Everything Else

The previous volumes discussed stuff that was ultimately (more or less) validated by actual experiments.
Everything below is essentially my exploring the world of Apple patents and trying to understand them. I probably missed some important patent, misunderstood many of those I read, and even when I understood them correctly, the idea may either already be obsolete or may not (yet?) be implemented. But as long as you bear all that in mind, there's a lot of interesting stuff in the patents!

Close to the Core

memory barriers

The issue of barriers is one worth considering.

There is a general pattern one sees repeatedly in computing. We start with operations that are in-order and synchronous. (In order CPU, or synchronous disk writes to a floppy disk.) To improve performance, we convert these operations to asynchronous and allow them to be re-ordered. Which is OK, except that there are always special cases where we need to preserve ordering.

In the case of IO, this happens for databases (including the file system as a whole). To maintain transaction guarantees (ie either all of a set of changes get persisted to disk, or none get persisted) even in the case of power failure partway through the transaction, databases do things like write an initial outline of the changes that will be made to a log, then make the actual changes; such that if there's a failure in writing the actual changes, the transaction can be either wiped from the database, or reconstructed, using what was previously (by definition) written to the log.
Obviously for this to work, the changes to the log need to be persisted *before* the actual changes!

In the case of CPUs, an example is two processors communicating about shared data. Again one wants transactional-type interactions:
- I make a set of changes to the shared state,
- then I flip a variable that says "my changes are done",
- you see that changed variable,
- and read the changes I made.
Sounds good – but again it only works if the system does not re-order memory transactions.
I need to be sure that once you see the signal variable has changed, any reads you make of the vari-

ables I modified will be of my *changed* values.
(This is not the same thing as cache consistency.
Cache consistency is about a single memory address holding the same apparent state for all CPUs.
What we are discussing is the relationship between changes to two or more different memory addresses, and the order in which I make them vs the order in which you see them.)

Confronted with these types of problems, there is a standard (far from optimal!) solution which always seems to be the first solution attempted. That solution is to add some sort of "sync" operation to the IO API, or to the CPU.
The idea is that a database will make a set of changes (eg writes to the log) then issue a sync command, then make the next set of changes (eg changes to the actual database file). The sync will force out the first set of changes to storage before the second set even get started.
You can imagine similar versions of this for inside a CPU, a sync that forces all pending loads and stores within the LSQ to commit all the way to the L1D before the CPU can continue. (And in fact Intel have something like this that flushes out cache lines all the way to DRAM for supporting Optane DIMMs, specifically for the database problem we already described).

This sounds good at first, but the problem with sync is that it does far more (and so is far more expensive) than you really need. All you actually want is that the *first set of transactions* must all occur before *the second set of transactions*. You don't care about *when* the transactions occur, and you have no particular desire to make the whole world wait while you flush all sorts of pending disk sectors or loads/stores. or cache lines, out to a much slower memory/storage tier.

The next level of doing things is barriers. A barrier implements the distinction we described; it says that things that happened before the barrier must complete before those after the barrier, but no more than that. Apple (but not Linux) provides barriers of this sort for IO, and ARM provides them for memory operations. (x86 kinda sorta provides them, but also makes sub-optimal ordering guarantees in the memory ordering model, and the whole thing is a mess).

There are different ways you can implement barriers (including treating them as flushes), but the obvious sensible way is to
- mark requests as they come in via some tag that changes when a barrier is encountered, and
- not allow any request with a tag later than N to complete until all requests tagged as N are complete.

Apple have a patent for this (2012) https://patents.google.com/patent/US9582276B2 *Processor and method for implementing barrier operation using speculative and architectural color values*.
Even with this idea there is something especially cute about Apple's implementation. The obvious place to implement such a barrier, at least to me, is in the LSU, so that it controls that everything is correctly written out to L1. But what Apple appear to do is implement the barrier between the L1 and L2, ensuring that the outside world (ie L2 and beyond) see correct ordering, but allowing more flexibility, and thus more performance, in how the operations occur between LSU and L1! I assume this ultimately controls the ordering of MESI state changes and response to snoops, which is closer to what

you *really* want. (Ultimately you don't really care how the loads and stores are ordered to your cache, what you care about is how the ordering associated with those loads and stores is conveyed to the other devices in the system.)
This sort of generational segregation is the best one can do if barriers are your API or ISA, but you can do better!

The problem with barriers is that they still state more than you actually want! You don't actually care about *every* request before the barrier completing earlier than *every* request after the barrier; all you care about is your particular requests (your stream of database changes, ignoring all the other IO on the machine; or the particular changes you make to a shared structure, ignoring all other load stores that happen as you read/write your private storage in the process of changing that shared structure). What you want is a way to tag the specialized requests, and then execute a barrier that only applies to your tagged requests.
This is in fact the IO API that Apple provides (IO tagged by a particular file handle). And it is something that ARM is investigating as part of their set of instructions for ordering cache lines out to persistent storage (ie Optane-type DIMMs). https://community.arm.com/developer/research/b/articles/posts/relaxed-persist-ordering-using-strand-persistency.

barriers in PIO

Once you understand this pattern, you see it everywhere! Consider, for example, this, apparently unrelated patent (2009) https://patents.google.com/patent/US8032673B2 *Transaction ID filtering for buffered programmed input/output (PIO) write acknowledgements*.
The issue is communicating with peripheral hardware (which can things like sensors, IO equipment, radios, whatever). Naturally for large data transfers one wants to use DMA, but these devices usually need to be configured at boot time (and, much more frequently, maybe every time there is a sleep-/wake transition), and that's usually done by PIO, ie writing configuration values to what look like memory addresses but which are actually routed, eventually, to registers in the peripheral.

The obvious way to do this is synchronously, to write a config value, check there was no error, write the next value, and so on. But this is a slow process with long waits between each write.

Slightly better would be to have the PIO controller buffer the writes in some queue and feed them on to the device at whatever speed it can handle. This at least prevents the CPU from having to wait a long time – but it still means each peripheral is brought up sequentially, so there's some delay until all the peripherals are ready. However the PIO controller can't randomly reorder the PIO transactions because they are designed to happen in a particular order, one item of functionality being brought up at a time! So we have the same sort of issue as above – what we want is strand ordering, so that we can label all transactions to a particular peripheral and retain the ordering of those transactions, while allowing transactions to other peripherals to be interleaved – each peripheral gets its instructions in order, but the PIO controller sends them out to the peripherals as soon as any device is able to accept a new

transaction.
And, in essence that's what the patent is about, a PIO controller, and a labelling of PIO transactions, that allows for reordering between strands but not within strands. (At least this is my interpretation. The patent talks about the transaction ID as being a source ID, but to me that makes no sense; it makes more sense as a destination ID.)

fusing barriers with load/store ops

A different way to optimize barriers is seen with (2010) https://patents.google.com/patent/US20110208915A1 *Fused Store Exclusive/Memory Barrier Operation*. The details are doubtless obsolete, but the big picture intuition remains.
The problem to be solved is that a common OS pattern involves a spinlock (ending with a Store Exclusive operation and a branch loopback if that store failed) followed by a memory barrier to "publish" the changed state of the spinlock. The intuition is that both operations (the store conditional and the barrier) are expensive because they require a trip out to the *point of coherency*, think the last level cache. The solution is to fuse them together to create a single op (understood by the caches and the fabric) that only requires one such expensive outward trip.
(In a way it's very similar to the techniques used by the internet protocol SPDY to try to piggyback as much connection setup as possible into just one or two packet exchanges.)

Equally interesting is when you consider some details this implies.
- One is that the memory barrier, at the point of the fusion, is probably speculative. When you define the precise semantics of this fused operation, you have to be sure that it behaves correctly, in that it doesn't do anything that's incorrect if that speculation turns out to be wrong.
- Secondly, one thing this implies is that in response, even to speculative barriers, and even at early detection of such barriers (as early as in Fetch...) one could start performing whatever "push" operations might be required to force out various cached data as required by the barrier. Such early detection and execution somewhat ameliorates the pile-up of such scheduled work that can be caused by such barriers; and it's feasible as long as all that's done is essentially pushing out cached state that's going to be pushed out anyway at some point.
- Third is that the operations being fused are separated by at least one instruction (a branch) and possibly more than one (cleanup after the spin loop). One thinks of instruction fusion is something performed in Decode by observing back-to-back pairs of instructions, but here the fusion has to happen at a later point; the patent suggests performing the fusion in the LSU in which case it will happen so long as the two operations are not separated by an intervening load or store operation.

Using colors to implement barriers obviates the need for some of this technology, but one place that seems like it might still benefit from these ideas (both early "push" and fusing expensive "far" operations) is TLB/page table manipulation.

inter-core memory ordering issues (poison bits)

We've described barriers and where to use them, but there are other unexpected issues for the CPU designer arising from multiple cores.
Suppose we have two loads, load A followed by load B, both loading from the same address. Remember that this is an OoO machine, so load B might execute first, with various other loads and stores occurring between the execution of load B and then load A.
What can go wrong?

First if there are no significant memory events between the two loads, who cares? They return the same value so let them happen in whatever order.

Second, the case we have already discussed in great detail is where there is a store C occurring in program order between A and B. We know how this is handled (that's what the load-store queue is all about) and we know the basic idea: when load B executes, it will check the various stores in the store queue, see that store C affects its address, and wait until store C executes. Likewise store C sees that it has to wait until load A executes. Things can go wrong at a mild level (slightly unfortunate timing) meaning that some of these instructions "collide", the collision is detected, and they have to replay. Or things can go very wrong (some of the addresses required are not available and are misprinted) in which case at the point where an affected instruction retires, the misprediction is detected, everything is flushed, and we restart at that point.
All covered before.

But there is a third thing that can go wrong! What if the following happens:
- load B executes
- because of activity in some other core, the relevant cache line is replaced in the L1D
- load A executes – and loads different data because it's loading from the changed cache line, storing changes made by another core

Note the issue here.
The issue is not one of barriers – we are making no claims about how one address has changed relative to another.
The issue is one of timing – load A is supposed to occur before load B, and we have broken that promise.
Yes, we broke it in a stupid way, because A sees "new" data as changed by some other core, whereas B sees "old data" before the other core changed it; but that just makes things worse – our program doesn't expect time to jump backwards between two loads!
So there is an rule in the memory model that seems so dumb that you hardly need to say it, but it's needed for cases like this – an older load can never see newer data than a younger load.

This is handled by yet another concept! Every load has associated with it a "poison bit", and if the cache line from which the load was read is modified before the load becomes non-speculative, then the

poison bit is set and we need to recover (which might be possible by Replay, or might require a Flush). This is described in (2013) https://patents.google.com/patent/US9383995B2 *Load ordering in a weakly-ordered processor*; which is followed up by (2016) https://patents.google.com/patent/US10747535B1 *Handling non-cacheable loads in a non-coherent processor*, which generalizes the idea.
(It should be obvious that this mechanism is rather coarser than strictly necessary, but this should be such an uncommon case that correctness is what matters, no point in spending extra resources to make it finer-grained and faster.)

energy reduction when using poison bits

Obviously these load and store queues, specifically the comparison of a new load or store against all the other items in one or other of the queues, eventually costs energy, no matter what smarts we use. However some of these tests are sufficiently rare that we can often avoid having to perform them. The above poison case is an example. Once a load queue entry is poisoned, we have to test all subsequent loads against the load queue to check that we aren't hitting a poisoned entry. But this is an unnecessary waste of energy in the usual case that nothing in the load queue is poisoned. Thus the LSQ tracks a single bit as to whether any entry in the queue is poisoned, and behaves differently in these two cases.

This is described in (2019) https://patents.google.com/patent/US20200264888A1 *Content-Addressable Memory Filtering based on Microarchitectural State*, which includes a few other similar such energy saving possibilities (for example tracking a single value representing something like "oldest age of the valid loads in the queue"; if a store address now comes in that is older than this age, then there is no need to probe it against the load queue for collisions.
The common theme in both of these, and some additional ideas, is to find a way to collapse the usual state of the queue into a single value than can be tested, rather than having to test every entry of the queue.

L2

L2 snoop filtering of L1's

The first patent giving us some feel for the L1 caching setup is (2006) https://patents.google.com/patent/US7752474B2, which suggests about as simple a (snooped, coherent) cache as you could imagine.

The patent is mainly interesting insofar as it compared with (2010) https://patents.google.com/patent/US9317102B2. By this second patent the following improvements are visible

- we now have a Coherence Point on the fabric. For the earlier design, all snoops routed directly from one core to both L2 and the second core. Now the L2 holds duplicate tags for the L1's of the various

cores attached to it.
The obvious consequence is that a snoop from one core can be handled by the L2 (which can filter out most snoops as being irrelevant to any other core, and so can discard them); this saves a little energy and maybe even allows for omitting some of the extra snoop handling machinery from the L1 (if L1 snoops are now rare, maybe it's OK to handle them via a slower path?)

- a second improvement is that the L1 now has various counters of things like how many valid blocks and how many modified blocks it contains, and presumably these counts can be used to inform decisions as to how rapidly to allow the cache to be put to sleep.

One problem I saw with the 2006 design was that flushing the cache seemed to require cycling through every way of every set (presumably one per cycle) asking that way to flush itself if it is modified. I don't see much of an improvement to this scheme in 2010. Of course you can exit slightly earlier (at the point where the count of modified lines goes to zero) but beyond that the same mindless walking through every lineID seems to be necessary.

L1/L2 snoops and energy saving

A less obvious benefit of having L2 performing snoop filtering is that snoops can be handled by the L2 while the L1 sleeps, and mostly without having to wake the L1. But this is a somewhat tricky business.
For super-short naps (rest of CPU is working, but no L1 load/stores this cycle), you'll probably want to save power just by freezing the clock going to the cache, but you'll still be paying leakage power.
For slightly longer sleeps, the CPU might sleep but you don't yet want to pay the cost of losing your cache data, so you allow the cache to remain powered up (just not clocked, probably also undervolted). Under these circumstances, you *really* want the L2 to perform snoop filtering so that the CPU and cache don't have to be woken up.

But at some point the sleep looks like it's lasting long enough that we might as well flush the L1D (ie write out all the modified line) and cut power completely. And at this point we don't want to wake up the CPU to flush the L1 cache lines; that's definitely not ideal.
So by 2013 we have https://patents.google.com/patent/US20140195737A1. Now we have a separate asynchronous engine that allows the main CPU to partially power down while the engine copies all modified lines out to L2, before fully powering down the core.

This might seem an obvious addition, but it's not as simple as you might think because of the eternal problem of coherency. Consider, for example, what happens if you receive a snoop on one of the modified lines that you haven't yet written out...
You can't *just* walk through all the modified cache lines, you have to maintain and handle some minimal level of snooping by the L1D and async engine right up till the moment of power down.

Interestingly the flush engine is associated with the L2 core rather than the L1 cores. Thus, in a sense, it pulls modified lines from the L1, it doesn't push them to the L2. (This is a nice version of the point I have stressed a few times, the area benefit Apple gets from clustering CPUs, and providing complicated logic at the cluster level so that it's shared by all the cores of a cluster.)
How does the engine know what lines to pull? Because (as mentioned) the L2 maintains duplicate tags for the lines of all the L1's, which it also uses as a snoop filter!

There are two further tweaks you can add to this system.

The first is that if *all* the clients of the L2 go to sleep, then after a while it makes sense to also shut down the entire L2. Once again you have the issue that you need to flush all the modified lines up to SLC. So the flush engine gets repurposed for that job, now walking the tags of the L2 looking for modified lines, until it has matched the count of modified lines that has been maintained, now pushing out lines rather than pulling them in.

The second tweak is that once an L1 client has lost power, there's no point in snooping for that client any more, it's like that cache does not exist! So Apple pulls power on the duplicate tags that were being maintained for that client: (2013) https://patents.google.com/patent/US20140189411A1 *Power control for cache structures*.

drowsy L2

The L2 as a whole is managed similarly to but with finer control, a so-called drowsy cache: banks that haven't been accessed for some time are put to sleep, with enough voltage to retain data, but requiring a cycle or two to wake up on access.
The unit that is actually put to sleep is a way in each set. Imagine that the L2 were, say, 8-way set associative; ten each set holds 8 ways. If, physically, the appropriate ways (way one, way two, etc) were each stored in one of eight independently powered banks, then one of those banks coul be put to sleep (thus making the set fully active for seven ways, while to access a line in the eighth way will take an extra cycle to wake up the bank).

A specifically interesting thing about how Apple does makes this choice to sleep is that they
- characterize the L2 by how leaky it. This will be one of those things that varies with manufacturing, some wafers just having all the transistors slightly better quality.
- measure the temperature just before the bank is to be put to sleep.

Based on the temperature and the leakiness of the L2, an idle count is established. The more aggressively the L2 is leaking current (sub-optimal transistors, or running hot) the lower the idle count is set. So you can think of it as rather than saying "we will wait 100 cycles of idle before we sleep a bank" it's like Apple is saying "we will wait 1 microjoule of energy wasted in idling before we sleep this bank". (2014) https://patents.google.com/patent/US9513693B2
L2 cache retention mode.

powering down a portion of L2

More aggressively, you can power down (rather than sleep) that whole bank and have the cache look 7/8th as large (as many sets as before, but only seven ways in every set). And of course you can take this further, eg powering down two ways, and sleeping three ways while three ways of every set remain active.
Like everything, there are complications to this once you start thinking about how to actually implement it! As detailed in (2014) https://patents.google.com/patent/US20150309939A1 *Selective cache way-group power down*.

Details have surely changed since then, but at that time Apple's L2 had 12 ways, and these were split into three separately powered collections, so that either one third or two thirds of the cache could be powered down.
However Apple still seems to operate L2 (and TLB2) at a 3-way granularity, that is each of these can run at full capacity, at 2/3, or at 1/3;s with the third or two thirds that is not “fully active” either sleeping or fully powered down. Hence L2 and TLB2 sizes being multiples of 3 (number of sets times some number of ways that’s a multiple of 3).

The first complication in this scheme is deciding whether to power down a third, and if so, which third? Remember that lines go into whichever was the least used way of a set, so that after some time the four least used ways of this set may be very different from the four least used ways of some other set...

There are two orthogonal sets of three counters. One set counts the number of references to each of the three physical banks. The lowest value tells us which bank will be powered down.
Separately we have three counters that count how many references have occurred to the four MRU lines, the 4 LRU lines, and the four midRU lines across all sets. If these indicate that usage is concentrated in one, and two, thirds of these three ranges, then (if other conditions are also satisfied) the decision is made to power down a third.
Once you decide which third to power down, you of course have to flush all modified lines before the power down. Is that all you do?
In principle you could imagine something like, for each set removing the least used lines from the way groups that will not be powered down, and copying across the most used lines from the way group that will be powered down. You could also modulate this with via tweaks like "if the line exists in an L1, then what the heck, let's just toss it, it'll bubble back to the L2 at some point as a victim".

The patent suggests that a cost is calculated for each way group that incorporates how much flushing is required, how many lines are in L1's, and how busy this way group has been, to decide the optimal way group of the three to power down, but does not suggest this next step of moving MRU lines that are unlucky enough to be in the power-down way group. Of course this is an obvious future improvement...
Finally, at some point you have to decide when to re-power-up the pieces of your cache that were powered down. Obviously the first signal is that you are missing frequently in the L2 (suggesting your

L2 is too small). But there are a bunch of subsidiary tests that essentially try to figure out if more capacity would really help (ie if you are missing because of compulsory misses, or because of streaming type behavior, increasing the cache size won't help much; what you want to detect is misses that are occurring that are somehow associated with reused lines).

There's an additional less obvious power saving available. Just like the L2 snoop-filtered the L1, so too the SLC snoop-filters the L2, using a set of duplicated tags stored with the SLC.
These duplicate tags are also powered by way, so that when a way group of the L2 is powered down, then the same way group of those SLC duplicate tags can be powered down. (2015) https://patents.google.com/patent/US9823730B2
Power management of cache duplicate tags.

compressed cache (sorta...)

Even better than a sleeping cache bank is a powered down cache bank! Consider this patent (2017) https://patents.google.com/patent/US10691610B2 *System control using sparse data*. The patent suggests multiple things but the starting point is

- detect “sparse” line writes
- don’t perform the write, instead note the line (eg by setting a bit in the cache line tag)
- service reads from the line by not performing a read and instead providing the “sparse” data.

So, obvious use cases
- tag all-zero lines in L2 and SLC/L3. (They talk about generic memory, but the diagrams, to my eye, look L2-'ish. Apple’s L2’s tend to have a 3-way replication, as we saw in the drowsy cache patent.)

- they open the way for a compressed cache (at L2 or SLC/3). It’s not a great fit once you get past “all zeros” and perhaps “all-1’s”; because compressed cache really wants something different (eg mechanism to read/write half a cache line). But, baby steps.

- in principle you can extend this up to RAM. In practice, I’m not sure – I can’t see a feasible way of recording the sparsity (bloom filter?) But maybe a flag in the TLB/page-table that records “all zeros”?

Equally interesting is the storage side. Essentially
- maintain one cache bank as the "loser" banks holding the invalid lines and the "all zero" lines
- as long as this holds, that bank can be kept powered off (even better than just sleeping!)
- once you have to break this (ie store real data in that bank) keep track of stats and, when it makes sense, repack the good data to a different bank, and re-power-down.

This is a more ambitious and more sophisticated version of the previous drowsy cache. Obviously powered down is better than sleeping; and providing machinery in the cache controller to move lines between "active", "schedule for sleep", and "powered down" banks makes the sleep and power downs

last that much longer.

It's probably misleading to call this a real compressed cache; but it does put in the place the first half of the sort of machinery one would like for real compressed cache support. As soon as you hear the term *compressed cache* you surely get the idea, but here's one example of the sorts of ideas that have been suggested:
(2012) http://www.cs.toronto.edu/~pekhimenko/courses/csc2231-f17/Papers/BDI.pdf *Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches.*

shared L2 and its consequences for shared frequency

This is small and no longer important, but it's interesting history. Look at (2017) https://patents.google.com/patent/US10147464B1 *Managing power state in one power domain based on power states in another power domain*. This sounds horribly technical, but in fact it describes the A10 (and only the A10). The idea is we have performance cores with an associated L2, and efficiency cores that use the L2 of the performance cores as their L2.
Doing this reveals a number of problems.
The more obvious, and easily fixed, is that you need to be able to keep the L2 powered up even when the performance cores are asleep, as long as the efficiency cores are working.
But the bigger issue is: at what frequency do you run the L2, and the performance cost of having to move traffic between the L1 frequency domain and the slightly different frequency of the L2 (meaning buffering across the domains), a problem with no great solutions.

All this is historically interesting, but raises a meta-issue. When cores share an L2 (and other hardware), either they all run at the same frequency, or the cost to access the L2 is substantially higher because of buffering across clock domains. So what's the right tradeoff?
Later below we'll see a very complicated (but clever!) Apple patent for optimal OS scheduling given the constraint of multiple cores having to run at the same frequency. But more generally, as we move from the simple world of one performance and one efficiency cluster to ever more cores, is four the optimal number of CPUs sharing an L2 (and thus sharing frequency)? There are clearly advantages to a large L2, and to a shared L2; but there are also costs.
Is a better tradeoff perhaps three cores sharing an L2, and the future low-end being something like two performance clusters (ie six performance cores), or the reverse of that going to 6 cores per cluster? ie either smaller, or larger, clusters but the same overall design.
Or how about two cores sharing a substantially smaller L2 and less hardware, and all the two-core clusters (including the efficiency two-core clusters) share a large CPU L3 and other hardware (distinct from the SLC), operating on its own frequency domain, and we just accept the cost of crossing to that frequency domain? ie we introduce an additional level of hierarchy into the system, going from core to cluster to CPU "block".

non - inclusive, non - exclusive

Also in the field of snooping we have (2018) https://patents.google.com/patent/US20200081838A1 *Parallel coherence and memory cache processing pipelines*. To me this seems most interesting insofar as it clarifies the following issue:
Intel have mostly used *inclusive* caches, eg the L3 contains all the L2's. The downside of this (cache space wasted on replicated data) is obvious, but the reason for it is that it makes your snoop filtering easy – the L3 can easily filter snoops for the L2, in that if the snoop misses in the L3, then there is no need to send it down to the L2 and test that maybe the line can be found there.
The next easiest alternative is an *exclusive* policy, as used by AMD. Depending on how this is done, you may or may not be able to snoop filter in the outer level cache, but what is easier is that the cache line state modification, because the cache line (and so its state flags) can only exist in one place.
Exclusion won't cost you space, like inclusion, but it may cost you some power to enforce that a line is always *moved* (not just copied) from one place to another.

What Apple do, however, is neither exclusive nor inclusive. To implement this, while retaining the advantages of snoop filtering by a higher level cache, the tags of lower level caches are duplicated in the higher level caches. As far as I can tell this happens in both the L2 and SLC, and the idea seems to be something like:
- core to core snooping will mostly occur within a core complex, and can be handled well by having the L2 for that complex snoop for all 4 processors of the complex
- snooping between different types of IP (like NPU snooping changes in a GPU cache) will be filtered at the SLC level
- left uncertain is whether snoops between the Performance Core Complex and the Efficiency Core Complex are treated as a special case or handled by the SLC.

Regardless, a consequence of this is that there are lots of duplicate cache tags! Obviously it's a fair bit of design work to ensure that they are kept in sync between the "original" cache (eg an L1) and a duplicating "higher cache" like the L2. One can see the appeal of the simpler inclusive or exclusive models!
Many of the Apple cache patents have to do with various aspects of these duplicate tags, for example the one referenced at the beginning of this paragraph has to do with sequencing issues. When a snoop comes in, the first thing a cache will do is compare the snoop to its tags, to see if the snoop matches a cached line. But does it first look at the cache-native tags, or at the duplicated tags?
The patent says "look at both simultaneously", with a bunch of complications that then result depending on if both tags hit (eg the line state has to be changed in both the L2 and (perhaps multiple) L1's), if the line is locked (eg through a load exclusive instruction), and whether the instruction has passed the "global ordering point".
And interesting side aspect to the patent is that its Fig 3 shows what is probably something like the current design of the SLC.

