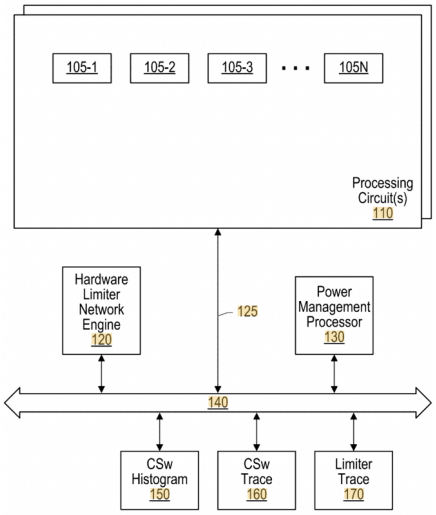
We've seen the CPU and SoC power management story a dozen times. The basic idea is always
- capture a large number of different measurements
- aggregate these (over time and space)
- extrapolate to how we expect the IP block to behave in the future
- establish power settings accordingly

This seems obvious, but much of it was new to Apple (though now widely copied). This includes just how many measurements were tracked, proactive power handling (predict what the system will do rather than just do what would have been optimal over the most recent past), and the rate at which power settings were changed (fast enough that this required dedicated hardware, rather than having the OS do this over a very longer aggregation time).

(2024) https://patents.google.com/patent/US20250103122A1 *Hardware Performance Information for Power Management* gives us more of the same, but in particular targeting the GPU.
The previous state of the art for GPUs appear to have been to track something like "GPU utilization" for

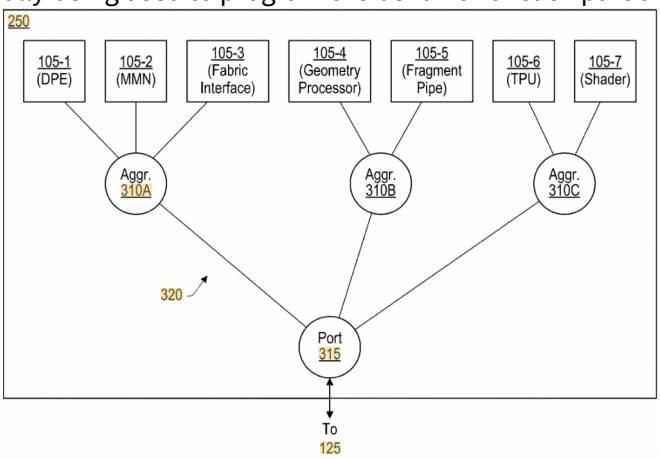
a particular core over a frame, and then to set the frequency in the next frame to something that would drive this utilization to about 95%. Problems Apple sees with this include:
- the frequency is too coarse. Much GPU work involves different phases, for example maybe an initial phase that is primarily compute limited (and so benefits from high frequency), followed by a second phase that is primarily bandwidth limited (loading in and compositing various elements calculated in the first phase and writing out the result), and this second phase is best handled via low compute frequency but high communication (NoC, SLC, DRAM) frequency
- the measurements are not careful enough. You don't want something as coarse as "GPU utilization", what you want to know is, for each phase of calculation, what was the limiting hardware (eg was it FMA compute? texture compute? ray compute? L2 bandwidth? etc etc) Obviously then you dial the limiting hardware to a frequency that gives that hardware around 95% utilization over the next frame (or fraction of a frame) while dialing the non-limiting hardware down to lower performance, to try to match the limiting throughput.

To achieve this we have hardware like so:



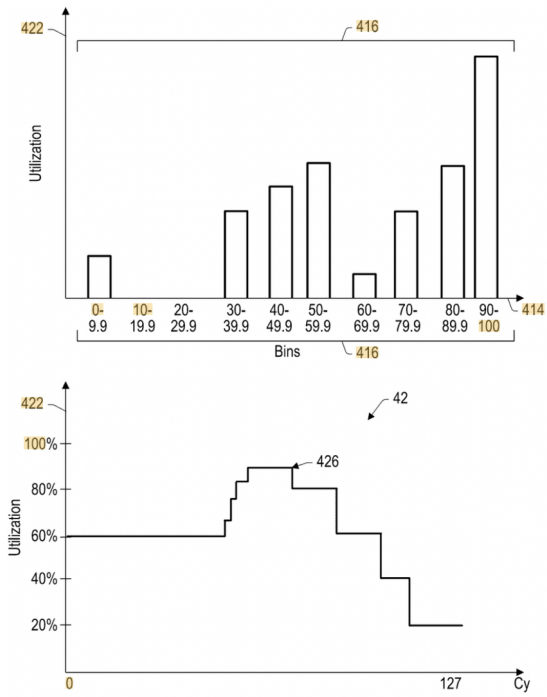
Each core replicates performance registers. The Limiter Engine reads these on a particular schedule (ie at least once per frame; ideally more often) and aggregates them into consolidated storage 150, 160, and 170. Finally The power Management Processor extrapolates from this storage to not just calculate the optimal settings for the previous frame (and hope the next frame is the same), but tries to detect and extrapolate trends.

This is reasonable enough, but there is more complexity going on. Moving the data around happens over a dedicated GPU-wide bus called the RIF (Register Interface Bus) which presumably is also continually being used to program the behavior of each part of the GPU.



Above shows us part of what's going on inside one GPU core. A broadcast goes out to each performance register asking for the latest values, which are (at least for a few related elements like 105-1 (Digital Power Estimate), 105-2 (memory bandwidth) and 105-3 (fabric bandwidth) normalized in some way (so maybe 0..255). These values can the be aggregated at aggregation points which shrink the values down to something like "max", "min" and "mean" [or "sum, which is equivalent to mean]). (You probably want to aggregate values that correspond to the same piece of hardware, eg texture compute utilization and texture bandwidth utilization, so you have an overall feeling that the texture unit is running at, say, 80% utilization. Similarly for memory bandwidth type values.)

After aggregated values are reported to higher levels, we can then create consolidated reports that look like:



One type of report is a histogram showing utilization of some block of hardware over the last relevant time frame, another is a graph of this utilization. From these (looking at both within a single frame and over the past few frames) we can then calculate a plan of expected utilization over the next frame, from which in turn we can calculate an optimal frequency for this particular block within a particular GPU core during some subphase of the frame computation.

An additional type of information collected for compute-style blocks is effectively “fraction of cycles that useful work was done” and “fraction of cycles that block was stalled”. These values are stored in a third consolidator called the Limiter Trace.
From this sort of information you can get insight into whether the optimal response is to increase frequency (maybe so if lots of work is being done) or to lower frequency and take the power saving (maybe so if most cycles we’re actually stalled waiting for some other unit to deliver data).

Obviously there are limits to how finely you can set frequency across a GPU (or any other IP block). Each
time a signal has to cross a clock domain, there's a delay and some buffering to synch the signal between the two frequencies. So, for example, within a single GPU core it's probably impractical to run the texture unit at a different frequency from the rest of the GPU core. Even so, we have two options. One is that we power down the texture unit (ie run it at 0GHz), which may be feasible if we have good reason to believe it's not being used. The alternative is to occasionally freeze the clock going to the texture unit; for example if we calculate that the optimal frequency for the texture unit is 80% of the GPU core, we can freeze the clock to the texture unit for one cycle in five.
One final detail is that, as opposed to earlier power metrics, where possible Apple now tries to track, for each IP block/subblock of interest, the "switching capacitance" which is a metric that takes into account the number of transistors, how they are fabricated/connected (ie their capacitance) and how often they switch (since some transistors may be wired to switch at say half the nominal frequency of their subblock). The switching capacitance for a subblock essentially tells you how much energy you will save by occasionally freezing the clock for a cycle or two.
There's way more detail than this, but you get the idea.

It's not mentioned, but I assume at least some subset of these various performance counters, or the associated histograms and traces, can be passed on to the OS and thus used to improve the Instruments data provided by XCode.

If we abstract from the above, we see a few basic ideas.
One is aggregation of data, so that we track the performance of multiple elements of an IP block, but subsequently compress all those numbers down to one or a few single usable numbers which can inform us of an appropriate performance level (GHz, bandwidth, whatever).
Another is collecting data in a scaled form (eg as fraction of maximum) so that various numbers (70% here vs 70% there) can be more usefully aggregated or compared.
Another is tracking history over longer than just one epoch; and
Finally splitting agent responsibility into
- one item that handles data collection from performance registers,
- one that handles policy generation (predicted future performance states), and
- a third that handles policy implementation (
+ translate the desired performance into register settings,
+ ensure that this policy is achievable [maybe we cannot change current that fast, maybe the system is close to overheating], and
+ transmit the actually achievable register settings over a dedicated register interface bus).

Something like this now seems to be applied SoC-wide. One advantage of these abstractions is that it's easier to write a single piece of firmware for multiple chips with just a few clear details that have to be modified next generation. Which means we now see much the same ideas and division of responsibilities in a few different places, eg

(2023) https://patents.google.com/patent/US20250076948A1 *Network Fabric Power Management*; and (2024) https://patents.google.com/patent/US20250093937A1 *Multi-Processor Power Management Circuit*.

This patent (2023) https://patents.google.com/patent/US20250086009A1 *Provisioning of performance states for central processing units (cpus)* is fairly minor but an interesting idea.
One of the ways we try to save energy is by never running code faster than required. One way to do this is annotations (in other words we mark the code as having a certain QoS like User Interface vs Background). Another way is certain IP blocks (eg media decode or generating graphics) run at a predictable rate, and we can slow down the media block or the GPU to a frequency that's just high enough to generate a new frame every 60th of a second (or whatever).
The primary idea, as far as I can tell in this patent, is that this doesn't just apply to the GPU. In other words if the CPU is tracking the same sort of pattern as the GPU, doing some work to create the next frame then sleeping, then we can likewise slow down the CPU to just fast enough to get that work done in time on the CPU. (Obviously there are details in this, and that's what the patent is about. For example there may be variation in the amount of work done by the CPU, or the work may be spread over multiple threads.)

SoC

Memory controller changes

Replacement of tags with a directory

We begin with a fairly technical change that, however, suggests Apple has big ambitions: (2024) https://-patents.google.com/patent/US20250103496A1 *Coherence Directory Way Tracking in Coherent Agents*.

The SoC has multiple caches, for example a few L2 caches for P and E clusters, maybe a cache for the media block(s), etc. Suppose the CPU generates load request that does not hit in the cluster L2. How do we know where to direct that request? I particular, which other possible L2 to direct it to.
The request first goes to the SLC, and in the past the SLC had a set of tags matching each relevant cache in the system, so the request address would be tested against all these sets of cache tags.
This works and is reasonably area efficient, but it's not idea in terms of energy efficiency, since many tests are required (tests in each set of tags), and this gets worse as we add more and more additional "L2-level" caches that are possible data sources.