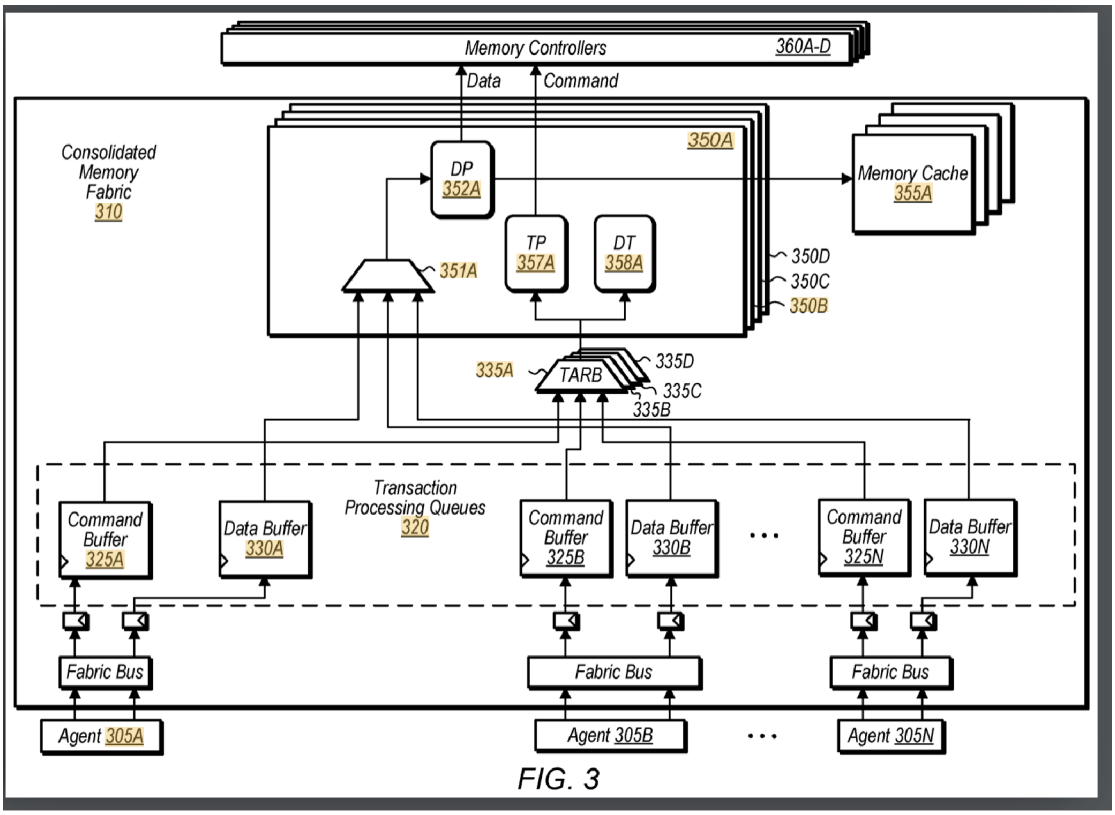


FIG. 3

The area within the dotted line is essentially a single pool of common queues which all requests flow through. Four arbitrators (in principle there could be more, but four seems reasonable for an M1 class SoC) route these requests to one of four "slices". No-one ever says exactly what they are doing here, but an obvious way to think about this is to consider (physical) addresses as having bits 0..5 specify a byte within a 64B line. Then in the simplest case, use bits 6 and 7 to choose one of the 4 arbitrators and slices. But of course one can use a more complicated hash that uses more address bits. And in the case of Pro, Max, and Ultra, this more complicated hash would kick in earlier, routing different lines to

different SLC's.
This is somewhat analogous to Intel using a hash to distribute addresses across different L3 slices in their recent cores.

We can be a little more precise about this business of cache inclusion/exclusion. If we trust (2007) https://patents.google.com/patent/US7702858B2
Latency reduction for cache coherent bus-based cache, which is admittedly from a long yime ago, what that says, among other things is that the L2 is

- inclusive for instructions (I really don't see the point of this. Presumably the idea is something like: code that's loaded by one core is often code that's used by other cores, either via a multi-threaded app or in shared libraries. Why not sideload such code from another core's I cache if it is there? Well that would require modifying the I-cache to support reading values out of it, and while that has to be done for the D-cache, for various reasons, it's not something the I-cache naturally has to do, so adding it would cost power and area.)
A constant point in the academic literature is that, ideally, you want to handle instructions in an L2 differently from data. It's much harder to hide the delay from an I-cache miss than a D-cache miss, so you want to prioritize I lines over D lines in L2 one way or another. For example, you want to make it much harder to kick out an I line than a D-line if there's ever a choice as to which to remove when a new line is installed. (You could do this by eg probabilistically flipping a coin, with the data line given a higher weighting in its probability of being the evicted line. This enforced inclusivity seems part of that tradition.

- victim for data. In other words a data line is loaded directly from its source (let's say DRAM) into L1, without being stored in L2. At some later point, when the line is evicted from the L1D, because a new line is loaded and needs to take its place, the old line is then moved to the L2. Subsequently it could be reloaded by the L1, at which point it would be supplied by L2 and would then be in both caches.

(The actual content of the patent is interesting, though surely now changed and essentially irrelevant. Suppose a requester makes a request. Recall that this is before the (2010) first snoop filtering patents. So the request goes to both the L2 and the two L1's (and any other caches on the SoC). Theoretically the system should wait for everyone to respond to the snoop before the next step of having one of the agents, if possible, supply the data. The patent outlines circumstances under which the L2 can return the line from its contents before having to see the responses by the L1's underneath it.)

moesi

BTW it seems likely that, from at least the A6 and A7 days, Apple was using MOESI as their cache protocol. You can find the details on Wikipedia, but one line summary is that MOESI augments MESI with an Owned state which allows the cache-to-cache transfer of m*odified* lines. In other words CPU A asks for a line, and if that line is held modified in CPU B's cache, then the protocol allows for the transfer of the line.

Compare this with MESI (which allows no cache to cache transfer), and a protocol which allows *unmodified* lines to be copied from one cache to another (this was introduced by IBM as MERSI, and then used by Intel as MESIF).
Obviously best of all would be a protocol that allowed (as much as practical) all lines, modified or not, to transfer cache to cache rather than having to be pulled in from a higher cache or DRAM, and such a MOESIF protocol is possible.

You can keep adding more and more cache states if you're willing to pay the complexity cost; primarily to try to include things like "this line is shared between multiple cores, but they're all on the same socket" vs "this line is shared between multiple cores on different sockets": think about all the different levels of cost involved communicating between cores that share an L2, vs those that share an L3 (but are on the same socket) vs those that are on different sockets. The more your protocol clarifies these different degrees of sharing, the less you might have to wait in response to any particular broadcast of "I am about to do something with this line; does anyone out there care?"
If you can't get enough of this stuff, IBM is the master, with extremely complicated protocols given that their big systems that go up to L4 caches, and consolidate multiple packages across multiple boards!

Even Apple has got into the game: (2009) https://patents.google.com/patent/US20100235586A1 *Multi-core processor snoop filtering*, though I've no idea what this was used for – presumably not for anything iPhone related, so some sort of custom cache controller for the dual-package first generation Mac Pro's?
The idea is, however, interesting and sensible. Most pages are never shared, so suppose we indicate in the page tables, propagated to the TLB and then to the cache, that pages are not shared. This would mean that interactions with cache lines of those pages could avoid many snoop broadcasts, saving energy and bandwidth!
This seems like a good idea that's the sort of thing very appropriate to Apple – co-ordinated changes to the OS, APIs and HW, that would be impossible for most vendors. So maybe it's present in the M1?

This is followed by (2011) https://patents.google.com/patent/US8856456B2 *Systems, methods, and devices for cache block coherence*, still Mac rather than iPhone related, but specifically discussing the cost of coherency between a CPU and a high-bandwidth device like a GPU.
The extension beyond the previous page-based system is that many lines can, in principle, be shared, but in practice they are not; eg they are constructed in the CPU, loaded by the GPU, removed from the CPU's cache, and the CPU never cares about them again.
So, from the pattern of snoops, and other shared info, each device builds up knowledge of not only what lines it holds, but also something about what lines other devices hold so that, for example, if it knows that no other device can be holding a particular line, snoop broadcasts related to that line can be suppressed.
There are some nice details in this the scheme. For example, initially (non)sharing is tracked at a high granularity, but stats are maintained and, if these warrant it, the granularity associated with an address range is reduced, to allow for more fine-grained tracking of (non)sharing.

manually managed cache

When we use the word cache, we tend to think of a *transparent* cache, in other words one that automatically decides which lines to retain and which to remove. But manually managed caches are another option, and are useful in a variety of situations.
One case is structured data access, where the pattern whereby you will access the data is well-defined (common in DSP code, or image processing code).
Another case is code that may be called infrequently but you want it to be low-latency when called (Apple give the example of interrupt routine code).

To handle these Apple have provided a variety of manual controls over the years. The traditional way of doing this is to allow some lines to be locked in an L1 or L2, easy to implement.
But Apple, as usual, has a much more interesting setup:

2019 address ranges mapped to particular cache lines

The most recent scheme is (2019) https://patents.google.com/patent/US10922232B1 *Using cache memory as RAM with external access support*. The idea here is to allow a cache (they seem to imagine this as a possibility from L1 up to L3, and give examples using L1) to have certain address ranges mapped (via in-cache registers) to some lines of the cache.

An obvious question is "what is the difference between locking a line in a cache" and "mapping an address range to a line in a cache", except that the second seems more complicated?
I think the win is for *ephemeral IO* data.

Consider a simple network stack.
Data comes in from some hardware, hopefully via DMA, into DRAM.
That DRAM buffer (owned by some low-level software like a driver) gets copied to the network stack which may copy it a few time each time stripping off headers and performing some level-appropriate (IP, then TCP, then HTTP) processing, finally copying it across the OS boundary into a user buffer.
That's a lot of copying, and so there has been a continual push to reduce the number of copies to zero. Essentially every level of the SW stack agrees on a protocol for how to allocate a buffer, how to transfer ownership between levels, and how to strip off headers or tails from these buffers by manipulating pointers associated with the buffer.
This all sounds great, but once this was all implemented, the results were disappointing; an improvement yes, but not nearly as much as hoped. The problem is that the first step, moving the data off the hardware (disk or network) is done via DMA – which dumps the data in DRAM. Every subsequent copying step is minor in cost compared to the initial cost of moving the data from DRAM to cache.
So if you eliminate all the copying, the OS-specific part of the operation looks fast; but your app is not much faster because now, as soon as it starts processing the data buffer it receives from the OS, every line of that buffer causes a miss in L1. Mainly what you have done is to move the primary cost associ-

ated with the operation from occurring within the OS to occurring within the client :-(

But suppose that I can designate the buffer address range as part of a cache (either L1, L2, or SLC) and have all the pieces along the way (the memory controller and all the cache controllers) understand what has been done. Now when the network or disk device performs its DMA, the data is dumped *directly* into a cache, and we no longer have to pay the cost of the initial copy from DRAM to cache!
(One can imagine even wilder use cases for this. For example if I'm only using one P-cluster, the OS could, conceptually anyway, map an address range to the L2 of the other P cluster and allow me to get some benefit from it.
I expect these sorts of conceptual optimizations are at an extremely early stage within Apple, but they will surely come once all the most obvious issues associated with the Apple Silicon transition have been handled and people can start thinking about more sophisticated ways to use the hardware.)
The technical focus of the 2019 patent is the fact that both memory and IO can simultaneously enqueue requests into a cache which is also providing a "memory range", and that the ordering in which these requests are serviced has to be handled carefully otherwise deadlocks can ensue.

2009 address ranges in SLC only, and with restrictions

With the above in mind, we can look at the much earlier (2009) https://patents.google.com/patent/US20110010504A1 *Combined Transparent/Non-Transparent Cache*. Superficially this looks like the same idea, marking part of a cache as an address range. But there are many more restrictions (and some interesting background ideas mentioned).
As restrictions, the idea seems limited to the SLC, and is apparently not available to IO (except as very restricted DMA from DRAM into/from the SLC). The idea seems to be provided for structured data patterns. So you can request a block of "fast memory" of a certain size and have that allocated in SLC rather than RAM for your temporary use case.

As interesting ideas, the patent makes three points.
- First is that this storage isn't used like traditional cache storage, and so doesn't require tags; rather a few range registers in the SLC can route addresses as appropriate to this storage. So while this is nominally part of SLC, it kinda lives on the side, invisible to code that doesn't explicitly take advantage of it, and more dense because of no tags and many fewer (no?) associated line state flags.
- Second is that while the requester can ask for memory that matches an existing address range (with flags that will optionally copy the data in DRAM into the SLC at the start and/or copy it back to DRAM at the end), an alternative is to ask for an "invisible" address, namely an address that's in the middle of the address map, above the address range that's mapped to DRAM, and below the address range that's mapped to various bit of IO and OS business.
- Third is that at first you probably thought of this as something to be used by CPUs, but these restrictions should make it clear that Apple appears to have in mind as primary clients things like the GPU and the ISP (as I said, structured data access, for internal Apple drivers and suchlike).

Ultimately the use case for this earlier patent is complementary to the use case for the 2019 patent, and I wouldn't be surprised if both are present on the M1.

I have seen hints that nVidia can do something similar to the above (allocate address ranges that route to their L3) but I know very little about nV and GPUs in general, so I may have misunderstood.

running the display out of SLC

Now put together some of these various ideas for unusual ways to use the SLC. How further can we save energy?How about (2013) https://patents.google.com/patent/US9261939B2 *Memory power savings in idle display case*.
The standard model for a computer (mobile or desktop) has the GPU generating data that is written out to DRAM, and the display reading that data from DRAM, both doing this say 60 times/second. So both directions we are paying the energy cost of reading/writing off-chip to DRAM. What if we could avoid that cost?
The 2013 patent does this in a limited way. When the system detects that the contents of the display buffer are static, the display buffer is copied to the SLC, and the display controller runs out of SLC rather than DRAM.

And there's even more that can be done! (2013) https://patents.google.com/patent/US9396122B2 *Cache allocation scheme optimized for browsing applications* points out that once you are using the SLC as framebuffer memory, you know how the data will be accessed. And you can time the sleeping of the SLC SRAM arrays so that you only wake up the specific array that needs to be read by the display controller at any given time, allowing all the other banks to sleep.

How do you discover the screen is static? You test the obvious things like changes to compositing buffer addresses, and the compositing queue instructions, but the main one is you calculate a CRC from the bytes read as you paint the display, and ensure that this CRC remains unchanged for N frames. This is described in (2013) https://patents.google.com/patent/US9058676B2 *Mechanism to detect idle screen on*
(The rest of the patent is uninteresting in that it's an early version of the static frame idea, before using the SLC; this early idea was to store a lightly compressed version of the static frame in DRAM, so that loading it and decompressing each time cost less than loading the data from multiple compositing buffers and merging them together.)

This is a nice way to get auxiliary use of the SLC under conditions where it's not otherwise being used, but one can surely do much better. I would not be surprised if modern Apple SoC's have permanent dedicated on-SoC storage for the primary display of the device, from at least Apple Watch up to MacBook Air, something like a block of memory-addressed (untagged) SRAM within the SLC where some part of it is configured to act as display storage (sized as appropriate for the device), and the rest is available to the OS for the sorts of use cases suggested above.

cache replacement using LRU or FIFO on a line-by-line basis

We can see more cache design choices in (2009) https://patents.google.com/patent/US8392658B2. *Cache implementing multiple replacement policies.*

The idea here is in a cache (conceivably everything from L1 to SLC) the cache tags include, along with all the other status bits like MOESI, provide a bit specifying the replacement policy for this particular cache set. The explanation is clumsy, but the idea is that while LRU usually works well, there are access patterns (specifically streaming through data) for which a limited FIFO replacement works better. (Limited FIFO meaning you restrict the streaming data to just a few of the available ways in any set, in the most extreme case just one way, but more generally perhaps two or four ways in the hope of some short-term reuse.)
There are papers available suggesting how a cache might dynamically figure out which of these techniques (LRU vs limited FIFO) is currently better, but the patent is not that ambitious. Instead, the cache is informed, in each request, at to whether the request should be treated as LRU or as part of a stream.

This information in turn is, either associated with pages (ie set up as part of the pages tables) or is held in range registers. These range registers appear to something of the equivalent of PPC BAT registers, situated in MMUs, and being consulted in parallel with the contents of the TLB. Presumably they are used for the obvious IO cases (like Display DRAM) but the patent also suggests that (some of them, somehow) are available to SW. We saw these same range registers in the earlier referenced 2009 TLB prefetching patent, where they were used to describe different memory classes being accessed by the GPU (texture, pixmap, etc) along with whether they would benefit from stream-like prefetching.

Different cache replacement policies sound cool, but I suspect that once caches become large enough, having them be essentially hash-direct-mapped (or with low associativity, and the different way choices determined by power, as in the L2 scheme) is more important (power savings) than the minor increase in misses resulting from conflicts.

cache telemetry

Consider the M1 and an L1 cache miss for a particular core. That miss may be serviced from a variety of other places, for example
- another L1, or the L2 of this cluster
- a cache (L1 or L2) of a second cluster
- the SLC or DRAM

Suppose that we track cache fills by these different sources; we can then use that information in a few different ways. (Precisely which options are optimal in particular conditions will depend on other details). For example
- if we see an above average number of fills from DRAM we might increase the frequency of DRAM and the SoC communications fabric.

- however if those are already at maximum, and we are still frequently waiting on DRAM, we should reduce the speed of the core or cluster since it is mostly burning energy waiting for memory. (As we've mentioned, the frequency of an entire cluster has to change together, so it's hard to directly slow down

a single core. However there's always the option of holding the clock so that that single core sees only every second or every fourth clock transition, which may be a good enough substitute; we don't get the voltage reduction, but we do save the power that would otherwise be expended on clock transitions.)

- if we see an above average number of fills from another cluster, we should increase the frequency of that cluster, so that we are not always waiting on them

- alternatively we should have the OS reconsider how threads are being allocated to clusters.
The baseline heuristics the OS uses for scheduling are static, things like allocate threads/processes by QoS.
But statistics gathered during execution can augment this by telling us that certain threads (in the same process or different processes) are constantly communicating through memory, in which case those threads should be scheduled together, on the same cluster.

These ideas, describing per-core counters that track these cache fill sources, are the content of (2019) https://patents.google.com/patent/US10942850B2 *Performance telemetry aided processing scheme*.

However there's a second, stranger, aspect to the patent! By now we've seen many Apple patents, and they tend to show a few common baseline designs, primarily an approximation of the the 1st gen up to A6 design, likewise for the second gen A7..A10 design, then the third gen A11..M1 design.
Not this patent! It describes a design with
- two clusters
- four cores in each cluster
- a per-core L2 and
- a shared per-cluster L3 (which the patent calls LLC)

Now this could just be the fancy of a different law firm writing up the patent. But it makes one wonder if we are seeing an aspect of the 4th gen design.

consolidated address translation unit

Given our experience with x86 (and similar designs that grew from a single simple CPU) we tend to treat the terms TLB and MMU as more or less synonymous.
However, as far as I can tell, Apple have likewise deconstructed this machinery. I can't be sure of the details, but approximately the TLB is something like a simple L1 cache, while the rest of the MMU (page walkers and suchlike) appears to be a unitary entity living up at the SLC/memory controller level (and shared across all agents).

There are many ways to slice the problem of page walking, but an issue one constantly has to remember is that Apple's concern is very much an entire SoC, not just CPUs, and this has constant ongoing implications. For example one likely has translation required for all IO elements (not least for security), which means there's something somewhere that's providing a page walker so as to populate a TLB for that IO element.

Look at this diagram from (2011) https://patents.google.com/patent/US9652560B1 *Non-blocking memory management unit*. This suggests (which is the obvious solution) that this functionality is associated with the SLC.

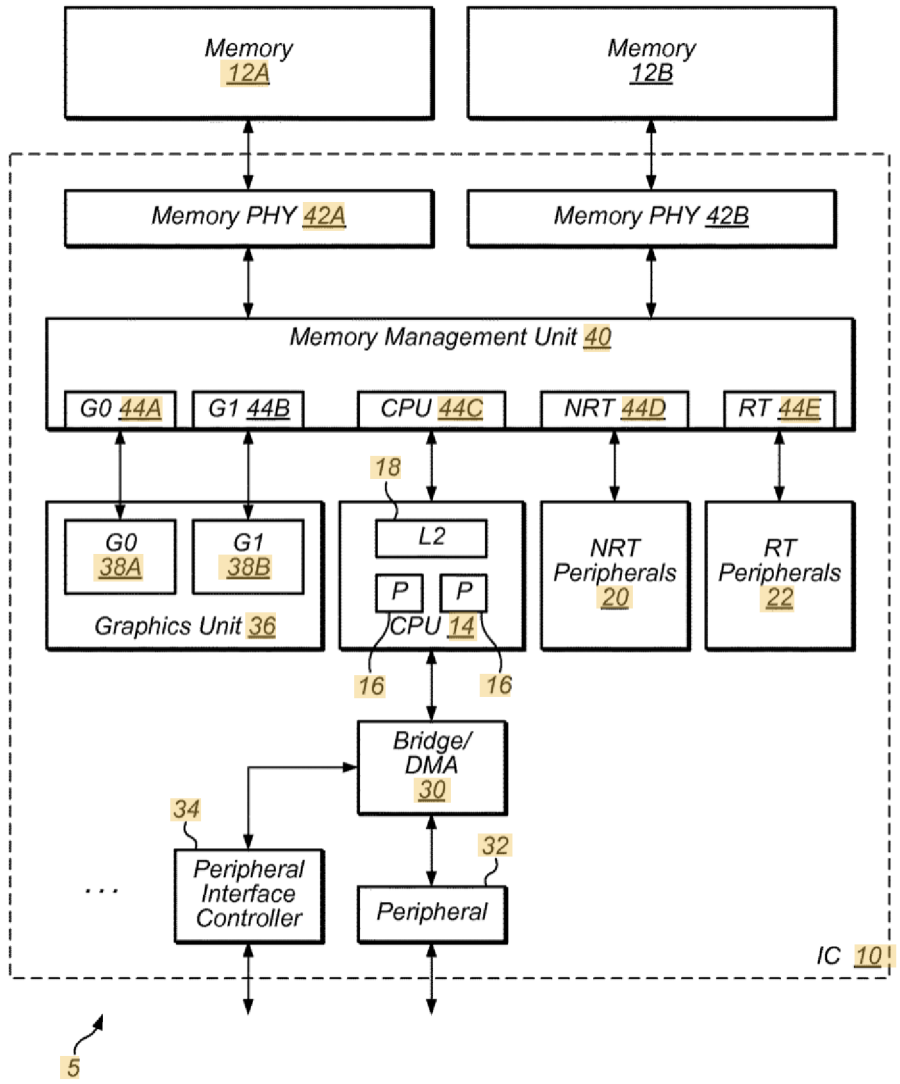


FIG. 1

(BTW, though more GPU than CPU, the 2011 patent is interesting in itself. As far as I can tell, at that time [remember, PowerVR GPUs...] the GPU internally operated purely in virtual address space [with the implication that GPU caches were flushed on context switch to a different GPU user address space].
So it's only when requests left the GPU that they needed to be translated. These requests flowed through the MMU as in the diagram above, on their way to servicing the request from the SLC or DRAM.
One consequence of this is that this "GPU-external" MMU can service page faults even though, nominally, the GPU has no support for virtual memory. What's required is that the MMU detect that the translation request matches a non-present page, and route a request to the CPU to fault in the page from storage. Once that is done, the GPU memory request [now translated, and the data serviced from the faulted in page] can be sent back to the GPU.

This is different from a CPU, where a page fault is accompanied by a context switch, to give control to some other process while storage services the fault. But it works because a GPU, of course, has thousands of active threads, all ready to step in as soon as a cache miss occurs; and to the GPU this page fault mostly looks like a cache miss that took longer than usual.

Why page faults anyway? I thought iOS had no VM?!
Not correct! iOS doesn't [usually] have page-outs and so what looks to the app like unlimited memory. But it does support other aspects of virtual memory including memory mapped files, and it is not uncommon to use memory mapping for things like large texture atlases.)

APRR/SPRR (changing permissions of pages)

With all this speculation in mind as to the economy and efficiency of having a single locus of truth for all TLB tables and all page walkers associated with the SLC, (2019) https://patents.google.com/patent/US20210064539A1 *Unified address translation* is very interesting.

The patent itself is essentially about APRR/SPRR, a sort-of security/sort -of performance feature described in partial detail here: https://blog.svenpeter.dev/posts/m1_sprr_gxf/

The basic idea is that for tighter security one wants to be able to flip the permissions of some pages fairly frequently. For example for JIT pages, one wants to be able to flip the page between Write mode (no Execute) and Execute mode (no Write) fairly frequently. But traditionally modifying page permissions is an expensive process. The page table entry in RAM has to be changed, then a broadcast sent to every TLB on the system (and remember that include the GPUs, NPUs, and various other IO devices) telling each to flush the relevant page from its TLB, followed by a wait till every TLB responds that the flush is done.
The Apple alternative is that a page no longer has a set of permission bits, rather it has a "permission index", which moves with the page translation into the TLB. This index is split into two parts. Under "A" conditions, the first part of the index is used to index a (short, 4 entry) table giving the appropriate permissions; while under "B conditions" the second part of the index is used.
The usage model, I assume, is something like
- normally the A conditions are valid and represent safe page usage
- if a temporary change is required (eg to write to a JIT page) the B conditions are established (by writing to a register or whatever)
- as soon as the change is done, conditions flip back to A
I *think* the idea is that the B conditions are only set (briefly) for the specific CPU that is making the (temporary) change to the page, so no other TLB needs to be informed of this change, avoiding all the cost of (two!) standard TLB teardowns for what will be a temporary modification.

Honestly, security is of no interest to me. But if its your thing, that 2019 patent builds on (2016) https://patents.google.com/patent/US9852084B1 *Access permissions modification*, which describes an early set of augmentations to the MMU to provide various security improvements. Among other things, these augmentations include
- a BAT-style register describing the range of genuine OS code, so that code outside that range cannot run with OS-level permissions
- a lock register for locking the above BAT, the page security remap tables and various other things after they have been constructed so that no attacker can modify them after a very short boot window
- ways to limit the abilities of OS and hypervisor code when they are accessing user pages.

split between TLB and ATU

More interesting however, IMHO, is the material that is hinted at in the 2019 patent. The existing state of the art describes CPUs as having a TLB and an ATU; the new design talks of the ATU being optional in the CPU and moved up to the processor complex.
What's the difference? The point is that this is what I described above: a CPU has a minimal L1 TLB, but no ATU (Address Translation Unit, ie all the extra machinery to perform table walks). If the L1 TLB misses, the request goes up to the Processor Complex (ie L2 cache) which will have a large L2 TLB, shared across the cores that share the L2 cache, the page walkers, and so on.
So it seems like versions of this idea in one form or another have persisted from 2011 till now.

A different aspect of the TLB/page walker is optimal performance. (2011) https://patents.google.com/patent/US9009445B2 *Memory management unit speculative hardware table walk scheme* is surely obsolete in many details, but the essential idea remains interesting. The patent describes a limited machine (think PA Semi, even before Swift) with the following interesting features

- there are the usual separate I and D TLB's, D-TLB tightly associated with the LSU; but there is a consolidated MMU that provides a second level TLB and a consolidated page walker. In other words even at this early stage we separate the *high performance cache* aspects of a TLB from the other "*overhead*" aspects of operating page walkers.

- the page walker provides a queue for TLB requests.

- the precise details don't make much sense to me (and may be an attempt to lower the energy/transistor budget) but the end result is that the page walker prioritizes all "definite" page walking (which could be on behalf of either I or D cache) before any "speculative" page walking. The target CPU appears to have been so restricted that speculative means literally what it says, load/stores were segregated by speculative (not yet latest in ROB) vs non-speculative!
However one could imagine a future system with multiple page walkers; and an obvious issue then is prioritization in the face of many simultaneous requests. A scheme with I misses as highest priority, then D misses, then I prefetch misses, then D prefetch misses as lowest priority seems most sensible.
My primary takeaway from the patent is that the more you have page walking centralized to a single locus of control, the more you can engage in this sort of prioritization of different classes of page lookup, even across the requests from different CPUs within the cluster.

A cluster-common (and SLC-common) ATU could also prove a win when changes to page tables are made and these changes have to propagated to every TLB, acting as a snoop filter for the L1 TLB's, just like L2 does for the L1D and L1I caches?
Going further, there have been academic papers bemoaning the fact that, even as the industry adopted hardware cache coherence many many years ago, TLB consistency, which looks like much the same problem, remains rooted in SW:
(2010) (UNITD) https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.438.2661&rep=rep1&type=pdf *UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All*
(2011) (DiDi) https://www.doc.ic.ac.uk/~lvilanov/publications/files/pact11_didi.pdf DiDi: *Mitigating The Performance Impact of TLB Shootdowns Using A Shared TLB Directory*

uncacheable address ranges (ie IO buffer storage)

Let's now consider uncacheable loads and stores.
This is a messy subject, in part because different spaces (embedded vs PC) and different companies use the same words in different ways. Let's try to understand the main ideas without obsessing over minor details.

Consider the following very simple system: We have DRAM with an associated memory controller, we have a CPU with an L1 and L2 cache, and we have one peripheral, which we will call a "network controller". This is our baseline simple system, so note that the network controller is minimal, in particular it knows nothing about caches.

In this system, suppose we want to send a network packet. The very general idea is that the CPU will fill out a buffer at address A of size S, then tell the peripheral to do something with that buffer.
But there are many important details within that general idea.

address translation

The first important detail is that this simple peripheral knows nothing about virtual memory and how to translate addresses. All it knows is the address it was given, so that's the address it will attach to any transaction. And so, in this simplest model, that address needs to be a physical address. That means that when the CPU allocates the buffer at address A, the code has to be aware of both the virtual address of the buffer (so the CPU can write to it) and the physical address (so that the peripheral can be informed of the physical address).

For most peripherals this is a minor hassle, nothing more. But it does have two aspects that are more than just hassles.
- There is a security issue here. Because peripherals access raw physical memory, they can access *any* raw physical memory. This was perhaps not an issue when peripherals were really simple; but it became an issue when peripherals added some smarts/programmability and so could, perhaps, be hijacked by hackers and could then read or write anywhere in memory without the constraints and protections of the page table.
- If you have a sophisticated peripheral (GPUs were first, but NPUs are similar) and the interaction between the CPU and peripheral is no longer "here's a buffer, go write it somewhere" but something more like "here's a complex data structure like a graph; go execute some graph algorithm on it", then you may want the block of data that you pass to the peripheral to include embedded pointers to other parts of the buffer; and at that point manually translating from VA to PA every pointer used to construct a graph or similar data structure becomes very unappealing!
- As a less important issue, but still a hassle that has to be tracked, that virtual address has to be locked to the physical address; the mapping cannot change until the transaction is over.

Of course now, in the modern world every sophisticated peripheral comes with some sort of MMU to perform VA to PA translation at the peripheral.

In the case of the GPU, this MMU may be essentially identical to the CPU's MMU, with the same structure of having translations tied to an address space tied to a process, and swapping the processID on context switch.

But for more basic peripherals like a network controller, the main concern is protection, so processIDs may not be relevant, mainly just limiting the physical address to a safe space desig-

nated by the OS, perhaps along with some sort of user/supervisor toggle.
Apple provide a very early partial solution with a System TLB (2009) https://patents.google.com/-patent/US8316212B2 *Translation lookaside buffer (TLB) with reserved areas for specific sources*, and perhaps they still use something similar for the simplest peripherals like microphones and speakers?

OK, so that's addressing. To simplify the discussion, we will ignore it from now on and just refer to "addresses".

control (basic discussion based on PIO)

Next is the issue of how we tell the peripheral what we want it to do, ie how do we control the peripheral? We will get to this, but for now let’s just imagine that we have special instructions called IN and OUT that somehow manage to communicate with a peripheral.

coherence of buffer data

So finally we have the question of interest, how the buffer *data* is handled. Here’s the issue:
+ The CPU fills up the buffer (writing S bytes at address A), and these S bytes are now in the L1 cache.
+ The code now executes an IO instruction telling the peripheral “write S bytes from address A to the network”.
+ The peripheral sends a request to DRAM for those bytes.
+ And DRAM sends back the *unmodified* bytes that are sitting in DRAM at address A, not the bytes in the L1 cache!

CPU’s do not have this problem because CPU’s are sophisticated devices that, among other things, include hardware that snoops all memory transactions. But we have no control over our network peripheral, and it was not built with such snooping hardware.

How do we fix this?
In increasing order of sophistication (and performance)

- We can mark a region of physical address space (say the first 1/16 of physical address space) as *uncacheable*. This means that the CPU cache controller knows that physical addresses in this range are treated differently, and go straight to DRAM; and that the OS knows to allocate IO buffers in this address range.
Now as the CPU fills in the S bytes of the buffer, each CPU store goes straight to DRAM, and they are all there when the peripheral requests them.

- Alternatively we can make the device driver a little smarter. We establish a rule at the OS level

that after any buffer is filled in, the device driver has to flush that buffer to DRAM before executing an IO instruction that references that buffer. This requires the CPU to have a "flush cache line" instruction, but that's no problem.

The second alternative is usually preferable, even though it requires more careful coding, because it's faster.
The first alternative delays every write of the S bytes, whether the CPU is writing a two byte CRC in the header, or 8 bytes of data in the packet payload.
The second alternative transfers an entire cache line for every memory round trip, so is clearly more desirable.

But it's not always so simple! In the bad old days before GPUs were as sophisticated as today (or before they even existed) writing to screens was especially problematic.
You might write to pixels in a fairly random'ish pattern (and then it's not easy to track which cache lines you should manually flush to screen VRAM if you allow the screen's physical address range to be cached).
The alternative is to not allow such caching, and try to be smart about coalescing uncached writes. So rather than simply executing an uncacheable write all the way to DRAM, store it in some intermediate buffer associated with the L1 cache, and try to aggregate some number of these stores (ideally up to a cache line) before transferring them to VRAM.
The problem is that different users want different types of attributes for memory requests, and unless you carefully disaggregate those attributes, you have to service the lowest common denominator. So, for example, write coalescing for screen VRAM makes sense – but other peripheral use cases may insist that they want *immediate* writes (no hanging around in a buffer for a while, waiting for more writes to coalesce into a larger unit) or *ordered* writes (ie each write results in a distinct memory transaction, that must happens in the order of the CPU performing the writes).

Back to the main theme. We have the issue that the CPU, for performance, and to write natural code, wants to fill in a buffer (eg a network packet) via standard store operations, but this will create the packet in the cache where the peripheral will not see it.
We've also given one possible solution to that, namely flushing the relevant cache lines.

- But there is a third option available that's smartest and best performing of all!
If you can't make the peripheral smarter, what about making the memory controller smarter? And that's what Apple does.
Suppose that we place, before the memory controller a System Level Cache (SLC) containing, among other things, a Coherence Point, and tags for all the other caches in the system (for example for the P L2 and the E L2).
Now what happens when we go through our previous example?

+ The CPU fills in the S bytes of a packet, writing data in the L1 cache of a CPU.
+ The L2 of the cluster containing that CPU knows that address A is present within that particular

L1.
+ Because the SLC mirrors the tags of the L2, the SLC also knows that address A is present within that particular L1.
+ The peripheral requests data from address A from the memory controller.
+ But all interactions with the memory controller first pass through the Coherence Point (which essentially ensures that they are appropriately ordered relative to each other) and then through the SLC.
+ In this case the SLC will see that address A is present within an L2 tag, and so will request the data from the L2, which will request it from the L1.
+ Eventually the data routes through the SLC on to the peripheral.

We got all the performance benefits of a cache, in fact substantially better than the second model above because the data never even needed to go off chip to DRAM; and we did not have to write any special additional code to flush cache lines.

So that covers the issue of *data* that is to be transferred between various devices on the SoC, why (in the past) we used to care about whether that data is constructed within a cache, and how Apple solves the problem nicely for many of the common cases.

control (discussion based on Device Memory Address Space)

Let's return to the issue of control. While x86 cores had `IN` and `OUT` instructions, many CPUs did not and still do not. Instead they use memory mapped peripherals, the idea being that a write to particular addresses in the physical address space acts to tell the peripheral to do something (and reads from other particular addresses may provide the status of the peripheral). The thing to note about memory mapped peripherals is that the memory mapping refers to control plane not to data plane, ie the expectation is that the OS will perform a few reads or writes to these control addresses, but bulk transfer of a network packet or a disk sector or a camera image will occur via the DMA mechanism described above.
ARMv8 (and presumably Apple) handle this via giving a (small) region of memory the attribute of *Device* memory. Device memory is an extreme form of uncacheable memory because reads and writes to device memory are not really reads or writes, they are "commands", and you want them to be treated as commands. If you want to see a little more about this, ARM has a nice explanation here: https://developer.arm.com/documentation/100941/0100/Memory-types *ARMv8-A Memory systems*.

The main takeaways so far should be
- Most of the use cases of uncacheable memory (for *data plane* purposes) are handled automatically and efficiently, behind the scenes, via the SLC

- But Device Memory is the one remaining form of uncacheable memory that really is uncacheable, and that needs to be handled very carefully (but is rarely performance critical)

optimizations built upon this SLC mechanism for IO buffers

Apple have a patent in this space, (2012) https://patents.google.com/patent/US9043554B2 *Cache policies for uncacheable memory requests suggesting* slight performance tweaks one can make, based on knowing that an address range acts as an IO buffer rather than as more general memory, however I won't discuss these given that they are probably obsolete and covered by newer mechanisms (handling of memory streams, write coalescing, streaming directly into a cache).

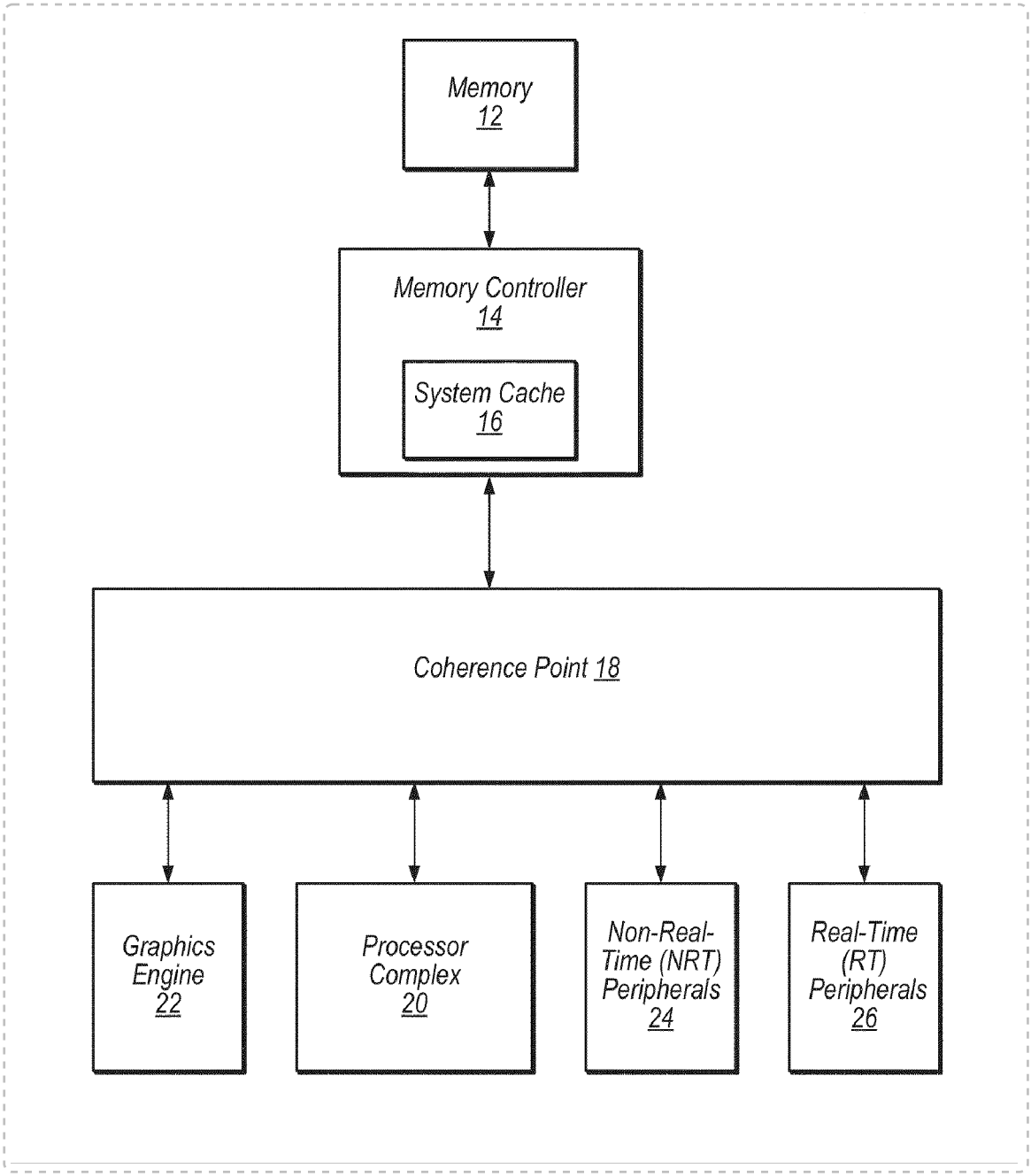
However, even as recently as 2019 uncacheable memory operations were still on Apple's mind, though I have to admit I have no idea what parts of the SoC are still operating as non-coherent memory, and the patent gives no suggestions. I think the concern at this point is to be able to control peripherals as efficiently as possible, ie to be able to send out loads (read status from a peripheral) or stores (set the status of a peripheral) as efficiently as possible, while maintaining the necessary ordering and timeliness.

The actual ideas in the patent, 2019 https://patents.google.com/patent/US20210056024A1 *Managing serial miss requests for load operations in a non-coherent memory system*, are fairly simple, essentially a load equivalent of store merging:
• there is a pool of buffers in each cache controller for the use of uncachable loads
• when a load is placed in one of these buffers a timer starts
• if a subsequent load comes in that can be aggregated with this load (ie the two together hit in the same cache line and so for a single wider load) that is done
• until either the timer expires, or a maximum width load is constructed

I assume the idea is that many use cases for these uncacheable loads consist of a sequence of back-to-back sequential loads of some limited width (maybe 32b or 64b depending on the block of the SoC) and this is a fairly easy way to consolidate them to full cache line width transactions. Then that single transaction goes out to a peripheral which likewise packs four or eight pieces of state into a single cache line that is returned to the CPU.
You'll also note the long gap between the early round of these patents, from the PA Semi days, and this patent. Perhaps, in a sense, the 2019 patent is a consolidation of the ideas from these early days, now optimized to take full advantage of the precise memory-ordering rules of the IP Apple currently cares about (latest versions of ARM, AMBA, PCIe, etc) without having to worry about some of the issues that seem to have limited all the earlier patents to only store aggregation, never load aggregation?

Memory controller

The Memory Controller and SLC are, in Apple's design, two halves of a single task, so they need to be discussed and considered together.
The SLC should be considered an extension of the memory controller and of DRAM, rather than as an L3 cache; Apple sometimes even refer to it as a Memory Cache or a Memory Controller Cache.
This difference is not mere verbal quibbling; it has implications for exactly how coherence is handled, and it allows the memory controller to be tightly integrated with the SLC (as in using it for a Virtual Write Queue, as I've already suggested).
From (2012) https://patents.google.com/patent/US9218040B2 *System cache with coarse grain power management* :



request scheduling (based on qos and affinity)

To someone familiar with PCs, one of the less familiar aspects of the Apple design is the handling of QoS, which permeates everything, from peripherals through the NoC to the SLC and Memory Controller.

This (2010) https://patents.google.com/patent/US8510521B2 *Reordering in the memory controller* looks very technical, but there's some interesting stuff described as asides, apart from the main patent. The system (ie early iPhones) has five independent ports into the memory controller, two for the two

"GPUs" (I would expect by now these are consolidated into a single port behind an L2 cache), one for the CPU system (two CPUs behind an L2 cache, with some peripherals connected to this subsystem), an NRT port (non-real-time, primarily for media encode and decode) and an RT port (real-time, primarily for display controller stuff).
The requests through each of these ports are tagged with various levels of QoS (at the time, three levels of QoS for RT, two levels for NRT, and everyone else gets whatever is left) along with a flowID. The requests are routed to one of two memory channel controllers.

The focus of the patent is optimizing for bandwidth while still servicing the QoS appropriately, and the way this is done is rather clever (at least to me, as someone unfamiliar with this field!)
- There is a first round of sorting of requests based on the requesting port and the QoS level.
These first round requests go into a queue in each channel controller which then sorts them according to traditional memory controller criteria. This is described in some detail, but essentially as you would expect: reads and writes are segregated, and then those which will hit in the same DRAM page are aggregated into what are called affinity groups, stored in second level queues.
- These second level queues are scheduled according to QoS+affinity criteria. These are complicated, but essentially they implement "schedule the affinity group with highest QoS, but include all the items of the affinity group not just the highest QoS item".
- The requests scheduled from these second level queues go into third level queues which are then scheduled to the DRAM according to DRAM criteria (ie so as to meet timing requirements and suchlike).
If you're unfamiliar with memory controllers, it's worth working through the whole thing to see exactly how the details I'm glossing over are handled.
(Apple seem to really like this patent! I found four version of it that all look identical!
I've seen a few other patents where there are two copies of the patent [essentially same words, just the title is changed to emphasize feature A rather than feature B] but this is the only one where I've seen four different features they liked so much they wanted each to get its own title!)

bandwidth allocation

Another way the flowID is used is for bandwidth allocation. The above tries to optimize total bandwidth, but within that total bandwidth we may not want one greedy client using up everything. This begins by also associating a QoS with every transaction. These QoS attributes are used as you would expect, to prioritize traffic at various arbitration points.

2011 basic QoS prioritization, with credits to divide bandwidth

In principle that sounds fine; in practice the problem is always how to implement the idea without too much overhead.
(2011) https://patents.google.com/patent/US9141568B2 *Proportional memory operation throttling* gives an early solution, describing various mechanisms that can be used at different queue levels.
One aspect of the issue is that when bandwidth becomes constrained we want to limit the number of nonrealtime requests that are serviced. This can be done by, eg, preventing the movement of requests from the per-agent first-level queues to the next level of memory queueing.

An alternative or augmentation to this is, when extreme conditions are detected, to start a counter. While the counter counts down, only realtime transactions can be processed throughout the various queueing pipelines, and nonrealtime transactions will simply be frozen in place.

A second aspect of the issue is splitting bandwidth among highQoS devices. The primary idea here is to service the agents using deficit-weighted round robin. A simple way to understand this is we begin an epoch with every highQoS device given some number of token, so eg GPU get 10 tokens, CPU gets 5 token, ISP gets 3 tokens. Each time a transaction is removed from the queue, the appropriate token count is reduced until an agent runs out of tokens. When no-one has any tokens (or there is no request in the queue for an agent still with tokens) we start a new epoch.

2018 better credit-based flow control

(2018) https://patents.google.com/patent/US10437758B1 gives a more sophisticated solution to sharing bandwidth. Many details are omitted (but can obviously be imagined); the idea that matters is that every "request stream" gets a pool of credits, you use up one credit when you make a read request, and the credit gets returned when the read request delivers the loaded data from DRAM. A stream can be fairly flexibly defined using some combination of the agentID, the flowID, and the QoS. The mechanics of this are fairly clear for reads, and the net result is that a quota can be enforced in a distributed fashion at the source of read requests, rather than in a centralized fashion in the memory controller.

For writes it's less clear what to do because there is no natural object that is returned when a write completes.
The patent is somewhat vague on details, but the idea seems to be that
- there is a natural reply to the write *request*. Maybe the NoC protocol is that every NoC packet automatically gets a reply validating that the packet was delivered?
- this reply gives some sort of information about the number of write credits left. (Which helps, but not that much, because how do you know when a write has happened and so you have an additional write credit?)
It's possible that the write issue is just not very important if the memory controller is using a *virtual write queue*, as has earlier been mentioned. Recall the idea is, rather than having the write queue being of limited size in the memory controller, allow it to be of more or less indefinitely large size , with the overflow stored in the last level cache. Conceptually we integrate the memory controller with the LLC, so that all stores are placed in the LLC, at which point they will then hang around indefinitely; but when the memory controller switches to write more it will aggressively try to drain as many dirty lines in the LLC as possible.

2013 vary QoS depending on latency vs bandwidth priority

A few early patents, like (2013) https://patents.google.com/patent/US8963938B2 *Modified quality of service (QoS) thresholds*, and the slightly later (2013) https://patents.google.com/patent/US9019291B2 *Multiple quality of service (QoS) thresholds or clock gating thresholds based on memory stress level*, give a

feel for how these QoS settings are used.
The case is described is of the Display Controller which needs to read data from DRAM (acting as VRAM), to use that data to modulate the screen. You want to keep every transaction at the lowest QoS that will do the job, because low QoS allows the memory controller to aggregate and sort different requests to maximize bandwidth; increasing QoS gives that particular agent better performance, but ruins things for the other agents.

The idea of the earlier patent is that the Display Controller has a local buffer of already loaded data, and is continually requesting new data into that local buffer. If the buffer fullness sinks too low, we increase QoS; then when the buffer fullness rises above a safety level, we reduce QoS.

The second patent augments this idea by having the Display Controller be aware of how busy the Memory Controller is. If the memory controller is busier, then we change the thresholds slightly, so that panic and tagging all requests with a higher QoS kick in earlier, when the local buffer is not quite as empty.

qos priority inversion

Something you should now remember is that whenever you have prioritization you have the potential for priority inversion...

2013 qos-aware merging of memory requests to the same line

(2013) https://patents.google.com/patent/US20140244920A1 *Scheme to escalate requests with address conflicts* deals with that. Suppose you have a pending request for a line from DRAM, sitting in the memory controller queue, and a second request for that same line comes in. The natural thing to do is to “merge” the two requests so that, once DRAM provides the data, it gets sent out over the NoC to both requestors. But doing that naively means that a high QoS request that comes in might be attached to a pre-existing low QoS request for that line, and then just sit in the queue at low priority!
The patent describes various scenarios (depending on how far the request has progressed through the different queues of the memory controller) so as to raise the priority of the request.

2013 flowID aware modification of the qos of requests

Elsewhere variants on this issue are described, both the obvious (transactions that sit in a queue for long enough have their priority raised, to provide some sort of eventual non-starvation), and the less obvious (a flowID that begins with low QoS transactions but switches to high QoS transactions will have, to the extent feasible, earlier enqueued transactions raised in their QoS). Presumably the idea is that
- the flow is a single unit that needs to happen by a deadline,
- the earlier transactions were optimistic about meeting the deadline,
- the new transactions are getting more desperate, and
- we don't want the deadline to be missed because the earlier enqueued transactions are still loitering

around unserviced!)

2012 qos aware upgrading of prior queued in-order requests

Yet a third version of the idea is described in (2012) https://patents.google.com/patent/US20140181824A1 *Qos inband upgrade*, now for the case of lower queues (at the egress of various agents, rather than inside the SLC or memory controller), but the idea is the same – if a priority request enters the queue, and if the semantics of this device are such that requests have to occur in-order, then upgrade the QoS of all earlier requests in the queue so that they will not block this request.