How can we improve this?
Ultimately we have some number, call it *N*, of addresses corresponding to the total of all the lines in all the L2-level caches. We want a fast way to look up any one of these *N* to find the particular cache in which the line lies, along with its MESI info and so on. This sounds like a job for a hash table!
So imagine we use a non-trivial hash on the address, to create an index with a value into a table sized somewhere between 1.5 *N* and 2 *N*. Experience and theory tells us that *most of the time* we can pack

N values into a hash table of that size without collisions. OK, that's a start, now how can we improve things?

When creating a SW hash table you use all sorts of clever tricks (quadratic hashing, cuckoo hashing, etc) to work around hash collisions. In HW our options are more limited if we need a result in a well-defined time 9as opposed to repeatedly probing the table).
The first, easy, option is instead of sizing the table as providing, say, 2 *N* indices, we size it to provide $\frac{2N}{4}$ indices, where each index refers to 4 slots. Now we can handle up to 4 hash collisions for any index before things go wrong, and this tends to work better. (This is obviously now a set with 4-ways design.)
There's nothing magical about 4! Various considerations might suggest that 8 or 6 or 12 are slightly better choices. A larger number means hash collisions occur less often (the extreme case is 2 *N* slots attached to a single index, which has become a fully associative table, and obviously never has a collision as long as there are fewer than 2 *N* entries...) On the other hand, a larger number means more searches (4 if there are four slots) for each lookup, and part of what we are trying to do is limit the number of lookups, since each costs energy.

We can also provide an additional small fully associative "overflow" table to hold a few collision cases; or we can use a skew scheme (a scheme that uses two different hash functions). Both of these work well, in the sense that they can take us from (making up numbers!) say a hash collision one in a thousand times to one in a million times. But we still have to deal with that (hopefully rare) collision.

So what are we actually talking about?
What we're saying is suppose the E-cluster allocates a new L2 cache line, and the address of that cache line results in a hash index (in the directory living inside the SLC) where the four slots corresponding to that index are already occupied. What to do?
We can't leave the line unallocated in the SLC, because that's how the rest of the system knows what's happening in the E L2 cache. So we're forced to throw away one of those pre-existing lines, which means sending a message to the relevant cache telling it to invalidate that line. This is obviously not ideal, but our hope is that if we design our hash function well and the hash table is large enough, it's a rare occurrence, rare enough not to worry about. Which of the four slots to sacrifice? If this really is a rare occurrence, then probably choosing a random slot is the best choice – easy to implement, and anything better won't improve performance much any way.
So at the end of the day, by making this change
- we probably use slightly more area
- our total effective L2 capacity goes down very slightly (because of rare collisions where five lines in the collective set of L2's maps to a single 4-slot index, so one of those lines has to be invalidated)
- but we use less power.

Intel's big Xeon's switched to a directory scheme somewhat like this around Skylake, but even they were not doing anything new; something similar (sometimes called a snoop filter) has been used in large machines for many many years. There are obvious downsides, but the upside is you can now

scale your machines larger. This suggests that Apple plans for more “L2-level” caches at the high end, which in turn suggests the mythical Extreme is getting closer. (And/or we’re also going to see Max machines maybe shifting to three P-clusters? Another way to go would be machines with maybe one P-cluster and many E-clusters, something that might be desirable for a certain type of server where most tasks are mainly waiting on disk, network, or DRAM?)

So what other issues do we face?
Suppose we have a MESI protocol. Then the problematic state is shared – how can this system tell us about a shared line? The obvious answer is a bitmap. So, suppose that this directory is supposed to cover 8 L2 caches. We could provide 3 bits per slot, which describe the *one* L2 in which the line sits. Or we could provide 8 bits, with each bit set if that particular cache holds the line.
Our cache protocol may be more elaborate. MERSI marks one of multiple shared lines as a “round robin” line. Suppose that L2 A asks for that line and it is present, shared, in L2 B and L2 C. The most basic option is the line is returned to L2 A from DRAM. But a faster, lower energy solution is to return the line from B or from C. Which one? I’d have though a random choice is good enough, but MERSI marks one of B or C as the “forwarder” for that line. This R status will move with the line so that next time someone asks, A will provide the line.
Another version of this sort of thing is MOESI. Now the line that’s shared is modified. So B modified a line, now A asks for the line. MESI handles this by writing the line back to memory so it’s no longer modified relative to memory, and A can get it from memory (or maybe even pick it as it is being moved to memory).
MOESI suggests that we consider this line as modified relative to memory, but still capable of being shared. So A will pick up the modified line, but now neither A nor B are allowed to further change it. This works if we have a line that occasionally gets changed but is mostly read only. The one trickiness in this, however, is we now have two copies of the line. Suppose for some reason the line needs to be forced to be written back to memory (as a basic example imagine the system is going to sleep so the caches will all lose power). Which of A or B will write the line back? Again MOESI solves this with an extra state, the owner state, which is responsible for this task.
In both these cases, MERSI or MOESI, we’d need an extra bit (or two bits if we do both) per cache to exactly follow the rules of the protocol. Who knows what Apple does! But with the SLC in control, it may be possible to get the advantages of MOESI or MERSI without the rules of these specific protocols and so without the extra bits? Maybe we can share read-only lines with the SLC making a random choice as to which L2 cache to use as the line source? And maybe, if the SLC is always involved in the kind of decision that requires one (but no more than one) of multiple L2’s holding a shared modified line, the SLC can again just decide as a random decision which L2 writes back the line? Certainly the patent gives us no insight into these details!

So the main significance of the patent is that it tells us Apple is now using a directory. But you can’t patent that idea, which is probably 50 years old!
What the patent specifically covers, after some background regarding the use of a directory, is a slight energy optimization. Suppose cache L2 A engages in some cache management, for example it invali-

dates a particular line. This fact needs to be transferred over the NoC to the SLC which will update the directory appropriately. This update means hashing the address (to find the hash index, looking into the index, and comparing the address against each of the four slots). Can we reduce that work?
The patent suggests that when the line is first included in L2 A, the relevant message from SLC to L2 include a few bits that state which of the four ways (ie which of the four slots) is being used by the directory to hold this address. Those bits are then stored in the L2 tags, and can be returned back to the SLC at invalidation time.
This seems like pretty minimal energy savings, but maybe the specifics of the L2 cache layout and current process technology mean that you can add two extra bits for free to each tag, for this very specialized purpose?

(2024) https://patents.google.com/patent/US20250103520A1 *Memory Controller Reservation of Retry Queue* is very technical.
The basic model of the memory controller/SLC is that on-chip networks deliver requests into a buffer sitting right in front of the memory controller/SLC. An arbiter looks into that buffer and decides on the next request to process.
That request then proceeds through a few (parallel) stages of checking the address against
- tags (does the address live in some cache, possibly the SLC, possibly some L2, in which case we route the request to that cache)
- the directory we just described above
- the various memory controller queues, which are basically requests pending in the memory controller (if we get a match, eg a request wants to read from a write that's pending to DRAM, we need to order things appropriately – either wait for the write to complete, or forward the write line to the request)
- the "snoop queue" which we can think of as earlier requests that are waiting for something to happen (a cache needs to change state, deliver data, acknowledge data, or whatever). We probably need to wait until that snoop action is complete before going further with the request.

Now, in the past what would happen in the case of a request matching in the snoop queue is that the request would be "rejected", meaning that it would have to go back to buffer sitting in front of the memory controller/SLC, wait for some amount of time, then try again. Similarly we may get a match to the directory, but the directory might be busy working with that line (for example it just got added to the directory, or it's being evicted from the directory because a hash collision). Either way we want to delay things until the directory has settled its handling of the matching line.

The modification of this patent is to create a dedicated *retry* queue within the memory controller/SLC. This saves a little energy in terms of data movement (since we're not crossing an IP boundary, the retry queue is closer to the rest of the memory controller/SLC).
It also boosts performance a little since we can wake up the relevant queue entry for a retry at the exact point that all criteria have been met to allow for a retry.
Finally it probably helps in terms of engineering effort to move this work onto one side of the memory controller/SLC boundary (where it matches the rest of the work that's happening) rather than requiring it to be split between the team that's doing the memory controller/SLC, and the team that's handling

the NoCs and their buffers.

We also see the usual Apple concern with QoS; for example although this is described as *a* retry queue, there are in fact multiple queues for different classes of client so that, for example, even if the IO retry queue fills up, that will not impact the CPU queue and thus slow down CPU performance.

So it's a cute optimization. But the other thing that makes the patent notable is that it's very detailed. If you're interested in the sort of work a real memory controller has to do, and all the sorts of interactions that have to be tracked, you could do worse than read this patent for at least a high level overview of the issues!

For example it's worth remembering, for both the above patents, that the *system* cache line length is 128B. This means a system line is two CPU lines. The system seems to handle this by transferring 128B lines to and from the CPU L2, while the CPU L2 transfers 64B lines to and from the CPU L1. One consequence of this is that there's a small optimization available in tracking which half of a 128B line has been modified, so that possibly only half the line has to be written back to DRAM.

We saw a first version of this idea in the already discussed (2023) https://patents.google.com/patent/US12007901B2 *Memory cache with partial cache line valid states*. In that case the patent allows tracking some aspects of an SLC line at half granularity, which seems to be mainly for more efficient handling of remote atomics, though it could also be used to optimize writeback to DRAM.
This *retry* patent includes one small additional feature tucked away towards the end; it mentions that in the past some of the L2 caches that used 128B lines might not track dirtiness at the 64B level, but now all caches in the system do. So 64B dirtiness is tracked all the way from lowest use to the SLC, allowing for a slight reduction in DRAM write traffic.

Another example has to do with memory bandwidths. in the first two generations of M series, essentially the A series has a single memory "controller", the M has two, the Pro has four, and the Max has eight. Translated to bandwidths, for a single specific grade of LPDDR5 this means essentially the iPhone has a bandwidth of 50GB/s, M has 100 GB/s, Pro has 200 GB/s, Max has 400 GB/s. This was essentially what we saw in the early days modulo specific complications (eg the speed grade of the early LPDDR5 meant the relevant A and M1 had 33 and 66GB/s, while Pro and Max were at 200 and 400; also for Max not all this bandwidth was available to the CPU clusters, only the GPUs could fully exploit it).
However this has changed. The M3 Pro tried an experiment that used three rather than four memory controllers. Apart from the obvious consequences (now a bandwidth of 150 GB/s, and the memory sizes are all multiples of 3) the main interesting effect of this is that we need an address hash circuit (to divide address requests between the three controllers) that's effectively a divide-by-three circuit, and we saw the existence of such with the *Routing Circuit for Computer Resource Topology* patent.
What appears to have happened with the M4 generation is a slightly clarified (and rather clever) segmentation strategy. The new split between work that's done "in the network" versus "in the memory controller" makes it fairly easy (probably dynamically, at startup) to disable some of the memory controllers. This in turn means that the segmentation we see for the M4 (and something like this might

stick going forward) looks like (again using approximate bandwidth numbers):
M4 has two memory controllers and bandwidth of 140 GB/s (actually 120, probably using slightly slower DRAM)
M4 Pro has four controllers and bandwidth of 280 GB/s
M4 Max cheap has 6 controllers and bandwidth of 420 GB/s
M4 Max expensive has 8 controllers and bandwidth of 560 GB/s
ie in the M4 Max cheap config (the one that has slightly fewer P cores and GPU cores) two of the memory controllers are also disabled.
You can also see that this works out by looking at the memory configs. The M4 Max cheap config is 36GB (six times 6GB DRAMs). The M4 Max expensive configs are 48, 64, 96 (ie 8 times 6GB, 8 times 8GB, 8 times 12GB) all corresponding to DRAMs you can buy. But there's no 4.5GB DRAM you'd need for 8 times 4.5GB=36GB. (You can now buy DRAM in 6GB or 12GB capacities.)

Yet another clarification example comes in discussing the NoCs. The patents have repeatedly discussed that certain situations do or don't allow memory re-ordering, while never being very clear about what this means. The new clarification is that, essentially
- we have three logical, four physical networks to the memory controllers, one for CPU, one for IO, possibly two (ie extra bandwidth) for GPU. Where does ANE fit? My guess is that right now it lives on the IO network but maybe it will get its own (or be incorporated into the GPU networks?) as AI becomes more important.
- of these, the CPU and IO networks are "ordered", the GPU network is "unordered". More precisely each network makes use of virtual channels, and these are unordered relative to each other. However within a virtual channel
+ the GPU allows extensive ("relaxed") reordering, except between transactions that refer to the same address. This means that, among other things, two transactions may begin one before the other, but complete in a different order, the later one not waiting for the earlier to complete first.
+ the CPU has to follow ARM rules which have some flexibility, but not the reordering flexibility of the GPU
+ IO has specific rules that can be less flexible than ARM (eg PCIe has to follow rules that essentially match x86). I don't know enough about IO details to understand or track this at all. It seems to mostly apply at the network boundaries, so that specific harsher ordering is enforced at the transition from the IO NoC to the PCIe hardware.

All this (dedicated GPU network[s], dedicated connection from each network to each memory controller) means that the GPU path can achieve higher bandwidth through maximum transaction reordering flexibility – at the cost of some latency, of course but that can be and is controlled.

Better accounting for evictions

XXX

We've seen that QoS is an ongoing obsession, with small (and sometimes large) changes every year. (2025) https://patents.google.com/patent/US12353913B1 *Handling eviction write operations caused by*

rate-limited traffic gives a very slight tweak to the existing system.
Some clients prioritize latency over bandwidth, and get various privileges as a result, but the tradeoff for that low latency is that they're not allowed to exceed a certain bandwidth. The CPU is a special case, but generally all low latency IP blocks include credit tracking circuits that track how much bandwidth they're using and throttle as the credits become low.
Now, how is that bandwidth calculated? In particular, suppose that an IP block generates a read request that results in the read being allocated in the SLC, and so a modified block being evicted from the SLC out to DRAM. In the past this writeback traffic was counted as bandwidth against the read-initiating IP-block. The patent suggests that this is sub-optimal (if you prefer anthropomorphic language, it's "not fair") because some other client could have placed that block in the SLC a long time ago, and so in this scenario the writeback no longer counts against the read-initiating IP block.

There are a bunch of somewhat mysterious aspects to this which maybe make sense if we knew everything about the SLC. For example I'm not sure how this scenario even arises. My understanding is that the CPU treats the the SLC as a victim cache, so that reads would never allocate in the SLC. Even if it makes sense for some IP blocks to allocate in SLC (eg if they only have a small local L1, no L2) I thought all allocations in the SLC were already tagged by the originating IP block in various ways, so its easy to charge the right client appropriately for castouts.
Maybe this patent is ultimately a bug fix (though they wouldn't phrase it that way)? They realized that castouts happen so much later than reads that it's counterproductive to penalize reads now based on writes far in the past, and the system runs smoother by not being quite so obsessive?

Network for non-DRAM targets

Obviously by volume most transactions target a DRAM memory address. But some transactions (eg to configure hardware, including the memory controllers) target a non-DRAM "address".

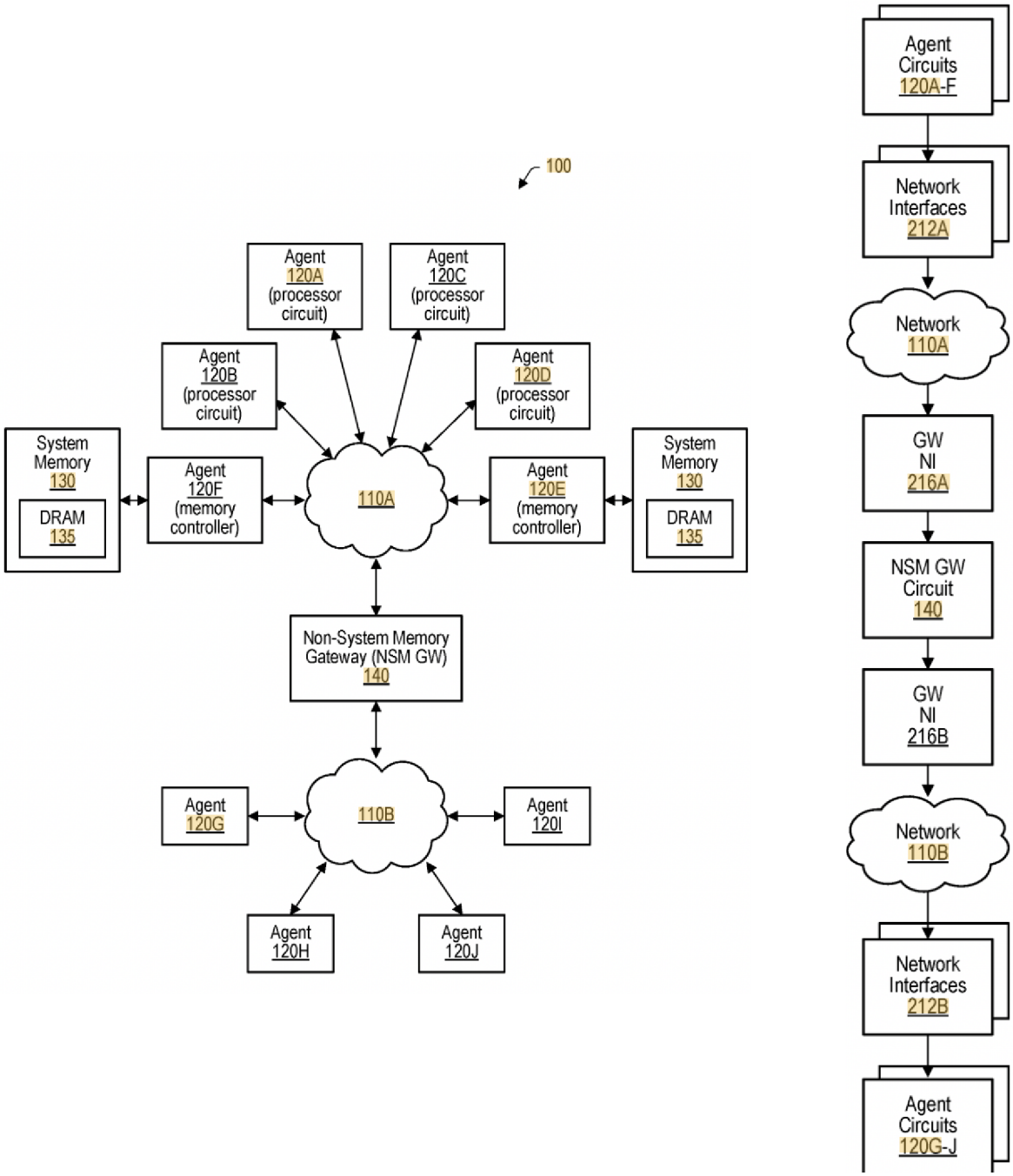
This might seem a fairly limited task, restricted to startup, shutdown, and sleep, but we've seen that over the years vast amounts of telemetry is happening in the chip as performance statistics are transferred from the edge to centralized locations and converted into a plan for DVFS and power distribution over the next few cycles. I'm assuming these sorts of "invisible" tasks also take place over this non-DRAM network.

The patent is somewhat vague on exactly how this used to be done. It seems like the physical transport used was one of the existing NoCs, but fancy routing functionality was not available (maybe the required routes all just happened, through a combination of luck and some small design effort, to be available as point to point connections?) and so the required arbitration and buffering/queues of transactions had to be replicated in or near each agent, with this memory and logic replicated for every possible source/destination pair.

This might have continued, non-ideal but sustainable, for a few more years, except that the patent

suggests that chiplets are coming, and chiplets require a rethinking anyway of a variety of aspects of the NoC. So, as part of this overhaul, we get a new mechanism for these sorts of transactions. The essential idea is (like to the other NoCs) to provide a centralized gateway consisting of buffers, arbitration, and routing logic, to which and from which all transactions of this sort flow. (Unfortunately such details as are described are all framed in terms of a monolithic chip! So apart from the introductory motivation, we get no insight into how Apple may split say a Max level design across two or three or four chiplets.)