You might wonder how all these various queues are scanned each cycle to figure out the highest priority transaction. The answer is something we have seen before in the Scheduling Queue: a tournament scheme whereby each entry is compared with its neighbor, the winner of each pair being compared against the next pair and so on through $\log_2$(num entries) rounds, as described in (2011) https://patents.google.com/patent/US9009369B2 *Lookahead scheme for prioritized reads*. Once again we see a pattern being re-used across many different problem domains.

I would imagine that all phones have similar concepts (like transactions organized by flowID, and associated QoS attributes) but I'd love to see a comparison with how other companies handle these ideas.

how the L2 assigns qos for CPU requests

You might wonder how the CPU interacts with this machinery of QoS and flowIDs (some of which we have already described, more of which is described soon with the SLC). Most of this machinery is about optimizing bandwidth, whereas for CPUs the primary concern is usually optimizing memory latency. The answer is that the L2 (of each cluster) handles this.

The L2 keeps track of the number of outstanding memory requests (with characterization of prefetch vs demand, read vs write, and code vs data) for each core. The heuristic is that,
- with a low-number of outstanding requests, the CPU is operating in a latency mode, and it is reasonable to tag requests from this core as high QoS (it's what the CPU wants, and there isn't enough traffic to much affect other realtime agents); but
- once there are enough outstanding requests, future requests are tagged as lowQoS (since the CPU is probably now running as a bandwidth engine, but shouldn't be considered a hard realtime device).

This is modulated in obvious ways by the particular request type (eg store castouts will always be low priority, likewise most or all prefetches; whereas instruction demand requests should almost always be highest priority, much more so than load data demands).
This is described in (2011) https://patents.google.com/patent/US8751746B2 *QoS management in the L2 cache*.

DRAM to CPU latency boosters (critical word first scheduling tweaks)

Obviously one of the main jobs of the memory controller is, on a cache miss, to move the data from RAM to the requesting cache/core as fast as possible. Consider what that entails.
A cache line is 64B. Let's say the width of the NoC (Network on Chip) that's connecting different blocks, like the Memory Controller and a Processor Complex, is 32B wide. That means it will take 2 beats (ie transfer operations) to move a line, and those beats run at the NoC speed which is probably half or so of the CPU speed.

An obvious tweak (ie this was already being done back in the late 90s) is Critical Word Forwarding which means exactly what it sounds like – the Memory Controller ensures that the exact "word" (whatever that means in terms of NoC/bus width) requested is transferred first, then the rest of them, and the L1D is set up to forward that Critical Word to the LSU before assembling the entire cache line. Of course this requires the NoC protocol to tell the memory controller not just the desired cache line but the desired address (or at least which half of the cache line is the higher priority).

That's great, but there's a secondary issue, as our CPU's get more complex, of the scheduling on the other side – how does the LSU know when to have that particular load lined up and ready to execute so as to grab the word it wants from the L1D as soon as possible ?
We've already mentioned the concept of Replay, and how dumb replay might retry a failed load every N instructions, while Apple's Replay adds a fake dependency to the dependency vector of the load instruction, with that dependency only satisfied when the L1D has the data available.
Again that's great, but it means we have a cycle or two delay between when the L1D receives the data and when the load executes.

2010 signal the LSU that a load critical word is on its way

So by 2010 Apple have https://patents.google.com/patent/US8713277B2 *Critical word forwarding with adaptive prediction* which has two pieces to it.

(a) The memory controller signals to the appropriate L1D that, in the next cycle, it will be dropping the critical word on the NoC.
In an ideal world this would achieve two things:
- all the machinery that's sitting between the memory controller and the L1D will prepare itself (fully power up bus interface units, buffers, and suchlike) so that we don't encounter any one or two cycle delays for that powering up
- the L1D knows (because it knows the speed of the NoC, etc) when the data should arrive, so it can signal the Scheduler as far in advance as necessary to ensure that the load is ready in the same cycle that the Critical Word data is ready.

(b) But sadly the world is imperfect. The above sounds good, but in reality even though the Memory Controller planned to drop the Critical Word on the NoC the next cycle, the NoC is a shared resource,

another client might have grabbed it, there might be routing congestion along the way, there can be a frequency mismatch between the CPU/L1D frequency and the NoC frequency necessitating a cycle or two clocking delay at the transition between the two, etc etc.
So the second part of the patent is: we have a little agent sitting between the Memory Controller and the L1D which tracks the discrepancy between when the data was promised and when it actually arrived, and which keeps adjusting the delivery time passed on to the L1D to match a best guess based on recent delays.

The patent does not exactly say this, but the above really only makes sense if we assume the NoC actually consists of something like two parallel NoCs, one that is “heavy-duty”, carrying data and most requests, and one that is “light-weight”, expected to rarely be congested, and which carries simple small high-priority messages.

2010 always send critical word first, even if this splits load transactions

That all sounds pretty good. Can we do even better? Well, if one load was waiting on DRAM, what if there's a second load also waiting on DRAM (either on the same CPU, or elsewhere in the system)? Doesn't that second load also want to get its critical word ASAP? Of course it does, and so we get the followup patent a few months later (2010) https://patents.google.com/patent/US20120137078A1 *Multiple Critical Word Bypassing in a Memory Controller*, where the memory controller does as before, for the first critical word, puts the rest of the line aside, sends out the critical word for a second request, repeats as necessary, then eventually gets round to providing all those clients with the rest of their cache lines.
Even this is not the end of the line.
Once you start sending the remainder beats for a line, are you locked into that transaction until the line is done? Or can you interrupt it if a new critical word becomes available? The patent also allows for this second possibility. (This was probably more relevant when, I’m guessing, the NoC width was 16B rather than 32B wide; but it’s certainly possible that future Apple cores will have reasons to load more than 64B in a single transaction, eg if a streaming read workload is detected.)

a strange aside – dynamic DRAM remapping to reduce power

There are other things a memory controller can do. A device frequently has two or more memory “banks”, by which we mean simply standalone DRAM storage devices that can be independently powered down.

Under some circumstances (“performance mode”) one wants all banks to be active, both
- to store as much as possible (less use of compressed RAM, more cached file blocks), and
- to stripe successive memory ranges over the different banks so that, insofar as possible, all banks contribute to providing immediately useful storage, thus lowering latency (more active DRAM pages) and increasing DRAM bandwidth.

But there are alternative circumstances (“efficiency mode”) where one might use lower energy (at the cost of slightly lower performance) by using fewer memory banks and allowing those not in use to power down.

Apple have a patent for doing precisely this, in the context of Intel Macs.
(2010) https://patents.google.com/patent/US8799553B2 *Memory controller mapping on-the-fly*.
Once again I’ve no idea if this was ever used. It’s basically support for hot-plugged DRAM; at the time of a switch down to performance mode, some data is flushed to disk, some is copied from active ranges in the victim bank to available ranges in the non-victim banks, and the memory controller is reprogrammed to map physical addresses from the old to the new mapping of address->(rank, bank, line).
This seems bizarre and something I’ve never heard of hot-plugging DRAM apart from data centers, but the patent is here:

I mention this because it seems like the sort of thing that could also be done (and done more easily, given Apple’s total control) in the context of an M1, shutting down, if appropriate, parts of the DRAM system on either the M1 (which has two such “banks”) to the Pro, Max, and Ultra, all providing ever more such DRAM banks.

System Cache (SLC)

(2012) https://patents.google.com/patent/US9218040B2 *System cache with coarse grain power management* discusses the SLC.
The patent, and the diagram below, suggest that iPhone SoC designs use two memory controllers, and two memory channels. The DRAM bandwidths sustained by iPhones mean these channels must each be 32-bits wide.
I raise this point because the concept of the DRAM channel is one of those computing words that has become useless for most communication purposes:
In the PC world, people seem to think a channel means a 64-bit wide connection to DRAM.
In the ARM Android world they seem to think it means a 16-bit wide connection.
Apple seems to think it means a 32-bit wide connection (which is I think how LPDDR4 uses the term).
Sigh.

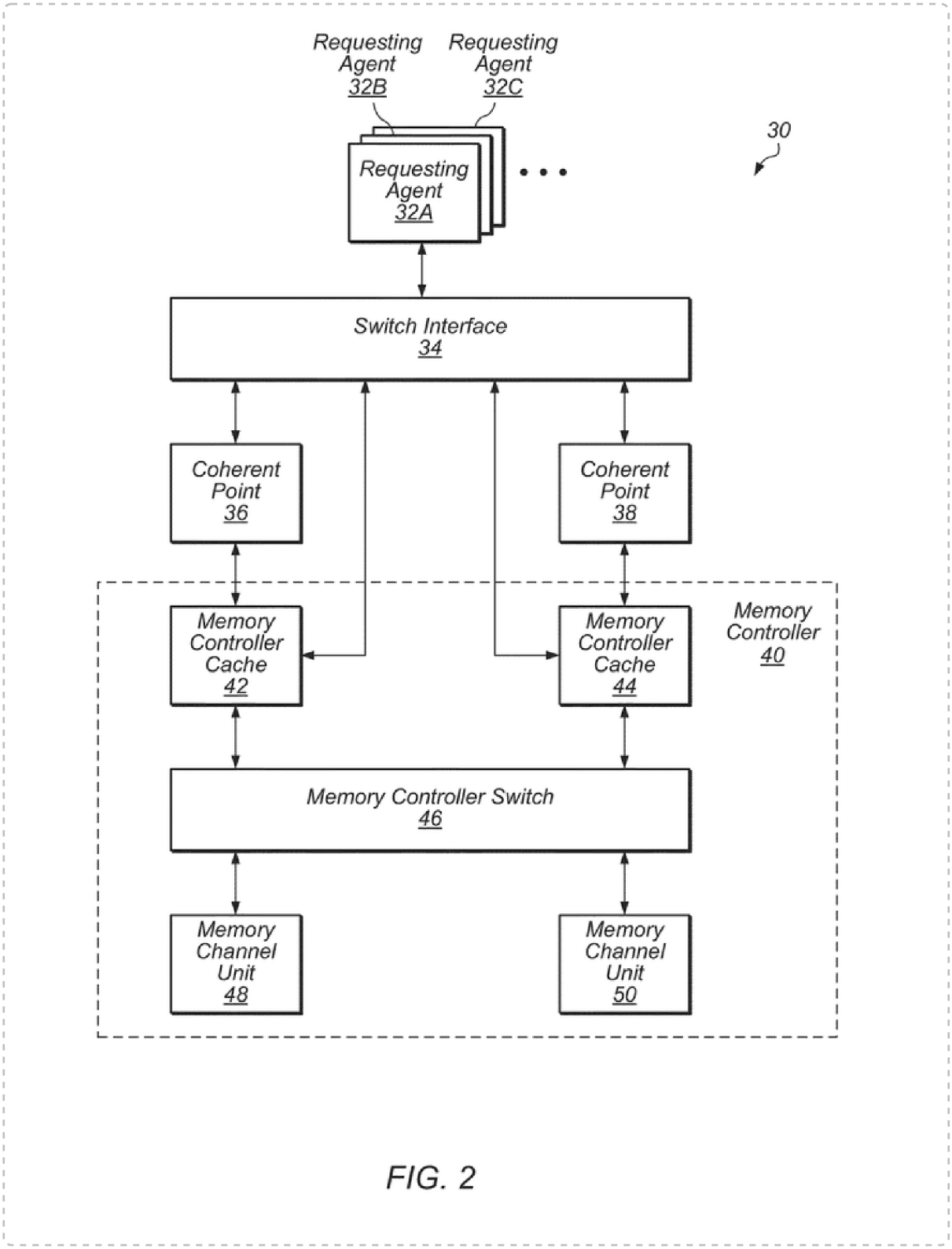


FIG. 2

The idea seems to be, once again, splitting by address (for A14, the obvious choice is to select via

address bit 6, so that even 64B lines and odd 64B lines go to alternate SLC caches; obviously some version of this scales up to any number of power-of-2 caches/memory controllers. One can get even fancier and hash addresses in some way that won't pile up some common patterns, for example addresses that are all an odd multiple of 64B, in one memory controller, should that become worth doing.)

This line-by-line design of the SLC "merges" in the Memory Controller Switch, where requests are arbitrated, and routed to the Memory Channel Units. Details are not given, but I assume the Memory Channel Units are each responsible for one channel of the two channels provided by a single LPDDR4 chip, something like that? Point is, while we have two memory channels, those are not operating at cacheline granularity, unlike all queues, caches, and arbitrators before them.
The iPad design doubles all this, so we have four SLC's and four memory controllers (and two LPDDR4 chips), and presumably forms the starting point for the current M1 design.

SLC sticky lines (cache quotas)

As mentioned above, memory requests are tagged with a groupID (the patent suggests there are 16 of these, 4 bits, though of course this may have risen in more modern SoCs). Note that elsewhere Apple refers to flowIDs rather than groupIDs. FlowID appears to be an evolutions of the concept.
The SLC makes use of these flowIDs, along with various other transaction characteristics, like a *sticky* bit.
This bit is attached to cache lines so that, when it comes to replacement, rather than a standard LRU type replacement, we have the following general properties:

- groups may have SLC quotas associated with them, so that given groupID cannot use more than a certain number of lines.
The details of this quota handling are given in (2012) https://patents.google.com/patent/US20140075118A1 *System cache with quota-based control*, and are more intricate than you might expect.

- lines marked as sticky will persist until they are marked non-sticky.
- when replacement occurs, obviously first priority is to use invalid lines. Next in (negative) priority is LRU lines that are dirty. This may seem non-obvious – LRU lines, sure, but why dirty. The idea is that you may sometimes want to read again from a line that you read "recently" (but LRU), but it's less likely that you will again want to access a line that you wrote recently.
- if we have no lines that invalid or LRU+dirty, then we generate a random number and try to replace that line. If it's sticky, we generate a new random number and try again.

- the GPU is hooked into the flowID mechanism, as are various other obvious candidates like video capture, video decode, or display controller. In fact part of how it works is that all data associated with a particular GPU frame has the same flowID. The next frame, two things can happen:
+ the GPU can tell the SLC to relabel all flowID X lines as now belonging to flowID Y

+ alternatively, if a request is made (by flowID Y) for a line that hits in cache as belonging to flowID X, the ownership will change to flowID Y
It's unclear why both these mechanisms exist, and it may be that Apple wasn't sure quite how this would all play out, so provided both for the OS/app teams to experiment with?

- there are additional flags, detailed in (2012) https://patents.google.com/patent/US20140075125A1 *System cache with cache hint control* that can be used to fine-tune these line allocations. These include
+ a do-not-allocate hint, which means what it says. If the request hits in cache, of course supply the data from the cache.
+ a de-allocate hint. As above, don't allocate the line in cache, but if there is a cache hit, mark the line as LRU+dirty so that it will be first to be removed if a better line needs a slot.
+ a (per flowID) sticky-replace hint. This allows the line to compete against other lines from the same groupID marked as sticky, while not being able to replace sticky lines with a different groupID.

The baseline quota mechanism allows for some over-subscription of the cache, so that essentially you have three levels of persistence via
- stickiness [competing only with your other sticky line],
- then your quota [competing with all your lines],
- then general cache persistence [competing with all lines of all users]
But the mechanism also augments the fine-tuning details. The most important of these is that you can flip the state of a quota from active to inactive (eg at the end of the GPU's construction of a frame). This would seem to be somewhat obvious but there is a twist.
One obvious possibility when a quota is ended is to mark the lines as LRU, so that dirty lines will be (at some point) written to DRAM, and these are lines first to be replaced. But a quota flag provides a second option; dirty lines of a quota that has ended can simply be dropped by having every line invalidated. This makes sense for transient multi-pass constructions, like textures holding normals or shadows, where you have no interest either in saving the state, or even in reading it next frame because it will be re-calculated in the next frame.

The SLC should be seen primarily as a means to facilitate communication at lower energy. Any latency savings are nice (and the system will happily use the SLC as an L3 cache in the absence of anything else going on) but it's designed, and provided with this extra machinery, to *expedite communication across space and time*.
In particular the flowIDs and sticky bits are an attempt to ensure that data required for a purpose, then not required for a while, but which will be required later, remains in the cache rather than being aged out. Think eg of graphics textures required for the construction of a frame every 60th of a second.
Even more precisely, you can have situations like a first GPU pass constructs pseudo-textures (normals, shadows, whatever) and locks them in the SLC, for the use of a second GPU pass some milliseconds later, and within the same frame construction.
Stickiness plus group/flowID's give us the control of a manually managed cache (where you lock lines in the cache so they can never be replaced), but with more flexibility, and delegating most of the detail

work to the cache rather than having the developer worry about it.

line allocation in the SLC

As with any cache, a question of interest for the SLC is how it allocates lines.
One easy, but inefficient, way to allocate lines in a hierarchy of caches is to store the line in each cache as the line makes its way to L1, so that a request that hits in DRAM is recorded in L3, then L2, then L1. As we've discussed, this inclusive cache property makes handling snooping easier. (But obviously it wastes space, and a better solution is just to replicate tags at each cache level, not whole lines.)

Suppose we don't do that? As far as I can tell, mostly Apple allocates lines right in L1. When a line in L1 is eventually replaced, it will be moved to L2 (ie L2 is a victim cache for L1), and likewise when a line in L2 is replaced, that line will be moved to SLC. One exception to this overall design is prefetching into the L2.

This scheme is easy to understand but leaves many questions unanswered, for example
- does an L1 hit in L2 copy the line to L1 or move it to L1?
- can lines move directly from one P-core to another (as a result of snooping) without having to pass through L2?
- similarly can lines pass from the P-cluster to the E-cluster without having to pass through SLC?
- apart from the CPU, how do other agents have lines allocated in the SLC?

(2012) https://patents.google.com/patent/US20140089600A1 *System cache with data pending state* appears to answer at least the last of these questions.

Consider any cache that allocates lines on a miss, when a new request comes in. What do you do with the pending request while you wait for the data?

- Well the easiest, but also worst-performing decision is to block until you have the data, but clearly that's no longer acceptable.

- Next option is to put the request in some sort of queue while you wait for the data. But this has the consequence that for every subsequent miss, you have to check against that queue (because if you've already sent the request to DRAM, you don't want to send it again).
That means checking a large associative structure, with attendant power issues.

- One could imagine different solutions to this, but the solution Apple chose (at least, as of 2012...) was to allocate the line without the data! Rather than just being marked invalid, the line is marked as "pending". Then any subsequent request for this line will also see it as pending. All requests to pending lines go into a replay buffer where they wait till their line arrives. Neat, no?
(Presumably they are awoken by some sort of selective wakeup, the same way we have seen this done for instruction Replay. An obvious scheme, for example, would be to hash the desired address down to

an appropriate length – 8 bits? – and wake up all requests matching that hashed address when the line comes in.)

The way this mechanism is described in the patent suggests that by default non-CPU requests are allocated in the SLC on read of a line, though there is probably modulation of this both for the obvious cases (devices like a GPU with their own cache; or flows that are marked as streaming and so will not benefit from taking up SLC lines).

decoupling of lineID from tag layout

(2006) https://patents.google.com/patent/US7624235B2 *Cache used both as cache and staging buffer* looks like some uninteresting technical details, but don't be fooled!
Th patent appears to have the SLC as its primary concern, but the ideas are interesting and could in principle be used in the L2 (possibly even, at least for some purposes, in the L1).

The nominal idea is that, rather than providing the cache with some number of various special purpose buffers (in particular buffers handling IO transactions with unusual characteristics), the general lines of the buffer can be used for this purpose. This requires a mechanism to indicate that those lines are being used in a special way, which is easy enough, just define a new cache state.

But the idea raises a concern – given the set-associative nature of the cache, what if a particular set is heavily used by IO, crowding out all the ways that should ideally be used as cache?
This is explained by a second point which is far more interesting than the patented point!

What is actually set-associative in this cache is only the tag storage; tag storage provides a lineID describing the placement of the line, but the connection between tag and lineID can be somewhat arbitrary.
The patent describes how using a line as a staging buffer (ie IO) purposes does not require a tag, and so using many lines in this way will reduce the cache capacity a little, but no more than that.

But of course, once you have severed the link between line placement and associativity all manner of possibilities open up! These include easily allowing parts of the cache to become drowsy, or totally powered down; or allocated for different purposes (eg per CPU QoS) while still retaining placement flexibility.
This ties into what has become a constant theme related to all the caches – the variety of hashing options that are possible, the evidence that Apple is using non-trivial hashes, and tricks that can be played with a cache that once you have clever, separated, control between the tags and the data lines, from segregating streaming data, to snoop filtering lower caches, to prioritizing the retention of certain classes of lines, to using pure tag (no data) to indicate a zero'd cache

line, to marking lines as allocated but not yet filled.

(This use of cache as a staging buffer is easier understood by comparing it with (2005) https://-patents.google.com/patent/US7412555B2, *Ordering rule and fairness implementation*, which describes an earlier version of the design, where a distinction is drawn between an IOC [IO cache] and IOM [IO memory, used as staging buffer].
A year later the design has evolved to consolidate the IOM and IOC as a single storage pool.

The 2005 patent itself is interesting in that I have never seen anything like it before. The problem to be solved is that buses, like PCIe, have various types of transactions and various rules for when these transactions may or may not be handled out of order. One wants to follow all the rules, necessary for correctness, while also boosting performance by taking maximum advantage of whatever ordering flexibility the rules allow. The patent describes a way of doing that, at least to some extent, with minimal additional logic.

Later below we will revise this problem – maximizing the performance of a stream of memory requests while honoring ordering requirements – in the context of the SLC.

SLC power management (drowsy or powered down ways)

All this background and asides has taken us away from the actual goal of (2012) https://patents.google.-com/patent/US9218040B2 *System cache with coarse grain power management*, which is the same sort of energy saving scheme we saw earlier with the L2
- track how large a fraction of the SLC is "really" being used
- limit the number of active ways to match that fraction
But the details differ. In particular the power-down granularity is by-way, not by one third of the cache; likewise different are the details tracking when to grow or shrink the cache. While the L2 cache provided some hints as to how the power-down is managed, we are not told this for the SLC.
It's unclear, at least to me, how the earlier patent I mentioned, decoupling tag layout (as sets and ways) from data layout (as an unstructured pool of lines) connects with this design. One obvious possibility is that, to use reasonable numbers, the SLC is 16-way associative, and so the data SRAM is split into at least 16 subarrays. Powering down or sleeping one way means shutting down a particular section of the tags, and (after some cleanup in the tags to invalidate lines appropriately) one of these particular subarrays?

The above energy saving patent soon gets two minor updates.
First is (2013) https://patents.google.com/patent/US20140297959A1 *Advanced coarse-grained cache power management*. We add two tweaks.
- A timer for some hysteresis, so the system doesn't constantly toggle between N and $N - 1$ active lines
- More careful sequencing of how the lines of the cache are powered up (rather than powering them all up at once) to reduce current draw and system noise.

We also get (2013) https://patents.google.com/patent/US20140298058A1 *Advanced fine-grained cache power management*. Fine-grained (vs coarse-grained) in this case refers to voltage. We retain the same

sort of ideas as before, for tracking cache usage and suchlike, but we allow an intermediate sleeping state between ways that are fully powered up and those that are fully powered down. So basically upgrade the SLC to a drowsy cache.
One additional tweak beyond a standard drowsy cache is that requests to the cache can be stored in a queue for a few cycles until it's felt appropriate to wake the relevant drowsing region of cache. This delay process is made more feasible by the QoS attributes attached to each request, which allow the cache controller to trade off urgency vs the energy savings of possibly consolidating multiple requests to a single cache region wake up.

SLC controlling other caches

Another unexpected aspect of the SLC is that it can act as a "puppet-master" controller for other caches. Consider (2013) https://patents.google.com/patent/US20150143044A1 *Mechanism for sharing private caches in a soc*. The idea is that
- multiple IP blocks on the SoC have caches
- sometimes those IP blocks are using none or only a fraction of their cache (eg the block is powered down)
- in which case, why not reuse that cache for some other client?
The SLC controller is tasked with doing this and making it work.

Your immediate reaction is probably to think of this as something like the ISP or Media Engine is being unused, so maybe the CPU gets to use some of their cache as an augmentation to the L3 of the SLC. But Apple actually give as example a case somewhat backwards from that, suggesting that the GPU might want access to more cache, and the CPU may be using only some part of the L2 and can let the SLC/GPU use the rest.
Presumably the primary win in this is energy (it's always more expensive to go off-chip) but there's also some latency win, maybe 50 ns or so (?) to get the data via a trip all the way to the SLC then rerouted to a private cache, rather than 100ns to DRAM.

You might wonder how this is actually done. At least part of the answer (which has been alluded to in some other sections above) is that the SLC maintains two sets of tags, one set being the traditional cache tags for the SLC, the second set being tags for all the other "relevant" caches in the system. These two are tested in parallel.
In some sense the point of the second set is coherency, to allow the SLC to act as a single coherence point rather than requiring every transaction to visit every cache, each one testing whether the transaction matches what's in that particular cache. But as Apple have realized, if you are performing this sort of test, you can add a little more data to those "remote" tags, initially to redirect a request to the appropriate cache, but then to control the cache and so populate it with data as you see fit.

SLC as a coherence point

This set of parallel tags and how they are used is described in (2018) https://patents.google.com/patent/US20200081838A1 *Parallel coherence and memory cache processing pipelines*. This looks scary to

read, until realize that the "parallel coherence pipeline" simply means this second set of tags that is probed in parallel with the normal SLC/ memory cache tags.
(Other systems may do this by forcing the outermost cache to be inclusive of all inner caches, which has the same effect, providing a single location to test coherency. But it's clearly more efficient to allow the caches not to be inclusive, and simply have the outermost cache replicate the tags of all inner caches, not the tags *and* the data.)

Sitting at the interface between the SLC+memory controller and the rest of the system is the Coherence Point. Even though the details get very technical, there are still some interesting aspects to it, as described in (2012) https://patents.google.com/patent/US20140173218A1 *Cross dependency checking logic*.
Recall first that different agents on the SoC have different ways to enforce ordering of memory transactions. Most ordering will happen at a very low level via MOESI state changes in the various caches; some will occur via explicit barriers in user code; some will occur via specific mechanisms in each agent (GPU, ISP, PCIe, etc) as indicated by the OS driver. Put that aside for now, just assume that, at some low level, memory transactions have been generated so as to follow some ordering. Given this fact, but given also that we are constantly reordering transactions to optimize QoS and throughput, how do we preserve order? The answer is (simplistically) the Coherence Point. More or less we require that transactions are generated in-order (relative to the requirements of the generating agent), transported to the Coherence Point in-order, have ordering stamped on them at the Coherence point, and can then only be reordered (via transport to the SLC, or to another agent, or to DRAM) in conformance with those ordering requirements.

how the strand ordering is implemented

A second element of this big picture is, as we have already mentioned, we can enforce ordering via flushing (extremely expensive, terrible and to be avoided at all costs), via barriers (much better, but still impose more ordering than really required), or via strands.
The Coherence Point essentially operates via the strands model.

So let's give an example of what we're trying to do. For simplicity, assume only two agents, the P-cluster and the E-cluster. These each have an L2 that moves lines between the L2 and the SLC/DRAM. There are two obvious types of transactions, namely reads (cache requires data) and writes (ie cast-outs, cache is writing back data either because that cache-line needs to be re-used by new data, or because the MOESI protocol has informed us that some other agent wants to read an address from this modified cache line).

In addition to various house-keeping transactions, there is also one interesting, non-obvious transaction type, the victim. The SLC can be used by the L2's as an L3 cache, but not in the way this is usually done by x86. Specifically, how do lines land up in an L3 cache?
We discussed this before, but to remind you: One answer is that a request from a core goes all the way to the memory controller, the data is returned *through* the L3, to the L2, to the L1, being deposited in each of these caches along the way. As mentioned, this enforces inclusion and is simple, but obviously

also duplicates lines so is sub-optimal use of the available storage. The alternative Apple uses is that the line first goes to the L1, then when L1 needs to use that space the lines moves out to L2, and for some substantial period of time exists only in the L2. After some period of non-use of that line, a new address will be referenced that wants to use that line in the L2, and the existing line (called a victim) will be transferred to the SLC, where it will sit for some time. Eventually, maybe it will be re-referenced by a core, or maybe it will dropped from the SLC.

Now you can already start to think of various interesting questions and ways this might be optimized (for example we know lines can be tagged as streaming, and clearly streaming lines might be subject to different placement/replacement protocols; or when another agent [eg the E-cluster or GPU] touches a line in the SLC, should that line be *copied* to the new agent, or *moved* to the new agent?)
But that's not our interest here; our interest is in the fact that this victim protocol is implemented by an optimized NoC requests which looks somewhat like a cast-out (writing back a line of data) except the cast-out is not necessarily modified data so it doesn't, eventually, have to land up in DRAM.
So an agent can generate a
- read request (address, no data) or a
- write request (address, line of data to be written) or a
- victim request (address, line of data, plus a flag saying "move ownership of this address+data from cache A to cache B")

These requests from various agents make their way to the Coherence Point. The Coherence Point was (2012) split into two halves, surely more by now on M1, then Pro, Max, and Ultra.
Each of these two (2012) halves holds a surprisingly large table of outstanding transactions (in the 2012 patent, the example given is 64 pending transactions). Each entry holds at least one address (read or write castout) and possibly two addresses (victim read address and victim write address are handled as two distinct addresses).

Now the problem we have is, assuming requests come in correctly ordered, we have to ensure we don't break that ordering in subsequent processing. This is done by matching each (possibly two for victims!) address in a new transaction against each (possibly two!) address in the table of pending requests. If we find a match, we create a linked list of transactions, so that the earliest transaction points to the next later one to the next and so on; the newest transaction that matches is added at the end of the list.

So to return to our E vs P-cluster example, suppose that an E-core wants to write to a line that is owned by a P-core. After all the MOESI work has been performed, at the Coherence Point we'll have an earlier transaction from the P-core, casting out the line (ie writing to the SLC), and a slightly later transaction from the E-core reading from the address of that line. Clearly we do not want these two transactions to be swapped in order, ie for the read (from SLC or DRAM) to occur before the write of the castout data into the SLC.

So this gives us our strands, with the rule that every strand can be processed independently, but the items in a strand (ie forming a linked list) must be processed in link order. Each time we consider the next item to extract from the Coherence Point into the SLC, and possibly on to the memory controller,

our first constraint is that we only allow stand-alone (ie non-linked) requests, or requests that are at the head of a list. From this restricted pool, we then use whatever additional prioritization scheme we want, whether looking for those that match a flowID, or that have the highest QoS, or whatever.

The patent doesn't say what happens next, but the obvious design would be that the next item in a linked list cannot be passed on to the SLC until the SLC confirms completion of the prior item; since preventing two items from the list being in the SLC or memory controller queues at the same time will prevent their being re-ordered relative to each other as part of those SLC and memory controller queues.

(The patent does not discuss this in any detail, but one diagram suggests that items can also be slotted into a second set of linked lists based on flowID. My guess is that a flowID can be configured by a driver as ordered or not, so that ordered flowIDs will have the same constraint imposed on them – as part of a linked list based on a flowID, only one item of a flowID at a time will ever be eligible for processing within the SLC, and those items will naturally be presented in-order.)

strand ordering in other places

Once you know this pattern, you see it in other places. For example (2018) https://patents.google.com/-patent/US20200050548A1 *Establishing dependency in a resource retry queue* has slightly different details, but the same essential issue: we want the performance freedom of being able to execute operations out of order, but we also need a way to enforce some ordering.
For this patent a cache (think L2 or SLC) can have both external requests and internal operations (like flushing out data prior to powering down some part of the cache). These various requests are enqueued, and dependencies are detected so as to ensure that if these different operations refer to the same address, they are added to the end of a linked list based on that address; and once again requests are scheduled based on priority, age, etc, but with the constraint that we cannot break into the middle of a linked list of requests.

Large vs Small core issues (some A10 history)

There’s not much particular to say about Apple’s use of large vs small cores. However there is some interesting historical trivia.

You will recall that the A10 had the interesting design that the large vs small CPU was essentially invisible to the OS, meaning that a single pseudo-CPU presented itself to the OS, but toggled between the large or small core depending on circumstances. This sort of pairing (large tied to small, rather than a cluster of large’s tied together and a separate cluster of small’s tied together) has been considered in the literature in other contexts (for example to save power by having the CPU switch to the small core when executing low-IPC code [either lots of unpredictable branches, or lots of misses to DRAM), but it’s not clear that there’s any point in doing this when OS-transparency is not essential.

One suspects Apple did this as a one-time experiment but with an understanding that clusters were the

long-term goal. Given this, they seem to have considered the A10 small core as something they could experiment with. What sort of experiments?
Well look at (2014) https://patents.google.com/patent/US20160147290A1 *Processor Including Multiple Dissimilar Processor Cores that Implement Different Portions of Instruction Set Architecture*. The idea is simple enough – put a limited ISA on the small core, and swap to the larger core when it's necessary to implement the ISA that's not on the small core. (Seems simple, but you'll note that Intel seems incapable of doing something like this for Lakefield.) The complication is that the swapping has to be done by the cores themselves because, remember, the OS only sees one core!

But that's not the interesting part. The interesting part is what's the part of the ISA you want to omit? We know, for example, that Firestorm and Icestorm apparently both implement the full Apple version of AArch64, even up to both implementing AMX. So what's omitted in the A10? The very cool answer is 32-bit support! That seems to have been part of the additional value Apple extracted from this one time-experiment, a core that ran only 64-bit on which they could presumably test OS modifications, see how much design could be simplified by omitting 32-bit complexities, etc etc. One wonders the extent to which the A10 Zephyr core (small core) was in fact a prototype for the micro-architecture of the A11 and subsequent family, rather than being a simplified version of the A10...

Their second patent in this space is (2015) https://patents.google.com/patent/US20170068575A1 *Hardware Migration between Dissimilar Cores*. This tells us essentially two things:
- the dual-CPU presents itself to the OS via its power states. The OS decides (based on its reasons) upon a power state for the pseudo-core, and the cores decide which one should be active based on this power state. This means they have to transparently move state between them; obviously the user visible registers, but also a few SPRs. The bulk of the patent describes how this is done, but it's not especially enlightening or generalizable.
- a dedicated path was provided between the two cores for transferring registers (as opposed to cheaper, but less performant, alternatives one could imagine, like transferring the register state through the L2)

Smart (functional) DMA

As we move further away from the CPU, we're covering territory about which I know less and less. Even so, there are still interesting things one can learn. For example consider
- (2005) https://patents.google.com/patent/US7620746B2 *Functional DMA performing operation on DMA data and writing result of operation*
- (2007) https://patents.google.com/patent/US20080222317A1 *Data Flow Control Within and Between DMA Channels*
- (2008) https://patents.google.com/patent/US20090248910A1.
Central dma with arbitrary processing functions.

The 2005 patent covers the idea of adding functionality to a DMA controller, so that the DMA can perform some function on the data as it streams through. The ideas suggested for this include data transformations like crypto, or reductions, like a CRC or a hash.

The 2007+2008 patents build on that by allowing the chaining, ordering. and dependency, of DMA descriptors so that the previously mentioned functionality can be stacked. The example given is something like a network stack where the TCP layer creates a checksum, the IP/Sec layer encrypts, and calculates a hash, and the ethernet layer calculates a CRC.
The point, of course, is that even though one doesn't think of it, there has been a TCP-offload engine in your iPhone from the early days!
One thing that's not clear to me if this mechanism is an ideal way to perform transformations that grow or shrink the data (ie compression/decompression, or codecs) as opposed to a more traditional accelerator scheme (ie pass an in buffer pointer and an out buffer pointer to the accelerator, and wait for it to give you an interrupt once it is done).

The obvious next stage, after the above bulk transformations, is the appending/prepending, and removal, of networking headers and the other paraphernalia of a full TCP offload engine, and we get that in (2008) https://patents.google.com/patent/US8359411B2 *Data filtering using central DMA mechanism*.
An interesting point, to which the patent draws attention, is: by placing this machinery in the DMA engine, not the networking hardware (eg in an ethernet chip) it becomes available to all network interfaces that use TCP. This is obviously nice insofar as it shares the HW across WiFi, cellular, even a Lighting/USB ethernet connector, but is especially cute when you think of something like Thread (Bluetooth radio, but using IPv6) which refits TCP/IP to a protocol that never before used it and is unlikely to have this offload on any chipset!

An especially interesting variant on this idea of "active" DMA occurs in (2008) https://patents.google.com/patent/US8610830B2 *Video rotation method and device*, which has the DMA engine between a local video decode buffer and display RAM reorder the data during the transfer so as to handle the four different (portrait vs landscape, top vs bottom) possible orientations of the display, though one suspects every aspect of this particular design, up to and including the swizzling DMA transfer, is now obsolete.

We discussed earlier DMA that can route data directly into the cache. We've discussed above DMA that provides for faster networking. Why not glue these two ideas together? 🤣
(2006) https://patents.google.com/patent/US7836220B2 *Network direct memory access* discusses networked DMA.
There is a standard for this already, called RDMA, but what Apple proposes is lighter-weight. (RDMA builds on TCP/IP, so it can be routed across networks. But if you don't need that degree of routing, you just want to work on the same ethernet, then, you can strip out the IP and TCP routing stuff.
The idea seems to be that machines A and B will each create a range of memory visible to the other machine, and essentially be able to DMA from one cache to the other and back via smart DMA with no CPU involvement!
Is this of any value? Did they ever do anything with it (or plan to)? Was it part of PA Semi's business plan that was abandoned after the acquisition? Is something like this already being used behind the scenes when Apple devices talk to each other locally (eg iPhone to Apple Watch, iPhone to Apple TV, Apple TV to AirPods)?
Who knows!

Maybe we'll never see it in a consumer product, but it will be part of whatever they plan for their data center hardware (and of course you know they are moving to their own data center hardware!)
One way to think of this is it's (not exactly, but analogous to) the network equivalent of Unified Memory; a way to strip out most of friction of networking so that it's much easier to talk to a remote device much like talking to a local peripheral.

NoC issues

I'm no expert on NoC! This is my rough summary of what I think is the significant part of some patents.

data transfer between clock domains

A constant concern is transferring data between clock domains. We have
- (2005) https://patents.google.com/patent/US7500044B2 *Digital phase relationship lock loop*
- (2007) https://patents.google.com/patent/US20080198671A1 *Enqueue Event First-In, First-Out Buffer (FIFO)*
- (2015) https://patents.google.com/patent/US20160328182A1 *Clock/power-domain crossing circuit with asynchronous fifo and independent transmitter and receiver sides*

The basic idea in transferring data between clock domains is that
- you have a queue of buffers straddling the two clock domains
- you write into the queue on one side and
- you read from the queue on the other side
There are many technical details (especially if the two clock domains are also different voltage domains) but so far so good. However within such a scheme we need to ensure that we don't under-run or over-run the queue. The read side and write side both need to know the current read and write pointers of the queue to ensure that this. And so we need to communicate, across the clock domain, whenever the read and write pointers changed.

The traditional way to do this is state-free, so it can safely transfer data between any two clock domains without knowing anything about the clocks, but the cost of this is some delay (at least two "local" cycles) while the relevant circuits that are synchronizing one side with the the other stabilize.
The 2005 patent points out that there is no reason to demand a state-free solution, and if you take advantage of the fact that you know that clocks have predictable transitions and record some history, you can then predict the transitions of the other clock domain and when it is safe to transfer data from one side to the other, without the enforced delays of the traditional scheme.

The 2005 solution transfers read and write pointers into the queue but becomes expensive (because of the gray coding required to make it work) as these read/write pointers become longer.
The 2007 solution deals with this by creating a hierarchical solution. We have a first queue of transfers buffers which can be fairly large, along with two secondary queue of "event" buffers, each of which is small (say 4 or 8 buffers). Every time the read or write buffer is incremented, that "change event" is entered in the event buffer, and data is transferred through the event buffer from one domain to the

other. The insight is that neither domain really needs to know the full read or write pointer every time a change occurs; it only needs to know that the pointer has been incremented (a single bit of information); and a very lightweight FIFO can transfer that information, using the digital phase looped locking of the 2005 patent.
The 2015 patent then updates all this so as to save power by allowing all but the most essential elements of the design to go to sleep as frequently as possible during the transfer process

split NoC transaction: presentation vs arbitration

(2005) https://patents.google.com/patent/US20070038791A1 *Non-blocking address switch with shallow per agent queues.*
A second NoC concern is arbitration. The big idea of this 2005 patent seems to be split the “presentation” of a NoC transaction from arbitration for the NoC. As I understand it,
(a) you have something like a "local" connection from every agent to, let's call it, the central fabric, with easy, fast, lightweight arbitration over this local connection.
(b) this local connection feeds into a *queue* attached to the fabric, not directly onto the fabric.

The end result of this is that every agent can (mostly without drama or contention) throw a succession of requests at the fabric and have them immediately queued in a per-agent queue. Later arbitration (based on the usual decisions of transaction type, QoS, etc) will extract requests from these queues and send it onto the fabric.
This is as opposed to a bus-like scheme where you first ask for access to the fabric, then, once it is granted, throw out your request; but until you are granted accessed you, the agent, have to hold onto a particular request.
An agent could have a local queue, so that it could continue to work while enqueueing requests until granted access. But by moving these queues to a central location, the arbiter can make better decisions, taking into account the entire state of the system.
(The cost of this is that you need something like a second set of dedicated wires, parallel to the NoC wires, to transport requests from each agent to the central queues. But wires, like transistors, are cheap if you have enough metal layers.)

structure of NoC arbitration

Fifteen years later we get an update. (2020) https://patents.google.com/patent/US10972408B1 *Configurable packet arbitration with minimum progress guarantees*, which explains how the central arbiter does its job. Obviously the goal is balance giving priority to high QoS packets and balancing bandwidth across devices against preventing starvation (ie limiting the maximum delay of a low QoS packet).
We’ve seen various mechanisms suggested for this type of problem in various places, but the 2020 version seems the most elegant so far, providing this balancing without requiring age tags attached to the various enqueued transaction.
To simplify, the mechanism includes two ideas:
Firstly we have an epoch (some number of cycles) for which each traffic class is given a “minimum

progress guarantee" counter. At the start of the epoch, arbitration is some sort of fair system (eg round robin) across all classes for which this minimum progress guarantee is positive, and each time a class gets a grant, its counter is reduced.
Once all the counters are zero, for the rest of the epoch we switch to a different scheme based on weights. Each class is given a weight (eg 100 for Real Time, 20 for Low Latency, and 1 for Bulk traffic). Each cycle the weight is reduced by 1, and the class that has reached 0 gets the grant, and has its weight reset to the initial value. This obviously allocates bandwidth by proportion, subject to obvious points like if there is little to no Real Time traffic then Low Latency and Bulk will share most of the bandwidth in the 20:1 ratio); while prioritizing Real Time when a Real Time packet does come in.

The set of weights (and various other aspects of the algorithm) is tunable by the OS, but to me that's less interesting than the general shape of the algorithm.

connecting from Apple Fabric to PCIe or AMBA

ordering rules

There are some patents about enforcing (or relaxing) ordering between different buses
- (2005) https://patents.google.com/patent/US7412555B2 *Ordering rule and fairness implementation* for communicating with PCIe.
The problem in this case is that
+ PCIe allows for some degree of operations (of different types) to be re-ordered, and obviously one wants to take advantage of that, but
+ the simplest way of handling the re-ordering, via independent (per ordering class) queues, can lead to starvation of the lowest priority ordering class.
+ the natural solution to that might be some sort of timestamp/age, so that low priority transactions that are too old are escalated in priority, but Apple claim that the precise semantics of PCIe require that this age counter be extremely long in practice (and of unlimited length in theory), hence
+ the patent describes an alternative "age-like" mechanism that describes the relative age of different transactions rather than age relative to a single timebase.

- (2012) https://patents.google.com/patent/US9229896B2 *Systems and methods for maintaining an order of read and write transactions in a computing system* for communicating with AMBA.
Like many of these older patents, this may no longer be relevant! The issue, at the time, was moving transactions between AMBA and Apple Fabric (at that time). Specifically Apple Fabric (at that time, still?) used a single bus for read and write, while AMBA AXI used one bus for read transactions, a second parallel bus for write transactions. As I say I am no expert, but it seems to me that you can use such a design when you have a point-to-point connection (so that it's clear which bus is read, which is write, whether you choose the point of view of the peripheral or of the CPU's), whereas with a distributed fabric such a split makes no sense, rather you just have a set of connections that can move stuff from any point A to any point B, and if you want the increased performance of a split bus, you either widen your single bus or created a second parallel fabric (like some Intel Xeon designs used two parallel rings,

look at either the E52600 v2 or E5 2600 v3 designs here: https://www.anandtech.com/show/8423/intel-xeon-e5-version-3-up-to-18-haswell-ep-cores-/4 .)

OK, you say, so what? Well once again this affects ordering. A stream of reads and writes on the single bus has a natural internal ordering that is lost when the reads are transferred to a read-only bus and the writes transferred to a write-only bus. One has to be careful that read transactions that need to occur after a write transaction are not allowed to be transferred to the read-only bus before the write transaction is completed.
One can imagine various ways of enforcing this rule; Apple describe a solution (based on counts of outstanding transactions) that allows for separate such queues. It's not clear to my eyes why this is a better solution than some others; maybe it was easy to implement given the rest of the bridge design? Or maybe it's a workaround because the more obvious solution was patented?

design of a PCIe controller

(2007) https://patents.google.com/patent/US8284792B2 *Buffer minimization in interface controller* is about the design of a PCIe controller. The primary problem to be solved appears to be
- the flexibility with which PCIe can aggregate multiple lanes (x1, x2, up to x16) to a single port,
- along with the fact that the spec allows the lane ordering can be reversed.
One way to deal with this flexibility is to aggregate data from the SerDes into a buffer, then permute the buffer contents as appropriate; but the Apple solution (lower latency) is create a programmable tree of muxes that, appropriately programmed, will perform the permute as the data flows through them.
Presumably this was inherited from PA Semi and slept for a few years, until the iPhone 6 (2015) shipped with a PCIe SSD.

qos -based snooping

Another thing Apple do (which seems like overkill, but presumably is born of experience) is that all requests sent over the NoC have a QoS attached to them. That seems not *that* surprising in the case of basic memory requests, and we discuss below, in memory controller, how this QoS is used to order requests in the memory controller. But something additional you may not have thought of is that this QoS is also used to prioritize snoop requests!
Perhaps in future the mechanism may become more aggressive, but as of 2018 https://patents.google.-com/patent/US10795818B1 *Method and apparatus for ensuring real-time snoop latency* the mechanism is fairly simple, mainly that each snooping machinery has a queue of incoming snoop requests, and once a priority snoop enters the queue, only a limited number of non-priority snoops are allowed to be serviced before the priority snoop is serviced.

It's interesting that this 2018 patent refers only to CPUs, but in that context refers to low-latency vs bulk snoops. How this distinction is drawn is unclear.
An obvious, simple, first pass could be to prioritize P-cluster snoops over E-cluster snoops. Slightly more aggressive might be to use the QoS of the currently executing code (this could, eg, be stored in a

special purpose register in each core, updated by the OS at context-switch), and even better would be to also attach a “use case” to the QoS, so that, for example, prefetch’s are marked as a lower QoS transaction than load-related snoops.
It also seems likely that the same mechanism is applied not just to CPUs, given that GPUs, NPUs (even the ISP and most other peripherals now?) also want to snoop?

optimally handling qos when crossing bus/noc bridges

Everything we have described so far is built around a QoS identifier, and superficially this seems like enough. Fire off a transaction that's marked as PRIORITY (in fact Apple seems to use three QoS levels that are frequently referred as Red [realtime], Yellow [non-realtime or if you prefer, soft realtime], and Green [best effort bulk transport]) and it gets sent to its destination as fast as possible. What more do you want?

Well the reality is that packets often have to cross multiple buses and boundaries to get from here to there. For example a packet may originate on a PCIe bus, be moved onto the NoC, then moved off the NoC to a USB bus. There’s also a hierarchy of the primary NoC feeding to each cluster (eg one E and two P clusters) as a single agent, only to have that cluster in turn have a local interconnect communicating between say four cores, L2, AMX, and any other cluster-specific resources.
At every transition between interconnects (and possibly at some internal locations of each of these interconnects) the packet may be placed in a queue. Even though the packet is sitting in the PRIORITY queue, and will be sent to the next stage as soon as possible, it may be delayed while prior transaction(s) complete.
Now the packet is on the next bus, it needs to be routed, and will be treated as PRIORITY again at the queue, but again there may be a delay.

The point, you should note, is that if all you have is a PRIORITY flag, that does not record how much an individual packet was delayed at intermediate stages.
In an ideal world, you would want to move a long-delayed packet ahead of all the other PRIORITY packets for all subsequent steps of a multi-step journey; but doing this requires additional information attached to the packet. What you want, in fact, is something like a timestamp. But that's difficult to implement in a distributed fashion, across multiple devices and buses all running at different frequencies.

Easier to implement as a practical matter is a concept of "urgency" which is filled in (and utilized) at the buffers and scheduling between buses, rather than being established at the end points. This urgency concept (essentially "this packet was delayed in a buffer for an unreasonably long time, so make up for it in subsequent routing/scheduling/arbitration decisions") is the subject of (2018) https://patents.google.com/patent/US20200057737A1 *Systems and methods for arbitrating traffic in a bus*.

Power issues

I'm no expert on power! This is my rough summary of what I think is the significant part of some patents. (Unsure if I will keep this section.)

Along with all the other stuff above, we have a few power-specific patents.
The on-going idea seems to be to centralize ever more power control in one place. This may seem obvious, and some of the patents may seem ridiculously dumb; I think they have to be placed in a context where a phone (even an iPhone) didn't consist of a single SoC with (almost) everything on it designed by Apple, but consisted of multiple different chips from different vendors that had to be tied together as best possible.

We start with (2007) https://patents.google.com/patent/US8645740B2 and https://patents.google.com/-patent/US7711864B2, where we have two big ideas
- central management
- management based on measuring the actual power draw of each component rather than some sort of spec or theoretical model.
But this central management is all done by SW on the CPU.
There's also (2007) https://patents.google.com/patent/US20080168285A1 which is surely obsolete, and has to do with ensuring that a CPU or GPU has finished executing all the current instructions before it's powered down.

By (2009) https://patents.google.com/patent/US7853817B2 we've evolved this to
(a) perform (some) power management using a dedicated power management controller. (As opposed to having the CPU power up and down various pieces of hardware, which means paying the energy cost of constantly waking the CPU up).
(b) make this power management controller aware of various dumb ways in which attached hardware needs to be put to sleep/woken up, because so much hardware hasn't yet transitioned to the modern unified way of handling sleep/wake.

This is followed by (2010) https://patents.google.com/patent/US8271812B2 where this power manager now sees the world as a set of domains (like the CPU domain, the video decoder domain, the audio playback domain), each domain has associated with it a set of performance states (which at the high end of degrees of speed, and at the low end are degrees of sleeping, with more sleep meaning a slower wakeup). Each domain, and each performance state, have associated info describing the control registers and their values to force that state.
Obviously this gives more centralized control, and is more flexible to set up and going forward, but a particular concern Apple has is to ramp up various separate functionalities in lock step. So that, eg, if we slow down the CPU we also slow down the L2 cache and maybe the bus interface -- we don't want a situation where any subsystem is running faster (or slower) than is appropriate for the rest of the subsystem.
Also, slightly later in 2010, we have https://patents.google.com/patent/US8806232B2. Now we are

giving the power manager a little more intelligence (not CPU-level intelligence, think more finite state machine-level intelligence) along with some non-volatile storage. The power manager can now save (and then restore) the state of some devices using that non-volatile storage, and can engage in more sophisticated power control without having to fall back on the CPU.