We can see the idea like so:

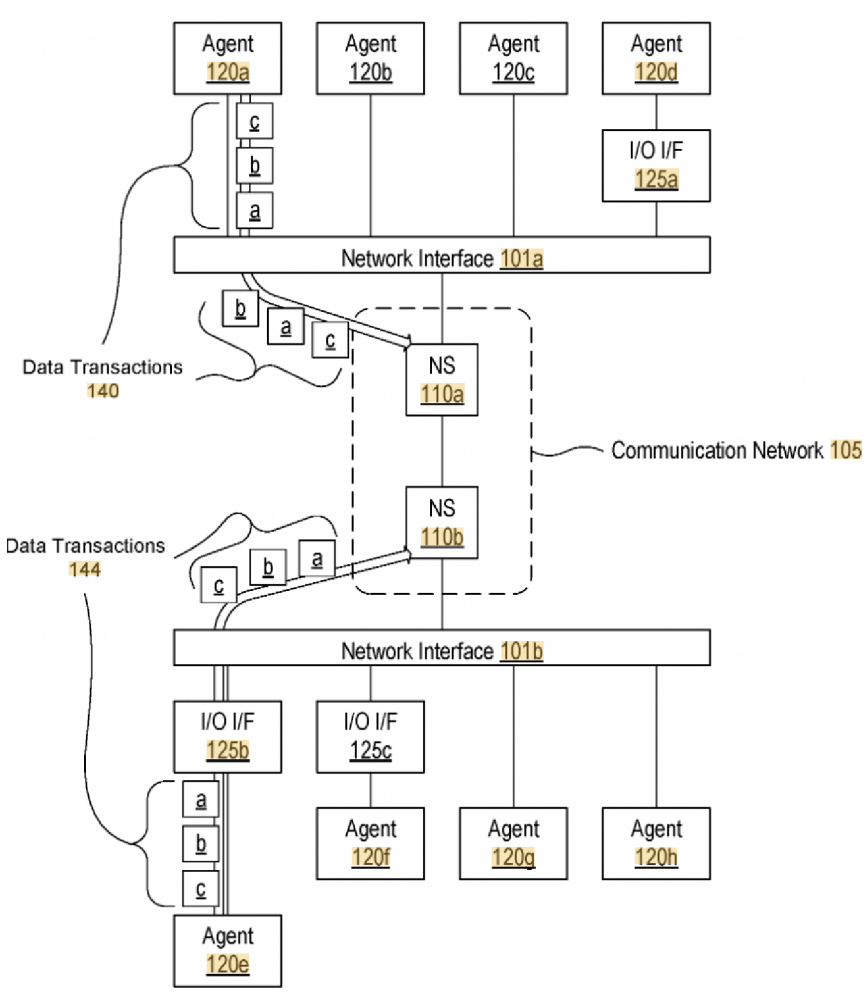

100
Agent 120A (processor circuit)
Agent 120C (processor circuit)
Agent 120B (processor circuit)
Agent 120D (processor circuit)
System Memory 130
DRAM 135
Agent 120F (memory controller)
110A
Agent 120E (memory controller)
System Memory 130
DRAM 135
Non-System Memory Gateway (NSM GW) 140
Agent 120G
110B
Agent 120I
Agent 120H
Agent 120J
Agent Circuits 120A-F
Network Interfaces 212A
Network 110A
GW NI 216A
NSM GW Circuit 140
GW NI 216B
Network 110B
Network Interfaces 212B
Agent Circuits 120G-J

Obviously this left diagram is suggestive of an IO chiplet separate from a compute chiplet, but we'll see how this plays out.
In more detail the flow looks like the right diagram, which shows Network Interfaces sitting between every Agent (which could be CPU, memory controller, IO block, etc) and the network.
We've seen before that the Network Interface performs two roles. One is buffering buffering and arbitration for access to the network. The other is conversion between however the Agent wishes to perform its logic and what the network requires. This conversion could, in principle, be voltage conversion (one side uses a different voltage from the other), or frequency conversion, or network width conversion (eg one side operates in units of 16B, the other side in units of 64B). Most sophisticated is if the IO device requires some specific unusual element in terms of ordering or grouping or whatever, where the Network Interface can handle these IO specifics.

Another aspect of this is is to view networks 110A and 110B as something like LANs; one might be an ethernet LAN, the other a WiFi LAN. Point is, each network can be specialized for a different task, with 110A specialized for high bandwidth and low latency (what CPU clusters require) while 110B is specialized for the lowest power possible (since it is not as active). The Gateway between them, like an internet gateway, can perform protocol and address transformations as required.

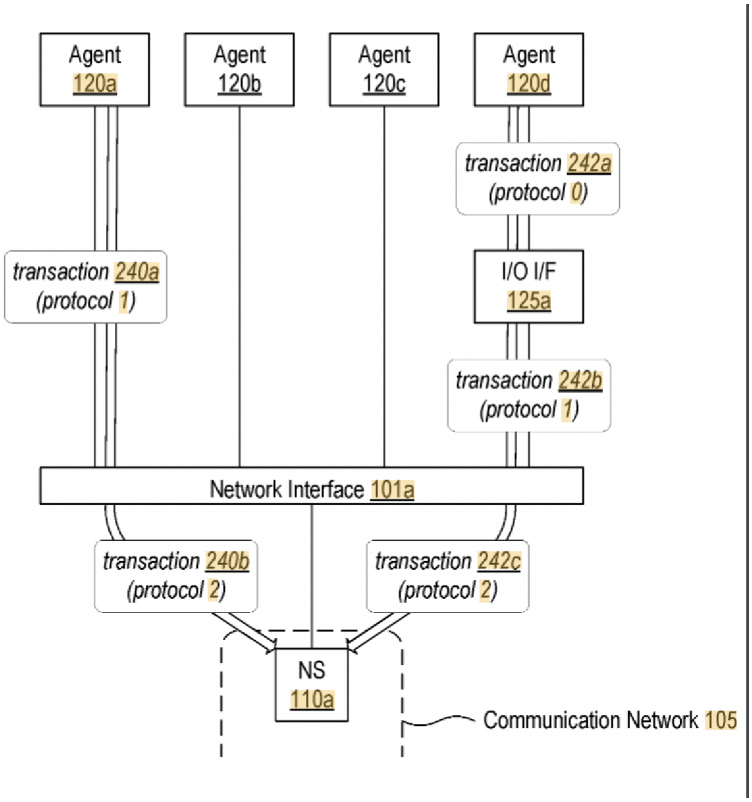
Most of the rest of the patent is routine; the one additional cute element is that they suggest that for "small" designs (think iPhone or Watch) the functionality of the Non-System Memory Gateway can be placed in the Memory Controller. The advantage of this is that the NSM GW can thereby share the Network Interface of the Memory Controller rather than requiring its own interface.

Another recent patent, (2024) https://patents.google.com/patent/US20250097166A1 *Communication Fabric Structures for Increased Bandwidth* fills out some details. Consider first


Agent 120a
Agent 120b
Agent 120c
Agent 120d
I/O I/F 125a
Network Interface 101a
Data Transactions 140
NS 110a
Communication Network 105
NS 110b
Data Transactions 144
Network Interface 101b
I/O I/F 125b
I/O I/F 125c
Agent 120f
Agent 120g
Agent 120h
Agent 120e

What this diagram is trying to show is that, when it make sense (certainly for Watch, probably for iPhone) the Network Interface hardware can be shared across multiple clients. In particular although the previous patent suggested that every different IO block might have a separate Network Interface which handled both buffering/arbitration and protocol translation, at least in some circumstances it makes sense to split some of the protocol to a separate small block and use common buffering/arbitration for multiple clients.

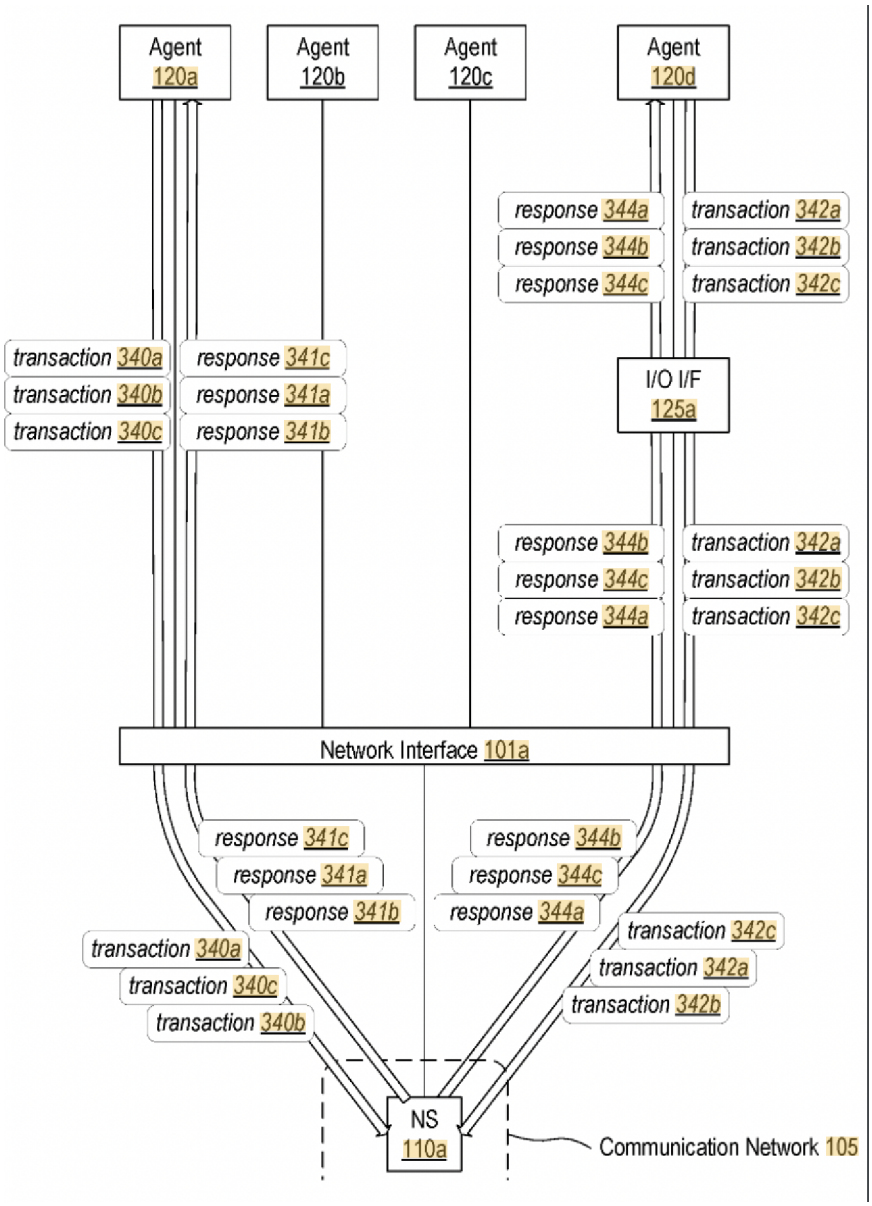
Next consider



This shows how the Network Interface and IO interface perform their translation roles.

So far these two diagrams simply expand on what we've already seen.