Next we get (2011) https://patents.google.com/patent/US20120185703A1. The patent is not at all clear(!) but I think the new feature here is that performance control has moved beyond on/sleeping/powered down to DVFS, and so the power manager needs to be extended to know a co-ordinated set of voltages to apply to each performance domain to move all its sub-components up through a set of frequency regimes.

(2013) https://patents.google.com/patent/US20140208135A1 seems like another patent based on working around sad third party hardware. The idea is that you may put an agent to sleep, but some other agent may (inappropriately) generate the wake code for the agent. How to fix this? Well, the fabric sits between every agent and every other agent, so you turn the fabric into a censor, blocking any inappropriate wake-style requests until the central power manager confirms that, in fact, this device is allowed to wake up.

But that's old school! By mid-2013 we've moved from the old era of Apple having to integrate third party HW to the new era of an Apple SoC. With this comes a much more designed power approach exemplified by https://patents.google.com/patent/US10303238B2. At a high level we see here both a PMU and a PMGR. As I understand these, the Power Management Unit
supplies actual power (ie it is what ensures that x amps flows into the SoC at a given time, that this rises or falls as demand changes, but never exceeds danger limits) while the Power Manager implements all the power saving stuff we've previously discussed like twiddling registers and changing voltages while ensuring everything happens in the correct order and remains in spec.
More significantly we see the existence of a CPU Complex, and an addition to this complex of two important elements, the APSC (automatic power state controller) and the DPE (digital power estimate). In a way these are not primarily about saving energy (unlike the PMU and stuff we have previously discussed), they are about ensuring that the maximum power draw never exceeds what the battery can supply. The APSC controls things like how how many CPUs can be powered up at a particular voltage/frequency setting, while the DPE tries to track instantaneous power usage and ensure it's always within bounds, if necessary reducing the issue/execution rate of instructions for a few cycles. The APSC is programmed so that "normal" code running on all cores will not exceed limits, but in theory power virus code could exceed limits, and the DPE is there to catch when limits are exceeded and signal to the offending CPU(s) within a cycle or two that they need to throttle instruction issue or execution. (I already mentioned the patent for the CPU tracking the count of "heavyweight" instructions and throttling those if they are too dense. That's in 2011, but note that it's internal to the CPU. These mechanisms I'm describing move the management to the core complex and to the entire SoC, so they can allow eg one core execute vector code at full speed if the other cores are powered down, or are running undemanding integer only code.)

An interesting side note is that when a new core is powered up (a process that takes up to 10 μseconds), there's a concern that even though the new voltage/frequency settings will "eventually" be safe, there might be overload spikes during the transition, and so the effective clocks to the CPUs are halved (easy to do, as opposed to the more complicated manipulations of a general frequency shift). All in all it looks like they tried to think of everything -- and mostly got it correct except for the one small detail of forgetting that the maximum power a battery can source will eventually drift below spec...

The companion patent https://patents.google.com/patent/US9195291B2 describe how the power estimator works. It receives ongoing counts from each CPU of "significant" events, eg how many vector instructions executed per cycle, aggregates these to a per-CPU instantaneous power estimate, and sends per-CPU throttle requests as necessary. The number of "power events" over some period of time of time is tracked, and if this is too high the fact is reported to the PMU and PMGR, and the frequency/-voltage settings for the offending core(s) are reduced; and similarly if a core has been operating within spec for some period of time, its frequency/voltage is allowed to rise. I assume this, essentially, is what Apple has meant by saying they perform DVFS is hardware as opposed to having an OS driver make these modifications (obviously at a far slower rate, and with less detailed info).

This 2013 model of DPE was concerned with not exceeding the allowed power draw. But you can look at it from a different direction. Rather than worrying about a power virus, what about code that uses less power than we modeled, for example code that is purely integer with no use of FP or vectors? This is the subject of (2014) https://patents.google.com/patent/US9195291B2. We use the same digital power estimate but if the DPE detects a pattern of substantially lower estimated power than has been budgeted for, the PMU+PMGR are again informed, and the voltage for that core is allowed to drop slightly (while maintaining frequency), or frequency is allowed to rise while maintaining voltage.

Can we scavenge some voltage in any other way? Well a non-obvious fact about digital circuits is that there is a (not unrealistic) temperature+frequency range for which they run faster at higher temperatures (ie you can run them at the same frequency but slightly lower voltage if they are hotter).
(Don't confuse this with power issues! Running hotter is not something you especially want to aim for because it increases leakage current; but if the world around you has heated up your chip, can you make use of this fact?)
(2012) https://patents.google.com/patent/US8766704B2 does the same sort of thing as the previous patent; it detects if the SoC is at such a higher temperature, and if so, slightly reduces the voltage applied to the things like the CPU while maintaining frequency.
This is an extension of the idea in (2009) https://patents.google.com/patent/US8169764B2, which talks generically about more precise temperature compensation for the f vs V curve rather than just assuming a single curve at all temperatures.
If this sort of stuff excites you, there is more of it covering things like how to calibrate thermal sensors, how to run the control loops that ensure the SoC never overheats, how temperature affects radio frequency components, how these power and temperature measurements are communicated to the

outside world while debugging/optimizing the device, how to extrapolate a few local sensor readings across the rest of the SoC, etc etc.
Some patents seem like refinements of an older idea (like a way to use smaller voltage guard bands) that should be irrelevant given these more sophisticated new ideas like measuring circuit behavior, or power modeling. I *think* what much of this boils down to is that there's a lot of IP on a SoC, and it's not all amenable to the sophistication and refinement of the CPU power reduction schemes. For something cruder like a memory PHY you may be stuck with older techniques like guard bands, either because the problem does not allow for a better option, or because there are enough higher priority issues with this particular IP block to postpone anything but refinement of the existing technique.

The culmination of this is (2019) https://patents.google.com/patent/US10948957B1. The earlier patent estimated power by multiplying event counts by a fixed weight. Now
- the weights can vary, and
- the exact energy (over some time range) is measured and compared with the estimate based on these weights.
The weights are then updated the bring the two in sync.
Obviously this allows for a more accurate estimate (not least because, again, the energy used by parts of the SoC will vary with the temperature of the SoC, and with its age). The patent also points out that, apart from using the estimator to limit instantaneous power (given battery constraints), it is also used to limit total energy released over some amount of time, to maintain thermal limits.
This has a companion patent https://patents.google.com/patent/US10969858B2 which is obviously (though it never says as much) about AMX. With AMX everything we've said about power draw becomes that much more of an issue because a single AMX operation can involve so many simultaneous floating point executions. The patent describes an even more sophisticated power estimator for each cycle of the AMX engine taking into account
- (at least approximately) how many operations will be performed, based on the sizes of the input vectors/matrices
- a model of the AMX pipeline
- not just absolute current levels, but also the inductive power resulting from rapid *changes* in the current sourced by the AMX engine.

So let's consolidate.
- We have the OS deciding on a generally appropriate performance level for each core based on what it knows (like number of processes that want to run and their associated QoS).
- The OS tells the PMU which tells the PMGR to put each core in the appropriate DVFS state.
- The APSC and DPE estimate from the overall activity level of a core, and its temperature, whether the voltage level can be shifted slightly down (to save power) or needs to move slightly up (to service many expensive instructions)
- The goal, at a per-core level, is to keep the voltage right on the edge of acceptable, to save reduce but ensure that power events (when execution of vector instructions has to be halted for a cycle or two) are at the correct level -- not zero, but not too frequent

OK that sounds great and we have high-level control from the OS, and low-level per-cycle/per-core control. What's missing is something in the middle. In particular we don't want a situation where all four cores simultaneously either want to run vector code, or decide that they're doing too much vector work and want to stop; both will generate large current swings and noise. We won't get this from the above per-CPU DPE control mechanism.
The solution for this is (2016) https://patents.google.com/patent/US9798375B1. One part of the solution is to give each processor a pool of "energy credits", which are used up as the CPU engages in more heavyweight work. In principle if you stagger the times at which each core gets given its credits, and each core runs till it has used up all its credits, you'd do something to offset the times at which multiple cores all want to run all their vector pipelines simultaneously. But in fact Apple do something more sophisticated than that, making it probabilistic, as your credits run low, whether an instruction is allowed to execute in this cycle or not. A random or pseudo-random value is compared to the credits available, and if smaller, execution can proceed. This will both throttle the cores that have been using excess credits as their credits deplete, and will throttle them at different times, and with different timing patterns, thereby avoiding large current spikes.
This is followed up a few months later with (2016) https://patents.google.com/patent/US10452117B1 which extends the above idea to allow either core complex (P or E) to transfer any unused credits over to the other core complex, so basically allowing either the E- or P- processors to run a little more aggressively if the other complex is not being very aggressive (or, to put it differently, sharing the credits across all processors, rather than maintaining two independent pools of energy credits).

Cluster scheduling

There remains the issue of how the OS decides on "appropriate performance levels". This (2016) https://patents.google.com/patent/US20170357302A1 is a basic explainer, with (2017) https://patents.google.com/patent/US10884811B2 giving much more detail. (This now very much straddles like line between OS/API functionality, but helped out by a lot of SoC/CPU hardware.)
It's hard to convey just how glorious this second patent is! The ideas are (once you finally understand them) pretty impressive, but the whole thing looks like a briefing from the Pentagon, complete with thousands of acronyms and impossibly complicated diagrams like

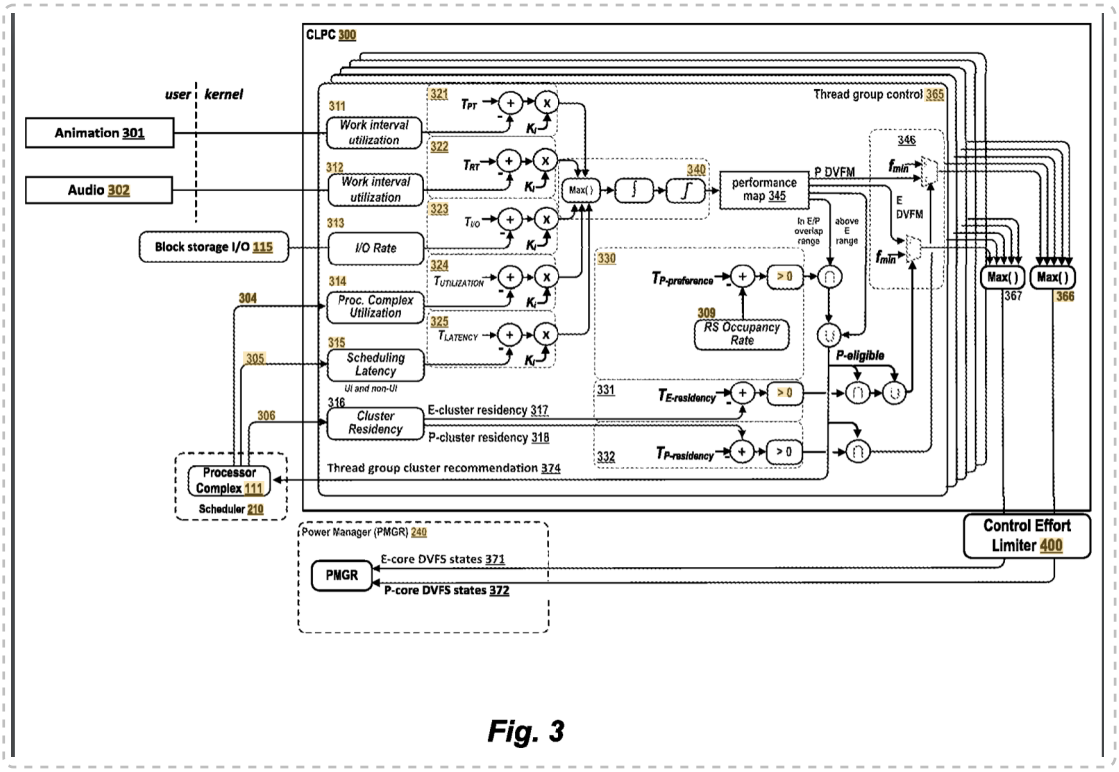


Fig. 3

Before even trying to understand this patent, understand the problem to be solved.
OS scheduling has always consisted of trying to balance off various desiderata.

On traditional (but fairly recent) desktop systems, the goal is to maximize responsivity while keeping throughput reasonable. This tends to be done by heuristics as to what are "user interacting" apps, and giving those priority boosts. There was a time when VM concerns were an essential part of this, and much of the scheduler's prime responsibility was to prevent thrashing (ie running simultaneously more than one app that was using a large amount of DRAM); presumably this is still a background concern but of much less immediate import.

Along a different dimension we have server scheduling. In this case throughput may be considered highest priority, and the most important trick is trying to pack together on different cores applications that each stress a different part of the overall machine, hence a few high memory bandwidth processes together with a few low memory bandwidth processes, perhaps a high FP/SIMD process on one hyper-thread together with an integer-only process on the second hyperthread, and so on.

Both of these versions of the problem are trying to select from a set of runnable processes, with a set of (somewhat known, depending on exactly what the SoC and the programmer provide) characteristics onto a set of cores, to optimize for something like performance. As a second tier goal, the OS will futz around with DVFS, but overall without much intelligence beyond a few very basic points like "don't exceed a certain temperature or power draw" or "if all the processes are known [somehow...] to be background, then reduce DVFS. Those readers who follow Linux will know of the never-ending travails

of Linux and Android scheduling in the face of DVFS, a matter that's still not really satisfactory after all these years.

Before the truly complex solution, let's consider the easier 2016 patent which is appropriate to A7..A10 class hardware. Neither clusters nor *visible* E vs P cores are part of this hardware.
Apple's goals are to hit performance targets (eg smooth UI) at minimal energy expenditure. The tools available are
- how many cores to power up
- DVFS of the cores
- less obvious, but also relevant, modifying the frequency of the SoC fabric and the DRAM

The basic flow of control is to have a first stage of scheduling that establishes performance goals. Think abstractly of "achieve state X by time Y" where X is some approximately knowable fraction of the total amount of work we want to achiever by a deadline.
How do know how fast to run the machine to achieve those goals? This is the role of the CLPC (closed loop performance controller) which essentially means you keep measuring your progress and, if you are going faster than necessary to meet the goal, slow the machine down; otherwise speed it up.
This, in turn, means that, to a first approximation, we describe performance as a monotonic map from *n* cores running at maximum frequency down to 1 core running at minimum frequency. Depending on how well we're meeting the target, we continually move up or down this performance map.
[Of course there are two extreme cases of "as fast as possible" which is the default for third party code where nothing special is indicated, like, eg Geekbench code; and "as low as energy as possible" which will be the case for almost all cases where the screen is dark and only background code is running. However much executing code is media decode, animation, audio, even much network and IO, which fit this deadline model.)

But that just gives performance, without energy concerns. So the output of this stage passes through a second level of control which worries about energy and performance.
One part of this concern is easy. Things like device skin temperature, temperature at various extreme parts of the SoC, and short-term power draw are tracked and high performance levels which might cause these to be exceeded are dialed back. So essentially step A says "I'd like to operate at this performance level" and step B says "given current circumstances that runs too hot/draws too much current; dial it back to this maximum performance level allowed by thermals, battery, low-battery mode, etc".

Beyond that upper level constraint, the efficiency controller is tracking an energy per instruction metric and trying to optimize this while not impairing the performance goals.
An especially interesting part of this is that, although there is an override mechanism (which forces the energy/instruction cost to 0 as long as it is active) most of the time the controller is trying to maximize energy efficiency. This might sound bad, like it will hurt performance, but mainly what it's saying is
- if you're constantly waiting on DRAM, then running the core at high frequency does you no good anyway

- if you're not running very wide (hard to predict branches, or long dependency chains) you can't take advantage of the big core anyway, so why waste power keeping you there rather than on an E-core?

To make all this work well
- one needs good metrics, and a lot of the patent is about how the energy is measured (over "long" time intervals) and estimated (over "short" time intervals) to inform these decisions
- one also needs a good control framework. Anyone who has ever tried to write control loop software knows that it's very easy to have the system oscillate wildly; or alternatively, if you try to prevent that, to respond to changes so slowly as to be useless. So another large part of the patent is about that control framework.

- not stated, but an obvious and easy extension would be metrics that are somewhat orthogonal to these primary metrics. For example performance might be lower than we wish because either the CPU is too slow or DRAM is too slow.
This makes our control problem two dimensional, but the basics remain the same – change one or the other of the DVFS knob and the DRAM frequency knob, see how performance and/or energy efficiency changes, this allows us to calculate scaling coefficients (ie derivatives) telling us the response to both of these knobs, and we can use those scaling coefficients to calculate an optimal point (which will of course keep changing as the code keeps changing).

This is an adequate first start but we need something much more sophisticated for the A12 and later.We have the same concern of course, for responsivity and low energy usage, but we now have the complications of
- there are both P and E cores
- these are grouped into clusters and clusters run at a fixed frequency. So we can, eg, use three cores of a 4-core cluster, but we can't run two of the cores as fast as possible and two as slow as possible

To do this as well as possible Apple introduce a few new concepts.

• One is the *Thread Group*, a group of threads that are scheduled together, so in a sense the cluster is given a unit of work rather than each core being given a unit of work. Thread Groups are dynamic. They are constructed at the start of execution from various data (heuristics, indications by the programmer, ...) but threads can constantly move between Thread Groups if this makes sense.
Various OS technologies allow the OS to see when two threads (perhaps from the same process, perhaps from different processes) are working together, and these threads will be united in a Thread Group to schedule together as much as possible. (One thing I never saw an answer to is what happens when one member of, say, a four element thread group, is required to yield, eg while waiting for IO. Perhaps the answer is that along with the "large" thread groups there also threads with no obvious affinity to any other thread, ie singleton thread groups, and these are opportunistically executed whenever such an occasion arises?)

• Another concept is the *Work Interval Object*, a way to describe a task (perhaps worked upon by multiple threads) which has a known deadline and a known progress towards that deadline. For tasks which match this model, (think eg of audio playback or UI animation) the task (ie the set of threads) can let the OS know of its performance as it progresses, and the OS can see that the threads either need more performance, or can relax a little, to meet the deadline.

Neither of these are meant to describe all code under all circumstances. But they do cover a lot of types of code, and they can be used by the OS to perform substantially better scheduling (ie fast enough, while saving a lot of energy).

Bearing that in mind, lets look in more detail. From the parts I can understand, I think the essential idea is
- the OS constructs an object (let's call it a *task*) for every thread that wants to run, along with an associated performance goal
- a performance goal is anything that ultimately turns into some sort of measurable rate (so many *x* per second). Certain tasks that are closely tied to the OS can provide precise versions of this performance goal (for example IO may have a goal of so many blocks/second, threads handling animation will have a goal of 60 frames/second). Deadline goals (ensure this frame is decoded by time T) can be converted into a rate goal as long as there is some reasonable proxy for what fraction of the frame has been decoded so far. A task can have more than one goal, so both a "throughput" type goal and a latency type goal.
- as code becomes more and more "normal" code, it's probably going to provide less and less of this info, with the most non-detailed levels being tasks that are simply associated with a QoS, and I *think* the QoS is associated with an amount of energy per instruction as the performance goal; so at one extreme we have "infinite nJ/ instruction" for maximum performance, at the other extreme we have "nJ/instruction as low as you can get" for background work, and in between we have specific "nJ/instruction" levels.

There are also a few non-obvious goals whose cleverness only becomes apparent after some thought. For example one of the goals is (essentially) time spent runnable but not running. Who cares? Well, all the goals mentioned earlier act to optimize a single task, but if we have more tasks than resources, we don't just want each task running at optimal CPU speed, we want the CPU speed to be higher, so that the core can do the work of more than one task over an interval. This runnable-but-not-running metric covers this aspect of sharing between tasks.

- the scheduling machinery ultimately wants to achieve that each task gets as close to its performance goal as possible, while
+ never exceeding power limits
+ never exceeding thermal limits (both of these are hard constraints)

The "main OS" provides the tasks, the hardware provides a whole lot of sensors, and a low-level subsys-

tem (which may be part of the OS, or firmware, or hardwired, or a microCPU on the SoC) tries to balance all this. The primary way of doing this is via Closed Loop Engineering which is a fancy way of saying
- continually compare the vector of goals to a vector of results (eg actual blocks/second vs desired blocks/second, actual energy/instruction vs desired energy/instruction)
- depending on how closely they match, move a "control effort" up or down.
The control effort is a scalar indicator that is supposed to indicate (as best possible) how much "performance" to throw at the task. It varies between 0. and 1. Each value of control effort is mapped to a table of "performance effort" which we can think of, for now, as essentially starting at 1.0 with a P-core running at maximum frequency, and moving downwards till we get to at 0.0 an E-core running at 400MHz or whatever.
The control effort is mostly discovered by the closed loop process I described above. The thread starts at a control effort of 0 (so minimum E-core speed). Next time round, if the desired metrics are too far from where they should be, the control effort is bumped up a little; and so we go, until hopefully at some point things stabilize.

So this is the basic idea as of 2016, but there are extra complications which are the subject of 2017. Some of these include
- in many ways the object that executes code is the core complex, not the individual core. I *think* each core complex can only run at a single frequency.
At first blush this seems non-ideal -- you can, it is true, power down individual cores, but you can't run one P core fast and the other one slow. I suspect this is because of the tight coupling of all the cores to the L2. To access the L2 rapidly, it needs to be synchronous with each core, which means all the cores (and L2) are all running at the same speed…
We have seen various patents that allow for slightly shifting the voltage of a core while at a given frequency. If the cores run on separate power planes, this remains feasible, and may work well enough that Apple doesn't consider it too much of a problem requiring this equal frequency across a cluster? I can't tell from the patents I've seen so far if all the fancy stuff Apple patented about voltage adjustment are in fact per core, or complex-wide.

The basic structure, now, is
- we create thread groups
- every so often we may move threads between thread groups depending on the heuristics described above (eg one thread is in a producer/consumer relationship with another)
- thread groups, based on the various metric and how well those metrics match our desired performance goals, will result in a *control effort*, a number between 0 and 1, which is mapped to a cluster (P vs E) and a DVFS level for that cluster
- this (optimal) control effort is looked at by various concerns (skin temperature, die temperatures, short term and long term current levels, user-performance settings [low battery mode] etc) and reduced to the minimum level that each demands
- the thread groups then sit in a queue (E queue and P queue) and, when they get their chance, are run

at the suggested settings.

Note a few consequences of this:
- suppose we have many desired low performance tasks but few to no high performance tasks. Eventually some of the low performance tasks will be far enough behind their desired metrics that the control effort will map them onto P cores. So no special effort needs to be made to wake up P cores when only the E queue is populated.
- what about the reverse situation where there are excess tasks in the P queue? Should they run on the E-core? The Apple answer is yes, but... Running them on an E-core is not an immediate win because there is some overhead involved in switching E-cores on, and in the transfer of state from the P-complex (data in the caches) to the E-complex. Apple handles this via a special interrupt with an associated timer – the interrupt is set to wait a certain amount of time before waking up an E-thread and having it begin executing the P-thread.
There is never an effort to move in the reverse direction; if E-threads will not move to vacant P-cores until the natural control effort mechanism drives them to demand P-level performance.

The one thing that is not explained is how this scheduling in terms of thread groups maps onto individual threads and cores.
Obviously a few cases are easy, eg if a thread group consists of four P-threads and nothing else is running, presumably on an !4 two threads will execute this scheduling quantum and two the next quantum, both at the same DVFS settings.
But what about imbalance situations?
This is not described (the patent cares about the concepts I have been describing) but I *think* the way it works is
- we try to schedule thread groups together, but
- if a core or two becomes free and there are runnable threads in the queue, they will be scheduled into a core and
- the DVFS setting (ie the control effort) will be at the highest level of all the thread groups that are running on that core.
A way to think of this is
- it's not that slow code is wasting energy by running at a higher performance level than necessary;
- rather it's that we are running fast code at its appropriate speed, and if there's some slow code that needs to execute, well, might well take advantage of the fact that the cluster is awake, the L2 is awake, so get it done even if it's executed faster than really necessary

- the actual object Apple wants to deal with for most scheduling is something called the *thread group*. The idea seems to be, approximately, that we want to schedule what looks to the user like an "application", rather than scheduling threads independently. But we don't just consider the threads created

by the application because a lot of work (animation, audio, decode, IO, ...) is done by OS threads on behalf of an app, so we dynamically move threads around into these scheduling constructs, as eg the audio thread does work on behalf of some app.
- each thread in one of these thread groups has its sets of performance goals, and these are all compared with measurements to generate a final control effort for the thread group as a whole
- this effort is again looked up in a table, but now the table is not just a list of single processor states, at maximum effort it might have 2 P cores at maximum frequency, down to 1 P-core at low-frequency or 4 E-cores at high frequency, down to 1 E-core at lowest frequency.
At various stages these desired performance levels are over-ridden either by the instantaneous power manager (which believes the battery, in its current state, cannot source the current required) or the thermal manager (which believes that if this performance is allowed the device will get too hot).

Along with all the above, also continually measured for each thread group are E- and P-complex residency (which is unsurprising, one can imagine ways this might be used).
More interesting is that also measured (presumably using counters deep in each CPU) is RS-residency (reservation station residency) which measures on average how long each instruction waits before being executed. Ultimately this is a measure of how much the code is actually using the capabilities of a large core -- if it is mostly waiting on DRAM, or has long runs of serial code that cannot exploit the width of large core, then, regardless of all its other goals, there's probably no value in running it either at high frequency (waiting for DRAM) or on a wide core (serial dependencies).
There are also implications that the system is constantly measuring the types of instruction used, so that it knows if a lot of vector work is being done by a particular thread.
These measurements based on CPU counters appear to be attached to threads as extra attributes, and the main way they are used is to decide between "overlap" cases -- if the performance range is a good match to either a fast E-core, or a slow P-core, which should we choose?