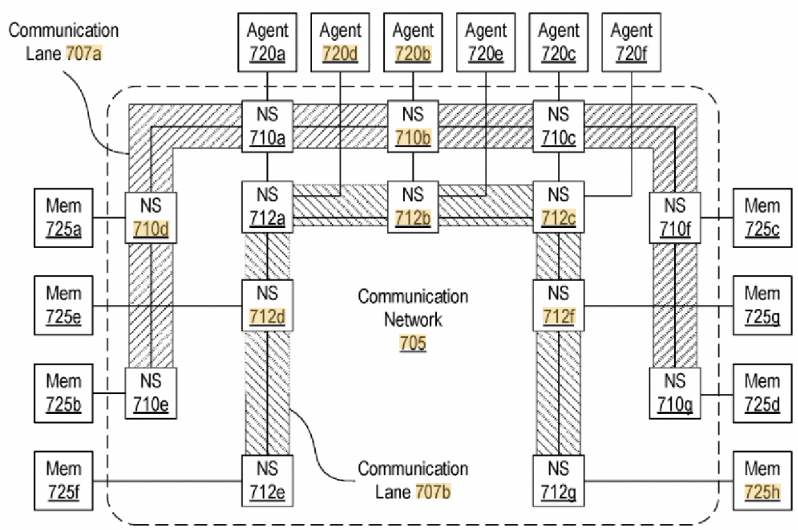
The part that's new is


Agent 120a
Agent 120b
Agent 120c
Agent 120d
response 344a
response 344b
response 344c
transaction 342a
transaction 342b
transaction 342c
transaction 340a
transaction 340b
transaction 340c
response 341c
response 341a
response 341b
I/O I/F 125a
response 344b
response 344c
response 344a
transaction 342a
transaction 342b
transaction 342c
Network Interface 101a
response 341c
response 341a
response 341b
response 344b
response 344c
response 344a
transaction 340a
transaction 340c
transaction 340b
transaction 342c
transaction 342a
transaction 342b
NS 110a
Communication Network 105

The idea here is that Agent 120a doesn't care about ordering, at least for these transactions. It sends them out as 340 a/b/c, the Network Interface sends them out in an opportunistic order, so that they hit the rest of the network as a/c/b, and the relevant responses are garbled even further in the return. Meanwhile Agent 120d does care about the order of transactions, but that ordering is handled by the I/O IF 125a. The I/O IF sends the transaction 342 (opportunistically, but in this case the packets happen to flow out in order) to the Network Interface which again sends them out opportunistically, and receives responses opportunistically. But then the I/O IF reorders the responses from 344 b/c/a to 344 a/b/c to match the transaction order.

Why would this make sense? I don't know the details of things like PCIe ordering, but I could imagine something like this
- transactions targeting different pieces of hardware are nominally independent, so can be reordered BUT
- responses don't have a clean way of designating what they are a response to, so the only way you know what a response corresponds to is that the responses come back in the same order the transactions went out.
The scheme above allows for that, while also not forcing any unnecessary delays for "unnecessary" ordering. This would work if the Network Interface and I/O IF add tags to responses and transactions (tags that are not present at the PCIe level) so that the I/O IF can use those tags to re-order responses, before stripping them off (because Agent 120d speaks PCIe, not Apple NoC). Obviously this is all rebuilding, with appropriate modifications, elements of what we know from large-scale networks (TCP encapsulating IP encapsulating ethernet)!

The second part of the patent describes how to grow the network. Suppose the current NoC design uses a 64B wide set of tracks. (I've no idea if this is true, let's just assume it is.) If you want to increase the NoC bandwidth, one option would be to double this to a 128B wide set of tracks. But a second, more flexible, option is to create a second parallel NoC also 64B wide. This is more flexible in the sense that the two now have some independence, each can operate on different addresses simultaneously. This is the path Apple is taking.



The above shows how the two “lanes” are laid out.
The inner lane and the outer lane are mostly independent. There are three switching stations that can route between them (NS 710 a/b/c) but the other switching stations (like NS 712d) are only concerned with buffering and arbitration within their lane. This of course allows them to be smaller and simpler.

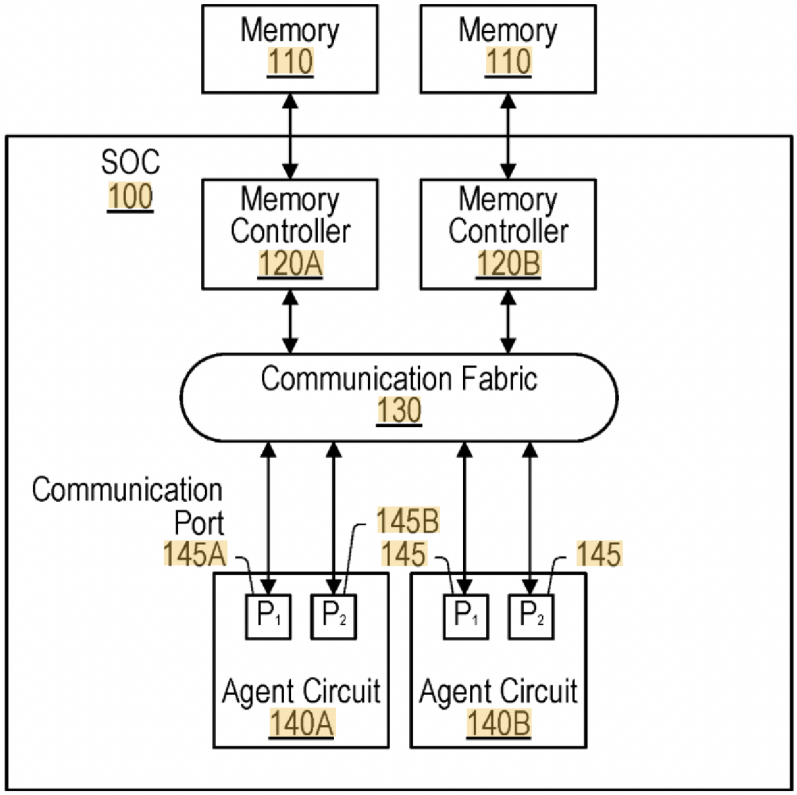
This joins a few other patents, for example (2023) https://patents.google.com/patent/US20250103117A1 *Power Management With Multiple Power Sources* which suggest that designs based not just on chiplets (as in the M1, M2, M3 Ultra) but are *asymmetric* chiplets (eg a chiplet for CPU, a chiplet for GPU, and a chiplet for IO) are being considered by Apple.

FWIW Qualcomm’s Falkor aka Centriq server chip (2017) used two bi-directional rings (each 64B wide). That chip supported up 48 CPUs (arranged as 24 clusters of 2 cores) but no GPU or NPU. So probably had requirements more or less as demanding as say a Max.
The way Falkor handled the two rings was that one transported “even address” data (even in the sense of the address of a 128B cache line), the other handled “odd address” data. This seems like a very Apple-like solution, but requires every ring stop (ie every IP block) to be connected to both rings. The scheme described in this patent seems more flexible in that it allows less demanding IP blocks to connect to only one of the rings. The QC solution may be better when you have a large number of identical high bandwidth IP blocks, whereas the Apple solution is better for a wide mix of IP blocks?

Cluster connection to the NoC

Something that’s been unclear from the start is some exact details of Apple’s NoC and how it delivers differential performance. For example, giving approximate numbers (and using M2 family because M1 used a different class of DRAM) we have that M2 maxes out at 100GB/s to DRAM, whether from CPU or GPU cluster.
M2 Pro maxes out at twice that, but a single CPU cluster still maxes out at 100GB/s. M2 Max maxes out at 400GB/s, attainable for GPU, but once again a single P cluster maxes out at 100GB/s. How can we understand this?

(2023) https://patents.google.com/patent/US12277074B1 *Mechanisms to utilize communication fabric via multi-port architecture starts to provide an answer*, and it's already visible in this diagram:



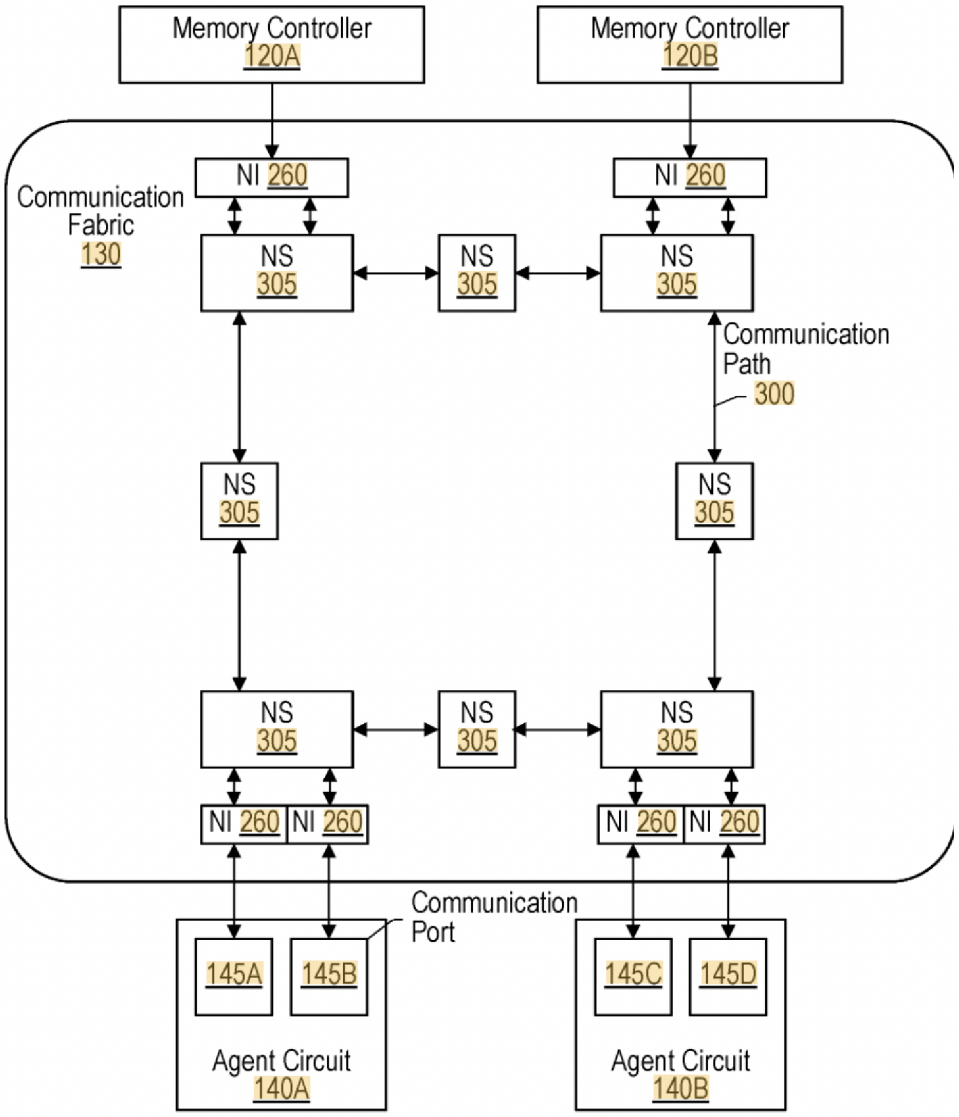
The significant point is that each (presumably just P) CPU cluster now has two ports (P1 and P2) connecting to the NoC, not just one. Presumably one port maxes out at something like 100GB/s (in the case of M2 DRAM, this will scale with M3 and M4, but not by much, just as DRAM bandwidth scales up, but slowly). Providing two ports gives us something like doubling the NoC to DRAM bandwidth available to the cluster.