So we have, for a thread group, a desired number of cores of a desired type at a desired frequency, and the threads are all marked as such.
They then go into either an E- queue or P-queue and are appropriately pulled out of this queue for execution.

Now if you have read all this so far, some things might seem problematic! In particular the system seems to really want to schedule a thread group as only P- or only E-. This would seem to have two problems.
The first is "what if I have, say, lots of hard-working threads, more than I have P-cores?". This is dealt with. The idea is to hold off on using E-cores for a while, but if this state of lots of P-threads persists, the system will give up and start moving threads placed in the P-queue across to the E-queue (presumably under these circumstance the understanding is "we are now trying for maximum performance" so issues of frequency no longer occur -- we have already dialed the P's to the maximum, and we'll run the E's as hot power and thermals allow).
I think it's important to recall the problem Apple is trying to solve! In a sense the "I have a large compu-

tation that uses lots of cores and goes on for many minutes" problem is very easy, with the obvious solution running every core as fast as allowed.
But that's not the hard problem; the hard problem is the one of "I have a constant stream of small, short-lived tasks; some of them have no performance concerns; some of them demand smooth animation; and while executing them all (on up to 6 cores) I want to use minimal energy". So the system is biased towards starting everything at the low end of "try running all the jobs, one after the other, on a single slow E-core, and heck, maybe that's good enough?" while climbing up from that minimum rapidly if circumstances indicate it's necessary.

The second issue is "what if my problem naturally decomposes into a few performance threads and a few supporting slow threads"? The answer seems to be that for almost all practical cases (where the slow threads are various Apple support threads) the OS knows how to handle the appropriately; and if you insist on writing a very strangely unbalanced app, there are APIs you can call to modify thread grouping.

I don't know how this works out, but there doesn't seem to be an answer. The answer may be at the level of the thread groups -- the thread groups are supposed to be constructed not just of threads working together, but of threads that have similar performance demands, and any threads that seem to be drifting from the rest of the app maybe get moved to a different thread group after time? Presumably, like all this stuff, if the issues only exists for a second or so, who really cares? And if the issue persists for a while, some sort of high-level movement of threads to a better-matching thread group seems good enough.

On the other hand this scheme also has advantages. For example it naturally groups together (running at the same time, on the same core complex) threads that may be interacting (producing data for each other, sharing data, using the same locks), and such threads will have rapid communication through their shared L2.

So what we get from all this is a collection of threads, tagged by core and frequency, that are placed in E and P- runnable queue. Finally at each scheduling operation we try to pull out from these queues something that matches all the work we have done.
Once again, the *common* case, the one we are optimizing for, is that most cores are idle most of the time, and we're not in fact trying to pack every core as tightly as possible with work!
If we have four independent light-weight tasks lined up, the preferred scheduling may very well be to run them all sequentially on a slow-E-core, rather than fire up four E-cores, or run that E-core any faster. Obviously there are escape mechanisms whereby the system is tracking if the runnable queues are growing too large, at which point the obvious types of things will kick in, but the interesting stuff Apple is doing is primarily about keeping the system "feeling" fast while in fact most of it runs as slow as possible, and it ramps up to running more cores faster as effectively as possible. (Effectively mean-

ing:
- know how fast you want to be, and compare that to what you're achieving. Measure, measure, measure!
- try to keep threads that are working together scheduled together)

Following this particular thread goes into a whole other world of OS technology, scheduling, transferring priority between threads, etc. Far far from our starting point of CPUs, so we'll leave it here!

Naturally all this power stuff has its counterparts on some other IP blocks, most obviously the GPU. For example an early version is (2011) https://patents.google.com/patent/US8856566B1.
This matches the thermal bounds tracking aspects of the CPU scheduling described above, but another way to think of it is that it's the equivalent of Intel's turbo'ing based on thermal inertia. The tech details differ, but the goal is essentially the same -- keep track of when the device is not running extra hot, and allow that to accumulate "credit" so that you can run the device hotter than optimal for a brief period, until the credit is used up (ie the thermal mass has absorbed all the heat generated, and continuing will raise temperatures too high).

- move what knobs you have available (primarily whether any thread goes on E core or P core, and the core frequency) to bring these two into better alignment
- while ensuring (obvious) that you don't go out of bounds and (less obvious) that you don't create state oscillations that grow larger and larger (eg run a CPU too slow, then panic and run it too fast, then panic more and run it even slower, ...)

This all sounds plausible enough as a rough start, but obviously many details are omitted. The part they seem to care about the most for the legal purposes of the patent is the use of control theory, which is not exactly the part we (as investigators of the system) are most interested in!
Things are improved (in the sense that we get more detail -- too much!) in
which of all patents I have read looks the most like a Pentagon briefing! God help you understanding this, but the new ideas that are introduced include
- the OS cares more about *thread groups* than individual threads. Thread groups appear to be something like a set of threads all working together to achieve the goals of a single app; but they are somewhat dynamic so that (I think this is how it works) OS threads like an animation thread or a media playback thread will join the group of an app while they act on behalf of that app

- these thread groups are treated as primary scheduling units (still using the previous ideas of a performance goal or two attached to each thread, ie task).
I think the idea here is to back away from the traditional OS model which schedules threads as the basic units, with a byproduct that an aggressively threaded app gets a lot more resources than a non-threaded app. All this dynamic thread group business seems to be creating a single object representing

"all resources required by one app at this point in its execution", to be scheduled as such, and balanced as such against other thread groups (ie other apps).

- based on thread groups and matching target metrics to measured metrics, a map of the best performance schedule for this thread group is created (ie each thread gets mapped onto a P vs E core at this frequency)

- these maps are then read by the *thread* scheduler (as opposed to this earlier app/system scheduler) which puts each thread into an E vs P queue, with the appropriate DVFS info attached to it. At this stage there may be some slight juggling of the queue across different thread groups to ensure maximal occupancy of every core.

Along with all the above, also continually measured for each thread group are E- and P-complex residency (which is unsurprising, one can imagine ways this might be used). More interesting is that also measured (presumably using counters deep in each CPU) is RS-residency (reservation station residency) which measures on average how long each instruction waits before being executed. Ultimately this is a measure of how much the code is actually using the capabilities of a large core -- if it is mostly waiting on DRAM, or has long runs of serial code that cannot exploit the width of large core, then, regardless of all its other goals, there's probably no value in running it either at high frequency (waiting for DRAM) or on a wide core (serial dependencies).

One thing to bear in mind as you read through these two patents is that I *think* each core complex can only run at a single frequency. At first blush this seems non-ideal -- you can, it is true, power down individual cores, but you can't run one P core fast and the other one slow. I suspect this is because of the tight coupling of all the cores to the L2. To access the L2 rapidly, it needs to be synchronous with each core, which means all the cores (and L2) are all running at the same speed.
We have seen various patents that allow for slightly shifting the voltage of a core while at a given frequency. If the cores run on separate power planes, this remains feasible, and may work well enough that Apple doesn't consider it too much of a problem requiring this equal frequency across a cluster?

A third issue that is not address is scheduling "closely interacting" threads that will be engaged in a lot of communication and lock-sharing. You'd want such threads to be scheduled on the same core complex (both E, both P, same of each when there Apple has larger SoCs with multiple P complexes) so that they can communicate rapidly through the L2. The mechanism as currently described does not seem to incorporate anything linking two (or more) threads together as "ideally schedule on the same core complex".

This all sounds good (once you see the big picture) and it certainly seems to work well for all the current use cases! iPhones run at low power, M1 macBooks feel very responsive, etc.
One thing that does seem to be missing is any sort of concern for

- programmer modification of thread topology. The Linux crowd go on about this a lot, the Windows a little. Apple seems uninterested in catering to them.
- affinity (ie retaining a thread on the same core across scheduling to take advantage of the prewarmed cache and branch predictors.) Apple's scheduler for intel seemed to positively fight affinity, constantly moving a thread, and I think this was to take advantage of thermal inertia when there were one or two high priority threads on a many-core system. This seems unimportant for Apple's cores. Perhaps they use affinity at the final stage of scheduling, but don't consider worth putting in the patent?
- (most important) scheduling nearby threads that will be engaged in a lot of communication and lock-sharing. You'd want such threads to be scheduled on the same core complex (both E, both P, same of each when there Apple has larger SoCs with multiple P complexes) so that they can communicate rapidly through the L2. The mechanism as currently described does not seem to incorporate anything linking two (or more) threads together as "ideally schedule on the same core complex".

- every executable object that can be scheduled (call it a thread) has an associated QoS
- each QoS has an associated performance price it is willing to pay (so many nJ/instruction)
- simultaneously the system is constantly measuring the instruction count and energy usage of each CPU so that the OS (or perhaps the PMGR) has an approximate idea of what config (P vs E core, at what frequency) will cost a given nJ/instruction; and these two are matched up.
But most of the patent is written in control theory language (which I don't speak), wwith system A modifying the choices of system B, and in turn being modified by system C, and I don't know what problem this complexity is supposed to be solving. I think maybe it's supposed to provide hysteresis so that each core isn't madly jumping from one state to another, but only moves after a few rounds of scheduling establish that it really isn't the best fit for the task?

Where does the coupling between frequency and voltage come from? Obviously these could be set at production time, but the settings would have to be sub-optimal, to handle both variations across chips

and variation as the chip ages. Instead Apple has (2009) https://patents.google.com/patent/US7915910B2 which occasionally performs a self-test (slowly reduce the voltage while keeping a particular frequency, until the test returns incorrect results) and stores those voltage/frequency pairs. This is an extension of the earlier (2005), also PA Semi, https://patents.google.com/patent/US7276925B2. The earlier patent seems to test that a few types of circuit elements work as voltage changes, whereas the later patent seems to test that an entire digital subsystem (cache, adder, whatever) works.
The patent doesn't say as much, but I assume a similar idea was used, for example, to establish the minimum voltage at which cache banks can sleep without losing data. Ultimately for cache banks we move to this fairly amazing patent (2016) https://patents.google.com/patent/US9922699B1. The idea is that each cache bank is powered through one of a set of (say eight or more) power diodes, each of different size and hence voltage drop. Based on temperature conditions, the system decides which diode to use for a given bank, thereby using the minimum voltage possible under current conditions.

Apart from the above ideas, a different power-saving strand is to use dedicated hardware as much as possible. This begins with (2008) https://patents.google.com/patent/US8359410B2 which talks about using a dedicated audio DSP and codec to handle either music playback or improve the audio quality of phone calls. Again this seems a no-brainer, but Apple say that the state of the art at the time is to perform this sort of work on the CPU where, of course, it consumes more energy.

Instruction Fetch

branch
https : // patents . google . com/patent/US9626185B2 - diagram of branch pipeline and training stage
https://patents.google.com/patent/US20150039860A1 proof that Apple are using checkpoints
https://patents.google.com/patent/US20150293577A1 a few loop buffer details

Now let’s consider Fetch and everything related to that. (More so than other sections, as I wrote this I kept finding that a concept I was explaining now relied on a future concept. I’ve done the best I can to order the material, but you may find it worth reading this once fairly rapidly, just to get the basic ideas, then again to see how the ideas all fit together.

Like so much, most people have in their minds a vague idea of Fetch that was maybe appropriate for the late 1990s but has little modern relevance. The way to think of Fetch is, like most modern micro-architecture, to start by thinking of the problem in abstract terms.

Program execution consists of a sequence of instructions, mostly short sequential runs of instructions (say five to ten instructions) followed by a new sequential run. The gaps between these sequential runs are, of course, branches of one form or another.
Now the *important* point is: a natural division of work arises. From Decode onward, the machine has no interest in the gaps between these runs of instructions. As long as we have branch prediction, the interior of the machine just sees a stream of instructions to execute. It's the job of Fetch to construct that stream as accurately as possible; it's the job of mispredict recovery to pick up the mess when something goes wrong; and it's the job of the interior machine to ignore both those two other jobs.
If you take this seriously, you should realize that we have exactly the same situation as much of the rest of the machine: we have two pieces that can operate mostly independently, and we should put a (possibly large) queue between them so that when one is blocked the other is not. Decode just wants to extract a steady 8 instructions/cycle from the queue. Fetch should try to deposit into that queue as much as possible per cycle. Some cycles Fetch puts nothing in the queue (I-cache miss), some cycles just a few instructions (the sequential run being fetched is not very long). To make up for this, when long sequential runs are encountered, Fetch should be capable of fully exploiting them, moving sixteen instructions or more from the I-cache to the Instruction Queue.

Next step in understanding. While Decode is going to operate as a blind engine that just grabs as much as it can (up to 8 instructions) per cycle from one end of the Instruction Queue, Fetch on the other end is going to operate as an asynchronous engine. This means that (in the absence of any indication of things going wrong) *every cycle* Fetch predicts
- where to load the next sequential run from
- how many instructions to load from that run
People talk of Branch Prediction (and we will get to that) but the more immediate concept you want to understand is Fetch Prediction, which refers to the two points above.

You may think this design is just obvious an common sense. And it's not new, it was proposed by Glenn Reinman in (2001) https://cseweb.ucsd.edu/~calder/papers/UCSD-CS2001-676.pdf *Hardware Optimizations Enabled by a Decoupled Fetch Architecture*, but you'd be amazed at how small the fraction is of people who get this and what it implies.
(The thesis is essentially two parts, the first describes Fetch, the second we'll cover later, and considers instruction Prefetch.)

Now let's consider further implications of this Decoupled Fetch design. Assuming prediction is correct, at any given time we have (distributed over multiple queues and pipeline stages) instructions that are in-order running from cache access stages through presence in the (deep) Instruction Queue through Decode, then Map and Rename. After rename the instructions become out of order.
Note a consequence of this -- if we realize that an earlier prediction was incorrect, we can correct it without too much pain a fair way down the pipeline. If we realize that instructions, even in the Instruction Queue, are incorrect, we can simply flush them and re-pack the Instruction Queue from the correct address. We lose a few cycle, but maybe not even those if we had enough queued instructions ahead in

the Instruction Queue.
If we catch an error in Decode by then we've started to allocate resources (in Decode we allocate, at least, ROB slots) so we can still correct an error without a total machine flush, but doing so is more work. And so further on through Map and Decode – if we want to correct a Fetch error at these points, instructions are still in order, and so doing so is technically feasible, but requires a lot of care to de-allocate all the resources that have been allocated.

So what, you say. Well what this means in practical terms is that we can utilize a number of different types of predictors, with different latencies. We need a basic Fetch predictor that can deliver a next fetch address within a cycle) but we can also have additional predictors that check the stream over the next few (perhaps two to even as high as five) cycles and can kill it, and reFetch, without too much pain and drama.

Here's the idea (from (2016) https://patents.google.com/patent/US10747539B1). Ignore the details, consider the big picture.
Pipe Stage 1 decides the next Fetch Address. This is derived from
- various single cycle predictors
- a multi-cycle predictor sending a correction signal (ie "flush the last two cycles worth of Fetch, and restart here")
- the Decode unit detecting an easy branch (direct, non-conditional)
- the eventual execution of the branch detecting a mispredict and forcing a flush
Ideally all but the last of these don't require much interaction with the bulk of the core, so they will cost neither too many wasted cycles nor too much energy.

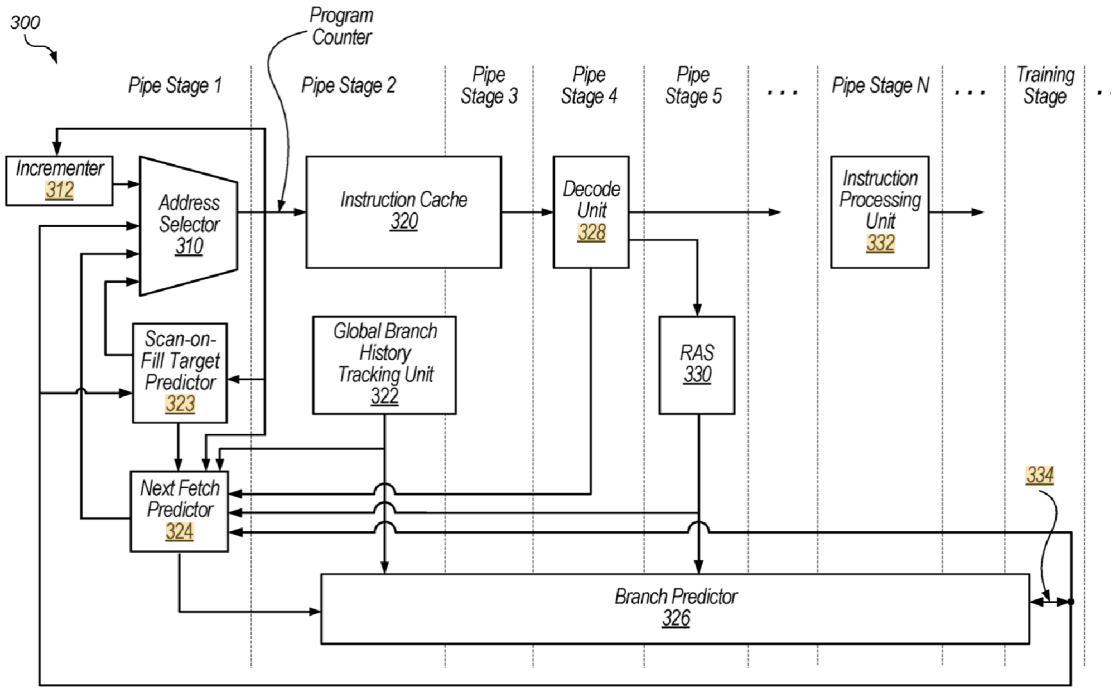


FIG. 3

More consequences.
One average we have a branch say every 6 or so instructions. There have been designs in the past that were essentially driven by branches (imagine something like
- predecode [scan instructions and mark those of interest as a cache line is moved into the I-cache] marks branches
- Fetch pulls in the I-cache line up till the next instruction marked as a branch
- the address of that marked instruction is fed into some sort of predictor which indicates the next Fetch address)
This is easy enough to implement but it means Fetch halts at every branch, even non-taken branches. And good luck running 8-wide if most of your fetches pull in only ~6 instructions...

So it's simple math that to do better we have to stop treating branches in the stream as special; what

matters as break-points in the stream; a branch that is predicted non-taken is of no interest to Fetch. This means that now, on average, maybe we're at ~10 instructions between taken branches. That's good, but a second issue is cache line boundaries. In the past it's usually not been worth reading two cache lines in a single cycle, rather you read till the end of a cache line, then pick up the rest of this sequential run next cycle. With 128B (32 instruction) cache-lines, that may still be feasible for Apple, the loss from runs that occasionally extend over a cache line may be small enough to ignore for now. But this once again reinforces the point that with all the things that can go wrong (I-cache misses, short instruction runs between non-sequential addresses, instruction runs that reach the end of a cache line) you really want Fetch to pull in as much as it can (much wider than the nominal width of the CPU) so that *on average* you maintain a flow of 8 instructions per cycle.
Apple is clearly getting close to the point where the sort of machinery I've described is no longer good enough. The next stage would be a predictor that spits out not one but two fetch predictions, the next sequential run, and the one after that, along with cache access machinery that can access two separate cache lines and move the results, appropriately ordered, into the Instruction Queue. A number of papers have talked about this, but I'm unaware of any actual implementation.

Now some general points about predictors, before we get specific.
In the early days of prediction,
- transistors were scarce enough that the same structure was used for both training and prediction. There may still be cases where this works well, but you'll notice with a lot of Apple predictors (not just branch prediction) that one structure is used for training, a different separate structure for generating predictions.

- people were somewhat fast and loose about the exact details of how predictors were updated. But prediction is now sufficiently accurate that you really don't want to pollute your predictor training with incorrect data. This has two consequences.
One is that you don't want irrelevant code (most obviously interrupts) being allowed to feed data into your predictors.
The second is that you don't want to train your predictors on speculative instruction streams.
This leads to a tension. On the one hand, you want every successive branch in the *speculative* stream to inform your prediction, because that is recent data that carries a lot of useful information about subsequent branches. But you don't want that same data locked forever in your long-term prediction data.
So the best predictors have a somewhat complicated structure (both storage and training) because they want to maintain one set of data that's informing predictions, but is provisional, along with a second set of long term data that's absolutely accurate, and they want to use both of these optimally.

- how do we handle context switching? Traditionally this has just been ignored, with an expectation that, sure, after you context switch things will such for a while until the predictor is retrained. Again, good enough for the old days, not acceptable if you want the best performance possible.
So how can one do better? I'll suggest three options I think are viable, and later we'll see what Apple does.

+ The (apparently) simple option is you just swap the branch predictor data every time there is a context switch. Add some appropriate instructions, get the OS to call them. Sure, it will work, but it suffers from the problem that there is no natural path from the branch predictor SRAMs to the rest of the machine. You have to add ways to extract, then replace this information and those paths have no value beyond context switching.
+ You could somehow tag every branch prediction entry with an ASID. When the predictor is referenced, we see if the ASID matches our current ASID. If so, we trust the predictor, if not we reset it to neutral. We would expect that in any given time slice, only a few branch prediction storage slots are used by a particular executable, so that, for the most part, at any given time the branch predictor holds relevant predictions (being used by the current app) plus predictions used by the last few apps, and if any of those apps are re-scheduled on this core, they'll be able to reuse those predictions. This should be familiar as the way many TLBs work (with an explicit ASID tag for each TLB entry) and it's also of course the way any physically addressed cache (ie pretty much all of them in machines of real interest) work, with the ASID being effectively embedded in the virtual-to-real translation.
+ Best of all, probably, would be have all branch prediction operate in physical rather than virtual space. Then you get the equivalent of every entry being tagged by ASID, but even better, you also get sharing. Most code executed on Apple devices is code in Apple shared libraries, and while there are surely counterexamples, one would expect that prediction trained on one execution of such a shared library is usually a good fit to a different execution of the same shared library. Sounds good, but it does require substantial infrastructure in the design because the code, as written, presents all addresses (whether relative PC offsets for branches, or subroutine call addresses, or indirect branches for virtual calls and procptrs) as virtual addresses. So if you want to fight that, you need various back-channels or such sneakiness to convert the virtual address stream present in the instructions to a physical address stream.

So, still operating at the abstract level, think about how you'd implement such machinery. When the machine boots up you know nothing, so you have to bootstrap from there. The obvious easy solution looks something like
- by default we just keep executing forward, pulling in as many instructions as we can from our current position
- the machine downstream at the point of execution of a branch will eventually detect errors (ie the instruction after a branch doesn't match the PC that the branch calculated as being the next PC)
- these will be reported to the Fetch Predictor, the Fetch Predictor will record in a large table with entries like "at PC x2, there is a jump to address x3'"
- the Fetch Predictor is constantly maintaining 3 PCs:
+ startOfRunPC
+ branchPC
+ branchTargetPC
- using these we can also fill in, for previous entries, how long the Fetch Group should be, and what PC to look up for the next cycle of Fetch
- so eventually this table will be reasonably densely populated, so that each cycle we can look in the

table for the address we plan to jump to

This gives the basic idea, now we need to fill in details.
First of all, in any given cycle what we try to look up in the predictor is
- the address of the next run of instructions, and the length of that run; both of which can be fed to the I-cache.
Suppose we don't have a hit in the Fetch Predictor? Then, in the absence of anything better, we keep going forward.
How many instructions to load for these unknown cases? One option is just load as many as you, the downstream will figure it out! But that may not be power optimal, if there's a good chance that you are stumbling blindly into unknown territory. Maybe a better option is to proceed one instruction at a time, slowly but cautiously, so that while you're in unknown territory you're wasting minimal energy as you build up a map? Or maybe simulations show the optimal length is 2 instructions of Fetch in this unknown case?

We want to correct errors as soon as possible.
One way to do that is at Decode (at which point you know where the branches are) you can handle various simple branches right away. For branches that are non-conditional (so you know they are taken) and for which the address is embedded in the instruction (eg branch to PC+offset, or the equivalent for branch and link [ie subroutine call]) Decode can check right away that the successor instruction matches the calculated target address and, if not, can flush everything after the branch and inform Fetch (including updating the Fetch predictor). This saves a few cycles compared to having to wait till the branch Executes or, even worse, Retires. We see that pattern in the Apple patents, in the Control Flow Evaluation block of Decode.
Even some more complicated cases could in principle be handled this way. Consider a branch-to-link-register or an indirect branch. Is it likely that these are directed to the instruction directly after the calling instruction? Of course not! So if the PC after a `BLR` (indirect call) or a `RET` is the current PC+4, that's likewise a strong indicator that we need to flush and train the Fetch Predictor for this PC. [should actually try to test for this case]

Another thing we can do is have certain especially difficult prediction cases (especially indirect call) offer a "halt" behavior. If we reach Decode and have reason to believe that a `BLR` is unlikely to be correctly predicted, we can just pause the Fetch pipeline until the `BLR` is executed and provides a value.
For simple taken/not-taken branches this is usually not worth doing because even if you guess the direction, there's a reasonable chance of being correct; but for indirect branches there are many ways to be wrong (wasting power) and only one way to be correct.

How much such a predictor actually be implemented? Essentially it's like a direct-mapped cache. Suppose we want 4096 entries. We'd take the 14 lowest bits of the PC (drop the 2 lowest bits which are always zero) and use those as index into a table. This is reasonably fast (it needs to operate faster than

a cycle!) and reasonably accurate. But note one consequence of this will be aliasing, ie if we have two PC's that match in the 14 lowest address bits, then they both can't live in the Fetch Predictor table at the same time.
We can reduce aliasing by using more PC address bits. 14 bits are within a single page; beyond 14 bits we start using pageNumber bits. Do we want these to be virtual page number or physical page number? Virtual is clearly easier, but physical is probably doable and may be better in a theoretical sense.
We could also reduce aliasing by using cache type technology. The bare minimum is attach a tag (say a few more higher bits from the PC) and compare those to the current PC. A match indicates a good Fetch Predictor entry, a mismatch means ignore the entry. This will prevent using bad entries, but won't allow two aliasing entries to co-exist. We could attach two entries to each slot (ie now we've created a 2-way set associative cache for the predictor), but that's probably more energy usage than it's worth, and it introduces tag comparison into the critical path. (The previous validation tag does not have to be on the critical path, because we can kill a lookup that has already been sent to the cache if the validation tag indicates a mismatch.)