A few obvious questions arise, some with answers in the patent.
- Why do this now? The answer is larger P-clusters. Suppose 100GB/s is “good enough” for a 4-core P cluster in the M2 generation. Now we both improve the P-cores (frequency and IPC) AND we move to 6 cores in a P-cluster. At that point perhaps simulations show that there’s a non-negligible performance boost to doubling the NoC?
The exact configuration of a cluster is a tricky point. There’s obvious value in sharing whatever you can (L2, L2 TLB and page walkers, AMX, page compression/decompression HW, etc). But there’s also a point at which sharing starts to overload a resource (as we see here with the NoC port being overloaded). You can do some duplication as long as it’s one or two resources that are overloaded, but at some point you might as well just split the cluster.
So does this mean we’ll see 8-core clusters soon?

- How is traffic split across the two ports? In the obvious way! Most transactions are DRAM-based (even if they will ultimately hit in the SLC) and those transactions already have the address hashed to decide which of the two or four (or eight for an Ultra) memory controllers they’ll route to. Essentially you choose the port based on that hash, so each port serves half the present memory controllers.
Some specialist traffic is targeted at an IP block, not a DRAM address (for example a SW interrupt and some snooping traffic). This is hardwire routed to P1. In theory this makes for slightly asymmetric usage and is slightly sub-optimal, but there’s probably not enough of this traffic to make any real difference.

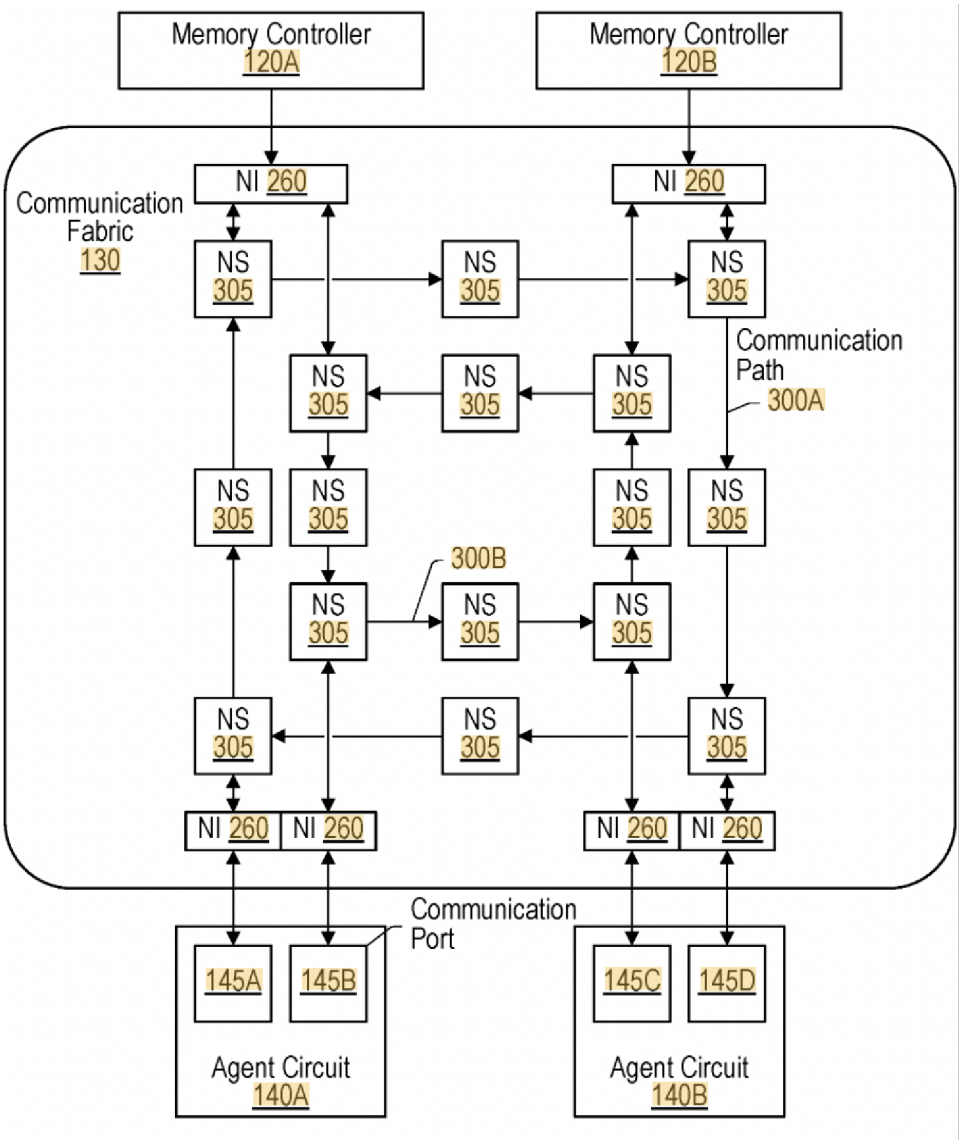
- How does the GPU do this? The patent doesn’t say, but my guess is that a single “cluster” of GPU cores (the full set in an M class) uses one port. A Pro essentially doubles this cluster of GPU cores, so picks up two ports. And a Max quadruples this cluster of GPUs and picks up four ports.
So we have something new in the patent in the sense that we have a single IP block (the CPU cluster) using two ports, whereas before every IP block (treating a “cluster” of 8..10 GPUs or so as a single IP block) had a single port.

We also get to see something of what the NoC looks like. A simple version (this is not the final version, though maybe it’s what we see on an A-class chip) looks like


Memory Controller
120A
Memory Controller
120B
Communication
Fabric
130
NI 260
NI 260
NS
305
NS
305
NS
305
Communication
Path
300
NS
305
NS
305
NS
305
NS
305
NS
305
NI 260
NI 260
NI 260
NI 260
Communication
Port
145A
145B
145C
145D
Agent Circuit
140A
Agent Circuit
140B

We see this design implements the NoC as a bidirectional ring. The apparently unused switches (NS 305) have connections like the ones we see to the CPU clusters, those connections going out to IO and other IP blocks, like media, ANE, GPU, etc.

If you have one ring, the next obvious step up, before you try something more complex, is two rings.


Memory Controller
120A
Memory Controller
120B
Communication
Fabric
130
NI 260
NS
305
Communication
Path
300A
300B
Communication
Port
145A
145B
145C
145D
Agent Circuit
140A
Agent Circuit
140B

This is apparently where Apple is today. Note that the first ring was bidirectional (which doesn't help with throughput, but can help with latency). The two rings are unidirectional in opposing directions, so by routing to the appropriate ring we should not see a latency drop, but two rings does provide doubled bandwidth.

There are still a few obvious questions about the NoC, but at least we now know a little more.

Power

There are always power changes, and most are very technical. But here are two fairly easily explained ideas:

(2024) https://patents.google.com/patent/US20250104742A1 *Adjustable Clock and Power Gating Control* starts by pointing that power control for DRAM (and related items like the memory controller and the fabric link to the memory controller) are more difficult than for most other items. Memory activity can be bursty, and there can be a long delay associated with memory sleep/wakeup or DVFS transition (since Apple can't control DRAM specs).
The essence of that patent is that each IP block has now adopted a uniform "vocabulary" with which to indicate to the central power manager its current and future expected memory usage. This allows for more insight than basing decisions purely one what memory activity has looked like over the past 1000 (or whatever) cycles.
The vocabulary can include things like current level of usage, and what expectations the block has for future usage. Obviously, then, the controller can make more informed decisions than if it were deciding based purely on a request from the memory controller based on recent memory controller history.

A slightly different way to look at the problem is (2023) https://patents.google.com/patent/US20250093932A1 *Memory hierarchy power management*. Now the question is: suppose I am willing to spend one extra unit of power. Is that power best spent on running DRAM faster or running the CPU faster? Obviously this depends on the type of code being executed, but it also depends on the relatives power costs and performance benefits of incrementing DRAM speed or CPU speed to the next available level. The patent describes what statistics are collected (so you have an idea of performance, power being used by each item, and the sensitivity of performance to changes in CPU or DRAM frequency) and how to choose the optimal tradeoff.

Process

Transistors between metal layers

One consequence of process and packaging tech evolving so fast is that there are probably multiple good ways in which we can extend existing tech, but which haven't yet been thought up (and which may not be, before we move on to the next stage of process evolution!)

We have seen one version of this in the idea that power delivery benefits from a distributed reservoir of capacitors (and to some extent inductors), and in response we have seen the construction of capacitors wherever they can be slotted it without cost – any free space on the SoC can be processed to be a capacitor, likewise any free space between two chips mounted on each other.

(2024) https://patents.google.com/patent/US20250105134A1 *Buffer and Inverter Transistors Embedded in Interconnect Metal Layers* is a similar sort of lateral thinking.
Traditionally chips have been made by fabricating a layer of transistors, then a few (initially very few) layers of metal wiring, the so called front end and back end processes. Over time the transistors have, of course, become vastly smaller and more sophisticated, and many more layers of metal routing have been added, but we still operate in this basic way.
Do we have to? Suppose we didn't?

In particular, Apple pose the question: what if, towards the end of the creation of the stack of wiring layers, we diverted the wafer back towards transistor processing and laid down another layer of transistors? Assuming this is feasible, and not too expensive, why bother?
The issue is consider sending a signal over a long distance of the chip. Right now doing this either requires that we "launch" the signal at high voltage (so that it maintains a tolerable SNR after the resistance of the long distance wiring) or we require that the signal periodically have to be diverted from the long distance wiring layers (at the top of the wiring stack) down to the transistor level to be regenerated then sent back up to long distance layer. The first burns excess energy, the second requires some wire cluttering (to periodically move the signal from transistors to long distance and back) and some wasted space at the transistor level (area spent in signal regeneration rather than in logic).

A natural question to ask (perhaps motivated after you start thinking about how to redesign a chip given the new facility of backside power delivery...) is "why exactly do we have to have those regeneration transistors laid down in the front end layer?" The signal regeneration transistors need some power, but they don't need much connection to the rest of the SoC, and they don't need the fine lithography or other fanciness of the front end; what if they could be slapped down using a fairly crude process between two of the upper metal layers?

Is this actually feasible? Presumably Apple will (has already?) discuss it with TSMC and one day we'll know.

Exotica

Very large systems

Most Apple patents fit into what we already know, and we can be fairly confident that they correspond to real products; in some cases the correspondence is as obvious as relevant new APIs that expose the

new functionality.
But every few months throws up some Apple patents that have much less connection to reality, and who knows how they will ultimately play out! The most interesting of these is the fairly long running series that considers ways to grow the M-series SoCs. Obviously we have seen two-chiplet versions of this design, and just as obviously we have not seen anything larger. Meanwhile earlier patents I've referenced have discussed
- ways to grow to four or more chiplets (the infamous M Extreme)
- using an Ultra (ie a pair of chiplets) as a basic unit that's then assembled into larger designs
- ways to add more memory to these sorts of designs, given how we run out perimeter as we add more compute chiplets side by side (for example by "spurs" jutting out the sides of the design.

None of these have seen reality (yet? outside Apple?) but parts of Apple still have grand ambitions, as seen in (2024) https://patents.google.com/patent/US20250094093A1 *Optics-Based Distributed Unified Memory System*. In some ways this is a rethinking of the whole project. We ask three questions
- where is Apple's design intrinsically superior to competitors?
- what can be done if we take chiplet decomposition seriously?
- how large do we want to go?

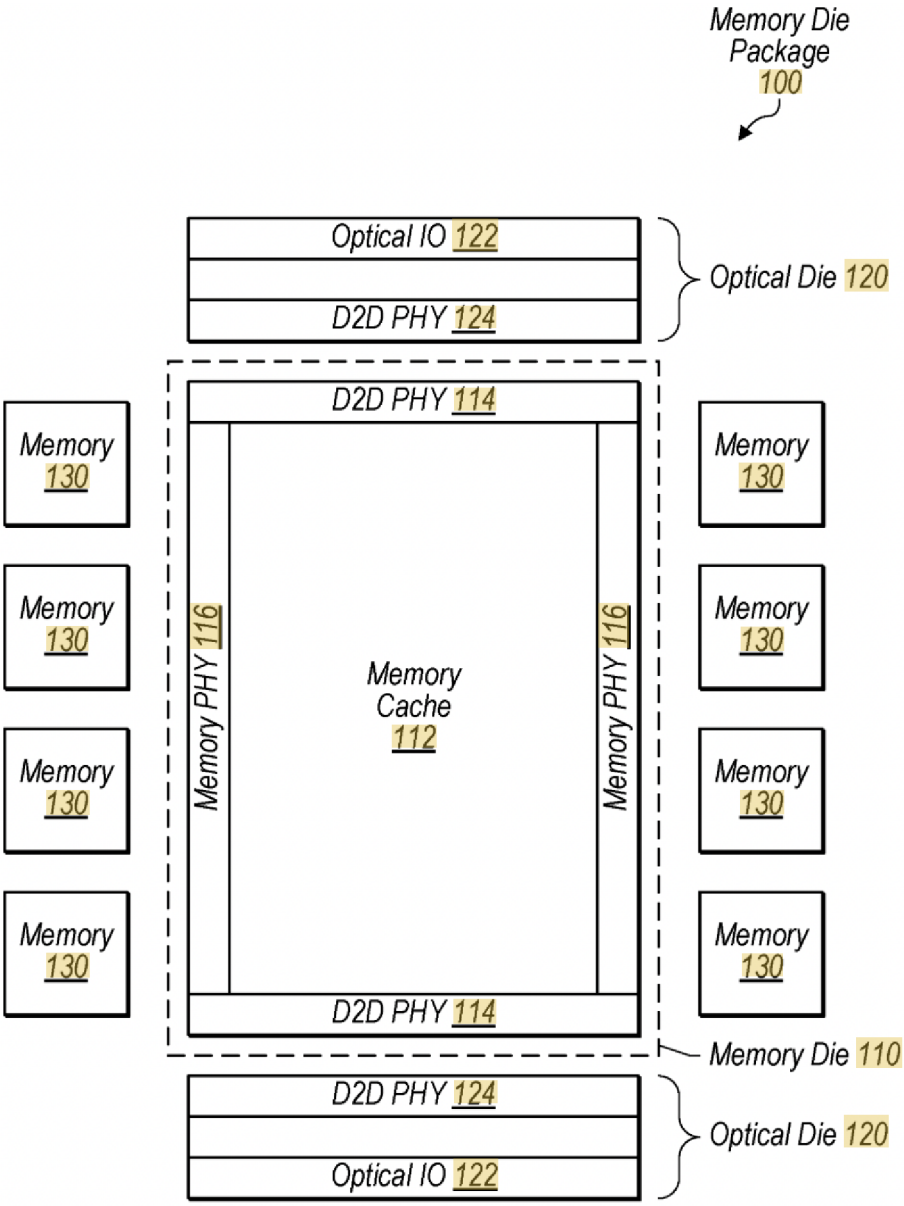
The answer to the third question is that nVidia provides systems like DGX that are massive (in size, cost, and capability) and can't manufacture them fast enough. So clearly there's a market, even if it's only for Apple internal use, for that sort of size.

The answer to the first question is that Apple provides Unified Memory. We tend to think of this as mainly meaning that the CPU can construct a complex pointer-based data structure which can then be handed over to the GPU and utilized without a problem, ie "transparent pointers", and that's part of it. But another part is that Apple provides a single pool of DRAM that can be used by all clients, you don't land up with DRAM stranded on the GPU card (or whatever) that's unavailable to a large CPU task. The extension of this is the problem current rack systems have where the largest they can "naturally" grow is something like a dual socket system with a single DRAM pool; beyond that the DRAM associated with a second socket pair is not available to the first socket pair. CXL is supposed to fix this in the server space, but like anything consortium related is taking forever to arrive and may be disappointing when it does arrive.

Put these together, and Apple sees an opportunity for a large system (rack, to multiple racks) sharing a common physical memory pool (no wasted memory when one set of compute chiplets can't access DRAM associated with a different set of compute chiplets) and a common address space (making large tasks, eg LLM training, that much easier).

So how to actually achieve this? That's where taking chiplets seriously comes in. The starting point is this, which makes sense as soon as you see it. It's what you'd get if you took a Max chip and removed everything not related to DRAM support. So it's a huge pool of SRAM (~1GB) in the middle, with memory controllers connected to maybe 64 8GB DRAM chips providing 512GB of DRAM at ~1TB/s bandwidth.

The D2D Phy is the standard Apple UltraFusion connector providing 2.5GB/s connectivity to another chip, in this case an optical die.
The optical die is two layers (this is standard for this sort of hardware), a logic layer created with an advanced process, connected to an optical layer created on a specialty optical process and which ultimately provides and modulates the laser(s) and connects to the fiber optic cable(s).

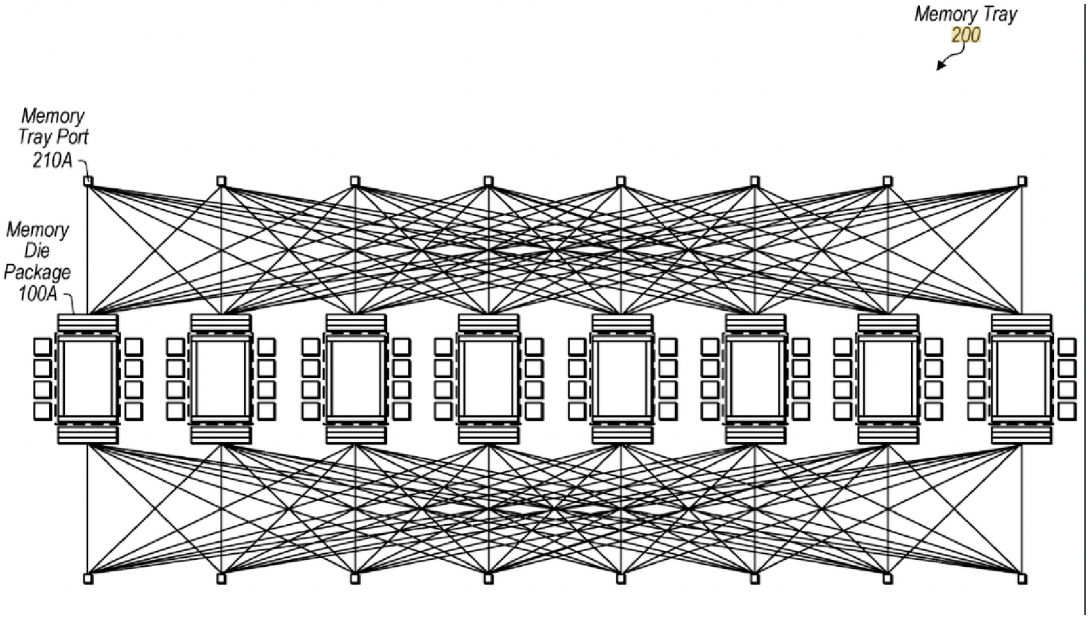

Memory Die Package 100
Optical IO 122
D2D PHY 124
Optical Die 120
D2D PHY 114
Memory 130
Memory PHY 116
Memory Cache 112
Memory PHY 116
Memory 130
Memory 130
Memory 130
Memory 130
Memory 130
Memory 130
Memory 130
D2D PHY 114
Memory Die 110
D2D PHY 124
Optical IO 122
Optical Die 120

So already we see a few things. One is that Apple appears to view UltraFusion as a generic chiplet interconnect, like UCIe is being used by the rest of the industry.

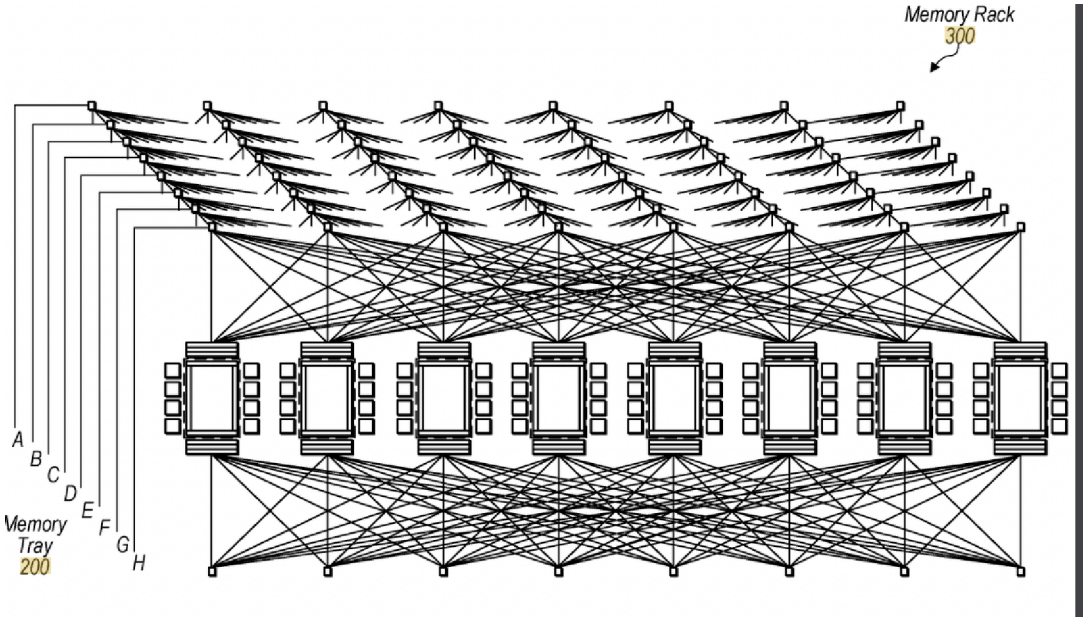
One possibly problematic issue is that they envisage using *directly modulated* lasers to generate the optical signal. There are two ways to modulate a laser signal. One is to directly vary the electrical voltage/current fed to the laser (direct modulation), alternatively we can send the laser through a modulator (some sort of material that, in response to an electrical signal, either transmits the laser light or absorbs it). Direct modulation seems more obvious, and it requires less hardware so it's smaller, but the belief in the past has always been that it's less reliable, in the sense that rapidly modulating the laser electrical this way reduces the laser lifetime. That might be acceptable in a telecom environment where the lasers are field-replaceable and someone comes around every week to swap out all the equipment that died over the last week, but it's not great for a chiplet living in a rack! But maybe the state of laser reliability has improved a lot recently?

The optical chiplet is then connected to fiber(s) which can connect to something else and (abstractly) we have a block of fast 512GB DRAM available at the end of some fiber.
So what can we do with this? Obviously we hook these up to create a larger version of an M SoC, but what are the details? Among other things the patent implies the familiar
- pages are spread across multiple of these memory dies, so the "unit of splitting" is smaller than 16kiB.
- at least two distinct optical networks connect to each memory die; one a coherent network (used by the CPUs, and maybe everything else), one is a relaxed ordering network (used by the GPU).

We collect these memory packages into a 4TB memory tray (with 8GB of cache) connected to 16 ports (which, recall, are going to be somewhere all these fiber optic cables converge).



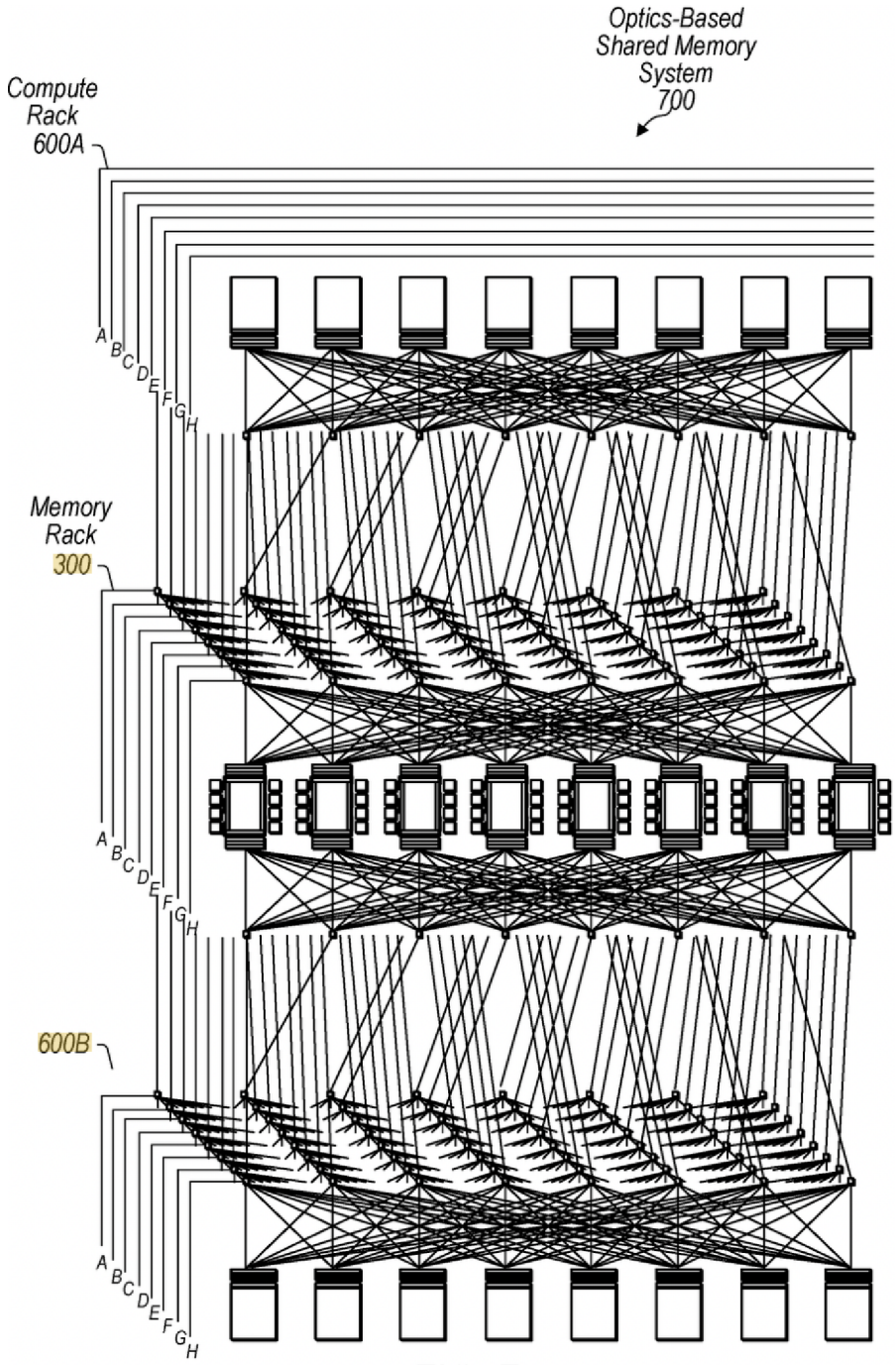
But come on, all this gives you is a pathetic 4TB of DRAM. Surely you want more? Never fear, you can create a rack of DRAM, for example holding 32TB.



Meanwhile we also have compute dies (which could presumably house some combination of 0..*N* CPU clusters and 0..*M* GPU cores), also connected by UltraFusion to an optical die. The compute die Apple talks about is 18 CPUs and 40 GPU cores which seems about a Max size (maybe four 4-P clusters and a 2-E cluster? maybe 2 6-P clusters and a 6-E cluster?). Probably some IO and display stuff, and the memory support removed, so smaller than an existing Max?

Again multiple compute packages can be placed, 8 or so, on a compute tray, all connected to 8 compute tray ports, and since nothing is stopping us, we can stack up 8 of these or so to give us a compute rack

End result of all this lunacy is we have 2×1152 CPUs and 2×2560 GPU cores connected to 4TB of DRAM (and 64GB of SRAM cache) as below



The big question is how does the switching happen?

Memory requests that don't hit in cache on the compute die can route to the attached optical die which figures out the target tray and package and sends the signal down the correct fiber. To do this directly would

require a lot(!) of point to point fibers, so there's a specific wiring pattern that connects each compute port to an appropriate memory port. Like any NoC, getting to your final destination will require a few hops, but the switching is distributed over the optical dies attached to each compute and memory package, rather than form a single central switch.

The patent says at this point, in a droll fashion, “Also note that each compute die package 400 can observe the same latency and bandwidth characteristics to main memory. *In doing so, node 700, in essence, is the largest UMA machine ever to have been designed*.”

So what we have is each compute rack has 64 ports, while the memory rack has 64 north-facing ports and 64 south facing ports. Each compute port is connected to its matching memory port via two fibers (one for transmit, one for receive). This forms an array of 128 fibers which is claimed to fit into about 2 cm wide! (Each fiber is only about .5 mm across)
If this wasn’t enough, we end with some details about how you could connect say four of these racks together.
more economics than anything else.

Will any of this ever happen? Who knows?
Optics at this sort of level is very much leading edge, meaning it works in the lab but going from lab to volume production remains a task of some difficulty! It’s fine to talk about all these ideas, but implementing them is far from easy. There is a reason that, for example nVidia has avoided optical for as long as possible, so that Blackwell is still copper.
But things are changing, and even nVidia is moving, however reluctantly, to optical for its next generation of large interconnects. So maybe the time is right for this tech, and it makes sense for Apple to suffer (and learn) at about the same speed as nVidia?

It’s probably significant that Apple is starting to accumulate patents in more technical areas of photonics, for example (2024) https://patents.google.com/patent/US20250110272A1 *Photonic switch with multi-wavelength routing capabilities*, which appears to be a scheme for an Mach-Zehnder *type* splitter that’s hopefully a little smaller. (The two most common ways to separate a fiber WDM signal, ie two or more signal-carrying wavelengths multiplexed on a single fiber use Mach-Zehnder or micro-ring resonators. Traditionally MZ has been bulky, while MR has been extremely temperature sensitive. Apple’ scheme is MZ-like in that it’s using the accumulation of phase differences along two distinct optical paths, but the phase differences are created by using appropriate materials rather than raw distance.If Apple can get this MZ-like scheme small enough to compete with MR, at least for the sort of rack-sized system they have in mind, that’s definitely helpful.)
There are many more in this vein...

spectroscopy

(2024) https://patents.google.com/patent/US20250109989A1 *Optical measurement systems with multi-wavelength emission*. Apple have been interested in spectroscopy for years. The dream of a tricorder (eg point your phone at something and get some sort of chemical analysis; or variants like non-contact glucose monitoring) remains strong. For a while they appeared to be working with Rockley Photonics, until Rockley imploded.

Similarly (2024) https://patents.google.com/patent/US20250110043A1 *Optical measurement systems and methods*.

The dream remains alive. Target product? Who knows?

phased array antennas

The Starlink antenna is a remarkable device consisting of over a thousand small antenna's all "printed" on a single "circuit board" and controlled so as to achieve remarkable things (like flexible, fast tracking of multiple signals).
One suspects that as soon as they became available, a whole lot of people started saying "duh" and realizing what's possible if you rethink antennae given the abilities of current digital control logic.

(2024) https://patents.google.com/patent/US20250118899A1 *Electronic Device Having Monolithic Phased Antenna Array* seems to be Apple's entry into this space, in this case creating the antennae through metalized cavities in silicon.
The Apple scheme described in the patent is pretty much the simplest scheme you could imagine. But presumably it will grow, especially now that Apple controls the associated modem.

attention free transformer

Even if you don't much about LLMs, you've heard that the Transformer architecture changed everything, really kickstarted current AI; and that Attention is a significant part of the Transformer architecture (as in the infamous "Attention is all you Need" paper).
However attention is expensive in compute, and even if you reuse previous compute (using a KV-cache, another frequent buzzword) the size of the KV-cache can be substantial.
Can we achieve the equivalent of attention (ie using previous words to shift the embedding of later words) without the matrix-multiply costs of attention?
(2021) https://patents.google.com/patent/US12271791B2 *Attention free transformer* describes one way of doing this. In a rare case for Apple, this is accompanied by a paper, (2021) https://arxiv.org/abs/2105.14103 *An Attention Free Transformer*, which is probably easier to read if you want to understand the details.

AI moves very fast, so the specific details of this idea are perhaps already obsolete, replaced by papers like (2023) https://arxiv.org/abs/2312.00752 *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*, which take the idea and build upon it.

There are other Apple investigations into ML related areas, for example fancier versions of the construction of a full-body moving avatar from video or a point cloud (2024) https://patents.google.com/patent/US20250148678A1 *Human subject gaussian splatting using machine learning*.
As always, who knows what the goals are or when they'll ship.