So we now have a single-cycle Fetch Predictor. Job over? Not even close!
We have accepted, for the sake of single-cycle performance, that the Fetch Predictor is not especially smart and not especially accurate. It's smart enough (basically do whatever worked last time) and good enough that most of its predictions are valid. But we want to augment it with a variety of specialized predictors that will take an additional cycle or two to validate the Fetch prediction, but if they disagree we can will the Fetch while the costs are still low, as previously discussed.

One obvious specialized predictor is the return address stack, since it's practically impossible to predict the address of a `RET` instruction without it, and trivial to do so accurately with such a stack. The main issue with a return address stack is simply making sure that you correct it (somehow) when something unexpected happens, most obviously an incorrectly speculated code path that generates an unmatched return or call before being corrected; but also a code path (using exceptions or longjmp or whatever) that breaks the usual rules of call/return.

Another specialized case is loops. Loops have the characteristic that they repeat the same code over and over – until they don't! Ideally you want to record whatever the exit condition for the loop is (usually, not always, a fixed count) and use that rather than being fooled by the fact that this loop has jumped backwards 1000 times so the way to be is that it will always keep jumping backwards. Loops also raise a whole set of possibilities for saving power. If you're confident in the loop, you can (for at least a few cycles) switch off the branch predictor, and the ITLB. Perhaps you can store the instructions in some sort of buffer and also switch off the I-cache?

Then there is what's traditionally called *the* branch predictor, the Branch Direction Predictor, the thing that guesses whether a given "branch based on condition" will or will not branch. The easiest versions of this are *local* predictors. Each branch is treated in isolation, and has, for example, a two bit saturating counter associated with it. The counter goes up each time the branch is taken, down each time it's

not taken, and you guess the direction based on the current count value.
This is implemented like the Fetch Predictor – use some number, say 14, bits from the PC of the branch to index into a table of these counters. Like always, lookup can mispredict for two reasons – maybe the prediction was just wrong? or maybe the prediction would have been correct except aliasing meant that two branches were using the same predictor slot and kept confusing each other.

Even with this super-simple local predictor, there are improvements possible, but the big improvement is to use non-local prediction. This assumes that the best way to guess a branch's direction is not what it did last time, but what was the pattern of code that got us to this particular branch. This pattern of code is most easily encoded in a history vector that records, eg whether the last N conditional branches were taken or not taken. This history vector is hashed with the PC (eg use the last 14 taken/not-taken decisions xor'd with the lowest 14 bits of the PC) to index a two-bit saturating counter used as before.
From this point on you can go wild with variations, but the current world champion (literally! https://jilp.org/cbp2016/program.html) is named TAGE, and it's basically an extension of the above history vector idea, but implemented in a way that allows for very long histories. The other trick you want to include is to generalize from the this basic history vector to a more sophisticated path history. Imagine identifying the taken path of execution as, say a sequence of branch target addresses. Obviously this path uniquely identifies a path of execution (starting from some point earlier); just as obviously it's a lot of data! But as usual we can hash it down to something a lot smaller, but still (usually) identifying a unique path. So, for example, for every taken branch, we can shift the path history by 3 bits, then xor in the new branch target. And once again one can go wild with variations on this theme.
The end point of all this is that you want to be able to identify each unique path of execution (over some number of previous branch points), so that all your prediction statistics (eg your counters that go up or down) are associated with that unique path, and we can extract whatever is predicable and associated with that path.

Let's consider the above in more detail.
First suppose we have a local predictor, but we index into our local predictor using the *full* PC (this is a thought experiment!) Even with this crazy amount of storage, the best we can predict for any branch is essentially "what it did last time". We cannot track well structured patterns for this branch (it keeps alternating taken/not taken), and we cannot exploit correlation (whenever that branch is taken, this one is usually not taken). So prediction is adequate, not great, even before aliasing. Implementing this predictor in a practical way means we cannot use the full PC as index, just, say, 12 bits; which means beyond the theoretical inaccuracy already discussed, we now add in some degree of inaccuracy from address aliasing, where two different branches find themselves allocated the same slot in the table (because the lowest 12 bits of their PC's match), so they keep fighting each other over how to update the counter.

Now introduce branch history. Once again imagine an insanely unlimited amount of space. We can construct a perfect history for every execution of every branch – we know that to get to that particular branch we followed a path of fifty thousand previous branches in this order, with the path consisting of

something like the address of every taken branch and the branch direction of every conditional branch along the way.
Now apart from impracticality in storing this lot(!) this does not actually help us! If we know that one (and only one) very particular path resulted in a taken branch, so what? The next time we reach that branch the path will have grown by a few entries, the branch corresponds to a different (not yet recorded) history, and we have no prediction.
Clearly we want some branch history – but not perfect branch history. How much?
What you probably want is something approximately like: “match the branch history backwards for as many entries as possible before the number of matching paths drops below a statistically significant level”. Sometimes the closest matching path we can find (with, say, at least 8 data points of how the branch was taken) is a match over the past three branch events; ie for this case we’d want to use a branch history of three. But sometimes a branch 200 events back is tightly correlated with this particular branch and you want to use a constructed branch history that looks something like “use events around 200 entries back, then skip 180 events, then use the newer events” and this synthetic branch history will match a number of branches, and so will be a good predictor (will be tightly correlated with) the current branch.
Obviously this is still impossible to implement. But how can we use the idea?
Firstly we implement the branch history, as discussed, by extracting a few bits on each branch event, from some combination of the branch address, the branch target, whether the branch was taken, etc etc, and folding those into a path vector which might be something like 256 bits long.
Secondly we hash down this path vector into a variety of shorter indexes (let’s say each 12 bits long) that fold in varying amounts of the path vector and the branch PC. At one extreme, we just use the lowest 12 bits of the branch PC and we have a fully local index. The next index may use the first two bits of the history folded in with 10 bits of the branch PC. We do this using geometrically more (that’s the G, for Geometric, in both O-GEHL and TAGE”) more of the history bits, sot 4 bits next time, then 8, 16, all the way to all the bits of the history. Once we get to the longer history lengths, we may drop some of the bits in the middle, on the assumption that the correlation is as described before, between an event 200 branches back and now, with most of the intervening events irrelevant.
This gives us something like nine indexes, into nine tables where for each table entry we maintain a 3 bit saturating counter. For each entry we also have 2 “usefulness” bits and a tag.
The usefulness bits are for updating the predictor; they mark entries that will be the best choice for sacrifice when we need to overwrite an entry.
The tag is a different hash of some combination of the PC and history bits from what we used for the index. The hope is that if we match on both the index (finding this entry) and the tag, then it’s highly likely this entry is not aliasing, it really refers to a single combination of path+endpoint branch.

This seems complicated, but not really.
The index based on zero bits case you understand: every time this branch is taken, its counter (as determined by the lowest 12 bits of the PC) goes up, not taken the counter goes down.
The index based on 2 bits case is essentially the same: for each path that had a certain structure (eg two branches immediately prior to this branch were not taken) again we modify the counter up or

down when this branch is taken or not taken.
And so so on across nine tables.
Your immediate response is probably one of two things:
- how can this possibly work? Isn't every branch history really unique? Well, yes, if you go out to the start of program execution. But you agree that the branch history going back two branches is probably not unique? There are just four possible taken/not taken patterns in the branches before this one, and in fact most of the time one of those four is strongly dominant, the other three almost never happen. And so it goes as you go further backward. The number of possibilities grows, of course, incredibly fast. But the number of actually executed paths, in most real code, grows much slower; most code follows a few, predictable, patterns of branches.
- then you worry about the reverse problem – doesn't this lead to crazy amounts of aliasing? Well again not really. Essentially we are hashing very long very different strings, and as long as our hash is not too dense (let's say we want to track 20,000 branches, but we are providing 36,000 hash slots) it's unlikely that two strings will match to the same hash slot; and we catch almost all those cases via the tag comparison.
So with those out of the way, the only question then is, we create nine indexes and perform nine lookups. Which one do we use?
For prediction we use the longest match that works (ie matches tag as well as index).
For training, we use the usefulness bits to decide which table to update.

This should give you enough of the idea that, if you want details, you can read the TAGE paper, (2006) https://www.irisa.fr/caps/people/seznec/JILP-COTTAGE.pdf *A case for (partially)- tagged geometric history length predictors*, without being overwhelmed. I'd recommend it! It's a beautiful paper, easy to understand (once you have the basic ideas in place as above), and it covers the particular details of things like how you choose which table to update, and why.
This same idea (use long history vectors, and pull out the best match from matches at various length) can be used for other predictions, both indirect branches and value prediction, though in these variants you have to modify the "counter" that are using. For branches a single counter can track both the prediction (taken, not taken) and the confidence; for other case you need two separate storage items for these two tasks, eg a storage for a branch target address, and a separate counter of "how often has this address been correct vs failed".

What matters for our purposes is that humanity knows how to build remarkably accurate branch direction predictors, but these are slow enough that, to get real value out of them, you need a Fetch implementation like I have described – something fast enough to generate a prediction every cycle, plus a mechanism that can validate those predictions over the next few cycles. In particular for any particular Fetch Group, in parallel with the Fetch Group being loaded from the I-cache
- you want to know where the conditional branches are in the Fetch Group
- you want to test each such branch (before the last) to validate that it was not taken

- you want to validate that the last branch in the Fetch Group (if it ends the Fetch Group and is a condi-

tional branch) was taken.
If you think about it, this is not exactly trivial! You could mark conditional branches in each cache line via pre-decode, but that doesn't help because, ideally, you'd be doing Fetch Group validation in parallel with cache access, to generate a correction as soon as possible.
I have no idea how Apple does it, but my guess would be that it involves the following elements
- the Fetch Predictor doesn't just store the next target address and the number of instructions to fetch; it also has a mini-map of the next Fetch Group. At the very least this would be a bitmap of which instructions in the Fetch Group are conditional branches; there may also be value in indicating indirect branches and returns (because if you know those are not present, you can save power by not activating those predictors?)

- even if you now know the PCs of the conditional branches that you now need to look up in your fancy TAGE predictor, there's a question of volume. Some Fetch Groups are short and have zero or one conditional branch, some may have many (all but the last hopefully not taken). How many conditional branches per cycle are you prepared to look up in your TAGE box? I suspect the optimal answer to this is to make the Branch Direction Prediction Validator machine yet another asynchronous machine that is coupled to Fetch via a queue. So the Branch Direction Prediction Validator is fed (via a queue) a list of "PC's to check, and the expected taken/not-take status", processes some number (two?) of these per cycle, and generates results (validation succeeded, validation failed), to be given to a pipeline stage somewhere after I-cache to check their fetched instructions against this queue to discover a mismatch as soon as possible.
In the worst possible case (code that's just nothing but a sequence of non-taken conditional branches one after the other) if the queue holding these conditional branch PC's to validate fills up then, like when any queue fills up, upstream (ie Fetch) will be informed and will pause until the queue frees up as much space as required for progress.

The other common type of predictor is the indirect branch predictor. If all that did was guess the previous indirect branch target of this particular branch, that would be an adequate guess but would also add nothing to the Fetch Predictor. However TAGE ideas can also be used for indirect branch prediction, (2011) https://hal.inria.fr/file/index/docid/639041/filename/ITTAGE.pdf *A 64-Kbytes ITTAGE indirect branch predictor*, so you can likewise use a heavyweight but slow predictor for these uncommon cases.

There are also weirder specialized-case predictors that are possible, and if you look through the JILP Championship papers you'll see some examples of them, but it's unclear whether any industrial CPU uses them.

As usual, we now see what we can learn (via patents or experimentation) of the above hypotheses.
We start with (2012) https://patents.google.com/patent/US20140075156A1 *Fetch width predictor*, which validates much of the above. You should make the effort to read through it, because you will see just

how much it matches my description above.

Beyond the explanations above, we also learn a few implementation details.
- Even as of 2012 (so around A6), Apple's maximum Fetch Width appears to have been 32 bytes. You may think of this as 8 instructions (already large for a 3-wide CPU) but remember this is ARMv7 days, so it could 16 Thumb instructions. Point is, Apple appreciated from the start that when you can gulp in as many instructions as possible, you do so, to make up for all the cycles where you're recovering from misprediction, waiting for cache misses, or whatever.
- Likewise even in that machine, Apple's Fetch Groups could straddle two consecutive cache lines.
- The index into the Fetch table appears to have a few higher order bits xor'd in with the low bits (which provide the primary entropy). My hypothesis for this is the same as when we saw the same thing being done for other predictors – it's possible that the low bits of an address are not perfectly uniform (linkers like to align things to page boundaries, compilers may believe it makes sense to align things to cache-line boundaries) and if the low bits are not perfectly uniform then you can boost the entropy slightly by mixing in a few higher order bits.
- Each entry in the Fetch Predictor is tagged by higher order bits of the PC. So, as I suggested, the system can at least catch aliasing (when an invalid entry is pulled up) and try to do something appropriate for the "zero fetch knowledge" case rather than just believing the incorrect entry. This is probably most useful, as I suggested, for minimizing the damage after context switches.
- A technical concern with Fetch Predictors is what do you do with a very long Fetch Group? For example suppose that we have an unrolled loop body that is 30 instructions long, but we expect the usual Fetch Group to be 16 instructions long. There are at least two issues. One is how many bits do we use to store the Fetch Group length; more difficult is what if we want to store auxiliary information related to the Fetch Group (for example, as I suggested, the locations of conditional branches and perhaps also other types of branches)?
The standard answer in the academic literature for this is you allow "overflow" Fetch Groups. Basically for the address that corresponds to 16 instructions into the long Fetch Group, create a fake entry that corresponds to the next 14 instructions. It's fake in the sense that it doesn't represent the *target* of a branch, like most Fetch Group entries, but it's perfectly legitimate in the sense that it represents a group of 14 sequential instructions, to be loaded from this address, and treated like any other Fetch Group. And Apple appears to be following this traditional answer.

Submitted at essentially the same time, we have (2012) https://patents.google.com/patent/US20140089647A1 *Branch Predictor for Wide Issue, Arbitrarily Aligned Fetch* filling in a few more details. These include
- The Branch Direction Predictor (as of 2012) was the O-GEHL version of the Perceptron predictor, which was considered to be the best predictor around that time before TAGE took the crown. (O-GEHL is an early version of the primary TAGE idea of how to handle long histories, while Perceptron refers to how to convert those histories into a tangible prediction).

- The Branch Predictor tracks its confidence level and uses this, among other things, to indicate whether further training is needed. (Most direction predictors calculate a number, maybe between 0 and 7, or maybe between -15 and 15. It's obvious that one end of the range vs the other end corresponds to taken vs not taken; and one can consider how far the number is from the extremes, say how close it is to 0 as opposed to ±15.). This training request is attached to the branch instruction early in the pipeline and propagates with the branch right up till Retire.
Suppose we have a branch for which we are very confident. Then it makes no sense to continue training the branch while it is predicting accurately; further training is just a waste of energy. Hence the value of this training request. Of course regardless of the training request, an incorrectly predicted branch has to transmitted to training!

- What about the issue of how to handle multiple branches in a Fetch Group? The Apple answer (at least as of 2012) is clumsy and inelegant, but not as bad as it first looks. Consider any of the direction predictors described earlier; easiest to imagine is the basic 2-bit saturating counter local predictor. We imagined our table indexed by the low bits of the branch PC, and the table holding a 2-bit counter for that PC.
But now assume that the table is indexed by the low bits of a cache line. In other words, suppose we were using 14 bits of the PC. A cache line is 128B, so 7 bits. So strip off the lowest 7 bits, and our table now has 7 bits (128) entries. Each entry is now a row of counters -- one for each instruction in the cache line, so 32 counters.
So if we want to access one particular counter, we would use the appropriate address bits (bits 7..14) of the branch's PC to find the appropriate row in the table, and then use the 5 bits 2..6 to identify the appropriate counter of the 32 counters. (Of course bits 0..1 of an instruction are always 0 and irrelevant to anything.)
The win in this design is that with a single Fetch Group PC we can identify the appropriate row of the Branch Direction Predictor, which holds prediction data for all possible branches in that line, and we can read out the entire row. At the point where we wish to actually validate the predictions in the Fetch Group (maybe at Decode, maybe just after I-cache access) we at that point know where the branches are in the Fetch Group , so we know which values from this set of 32 possible values to look at.

Your first impression on looking at that design is that sure, it may work, but doesn't it waste so much space? All those possible counter locations that correspond to non-branch instructions. No!!! What we have done is simply rearrange the same data that was present in the original (non-cache-line) design, with the same degree of aliasing, neither more nor less! A single row of the predictor does not correspond to a single cache line of instructions; it corresponds to all the cache lines of instructions that alias to this row (ie that have the same lowest 7 address bits of the cache line address). So while one particular I-cache line may use 8 of the 32 slots, another may use 5, and another may use 7. Aliasing exists – but no worse (or better) than before.

You second impression might be that no way this could actually work for a global (as opposed to a local) direction predictor, especially with the multiple tables and multiple weights of O-GEHL Percep-

tron. But it does! It all works as I described; just operate the predictor as you did before, but use as the address (to decide in which row to store a value) the 7 cache line bits, not the full PC address; and then use the remaining low 5 bits to decide which entry of the row to read from (when making a prediction) or write to (when training the predictor). The patent gives the full details.

So technically this is, IMHO, pretty cool. It uses a little more power than one would like, but solves the multiple branch location problem in a way that I never thought of.

By 2015 Apple have moved on to a proper TAGE-like system, but they add one interesting twist: (2015) https://patents.google.com/patent/US10719327B1 *Branch prediction system*. We described how, with TAGE, we construct a number of indices (in my example nine) which are used to look up in nine tables incorporating each incorporating ever longer runs of history. We also pointed out that the tagging of each entry means that aliasing should be rare. That's good, since it means two different branches won't be fighting each other over whether to increase or decrease the branch direction; but it means the usual problem with direct-mapped caches, ie that only one entry with the index hash can live in the cache at one time. We can fix this with normal caches via going to two- or four-way set associative, and Apple suggest doing the same thing for TAGE. Each of the TAGE tables can be implemented not as a direct-mapped cache but as two- or four-way, with different tables having a different number of ways (and a different number of indices) as best suggested by simulations.

Apple also take interesting advantage of this TAGE structure for energy savings. The lowest power-saving modes stop the clock but continue to power memories (include the branch predictor memories), but more aggressive power saving modes start to cut power to SRAMs and thus lose memory contents. Because TAGE has multiple memories, one can make choices about which of these memories (associated with different history lengths) to cut first. The two main ideas in
(2016) https://patents.google.com/patent/US10223123B1 *Methods for partially saving a branch predictor state* are
- maintain a measure of how many useful values are in each of the tables. Obviously you'd prefer to cut power to tables with fewer useful entries.
- but the tables associated with longer histories take longer to build up, so we need to balance these, but preferentially cut power to the shorter-history tables.
The patent also points out that, once a table has power restored, it's still not useful until it has at least one entry, so it might as well be kept powered off until training deposits one (or more) valid entries into it.

Now let's consider indirect branch prediction. The most trivial solution would be a table indexed by some number of PC address bits, and holding the most recent target of the branch that matched these PC address bits. This captures one common case, where vptr's of some form use a value that changes slowly or never. But it doesn't capture other cases where there is a pattern to the changing value of a vptr, for example the vptr alternates between two different values. To capture this sort of structure once again you want to use a history/path vector for your indexing.

With this background, let's examine the baseline Apple indirect branch predictor (2013) https://patents.-google.com/patent/US9311100B2 *Usefulness indication for indirect branch prediction training*.
This iteration of the predictor (as always, a long time ago, and probably replaced by something)like ITTAGE/COTTAGE) has the following form:
- the predictor consists of two tables. The tables are read in parallel. One table is indexed by history data, the other by PC. A hit in the history data table is preferred over a hit in the PC table if both match. (This is a rare case, lots of things would have to alias, and my guess is that the thinking is the history table is much larger, so aliasing [ie false match] is more likely in the smaller table)

- the history table has 1024 entries (indexed by the lowest 10 bits of path data)
- the PC table has 32 entries indexed by bits 5..9 of the PC (not the lowest bits, essentially the cache line index of the PC, for 32B caches lines)
- but the PC table is two way set associative. So each index provides a row with two entries in it. Both are tested, and the matching one (if any) used.
The diagram should make much of this clear.
(Ultimately, though you might not see it at first, this is an implementation of Driesen and Holzle (1998) http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=FCAA5DE4595332D2186B8E5CDC017A2C?doi=10.1.1.125.5000&rep=rep1&type=pdf *The Cascaded Predictor:*
Economical and Adaptive Branch Target Prediction.

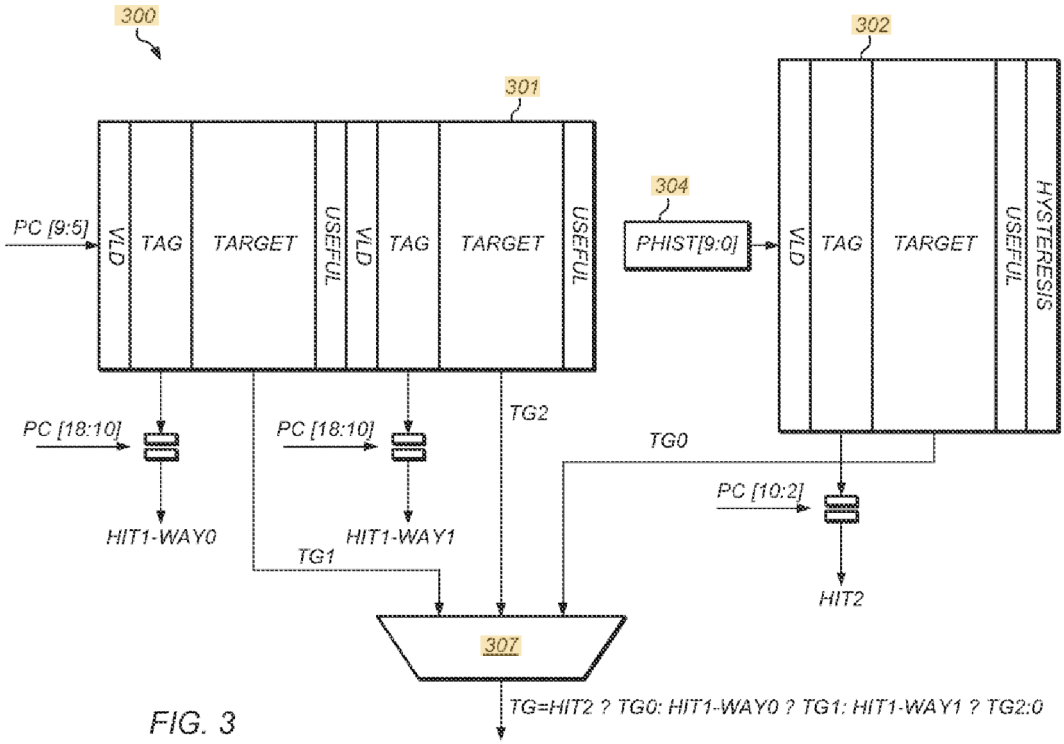


FIG. 3

So each entry has the usual validity bit. And a target which is the point of the exercise, telling us where the indirect branch should jump.

More interesting is the tag bit. The tag, as you can see, is bits 10..8 of the PC for one set of entries, and bits 2..10 of the PC for the other set of entries. So the idea for lookup is

- construct the index (one from PC, one from branch history vector)
- lookup the entry
- test if there is a match of the other bits of the PC against the tag.If so we can be reasonably confident that this entry corresponds to a value that was placed here earlier associated with this particular branch (ie chance of aliasing is low).

- if the entry is valid, and tag matches, then we have a prediction.

The other fields are used for training the predictor. You can look at the patent for the details but the approximate idea is that all entries have a usefulness count that's some small number if bits (possibly as low as one, but let's assume its two bits). Usefulness of a new entry begins at 0, but each time the entry hits and predicts successfully its usefulness count goes up. For the PC table, usefulness is used to replace entries, with the less useful of the two entries being thrown out when a new entry is required. For the history-based table it's used to decide if we should replace an existing entry with a new possible entry that matches this index value.
You can puzzle your way through the patent if you like, but to my quick scan it looks like the hysteresis value allows entries in the history-based table to make one mistake while retaining the current target. So imagine an indirect branch that usually goes to A but occasionally goes to B. The predictor will be trained to always give A as the result, and if makes one mistake where the actual target was B, it won't be trained to switch to B right away; it will only switch to B if it makes two successive mistakes.

You can see that this is already fairly sophisticated, trying to capture a variety of common patterns by various mechanisms (the two tables, the fact that one is 2-way, the tags to prevent aliasing, and the hysteresis bit). It's interesting to note that, even at this stage, the size of this predictor storage is ~1000 entries, call them 8 bytes long (if we have a 64B architecture, and are using say 6B of the actual address, with 2B for tag, usefulness and suchlike). So we're talking ~8kB for the indirect branch predictor, to give one a feel for the size.
At least one item that is, however, missing, is sophisticated use of a very long branch history as in IT-TAGE, but I would assume that by now with the M1 that has been corrected. Compare the 8kB above with the 64kB suggested in the IT-TAGE paper.

Now think of the generic issue. We have Fetch running asynchronously ahead the core, using the Fetch Predictor to generate a stream of Fetch addresses, with this stream being validated (and occasionally corrected) by the Branch Direction and Indirect Branch predictors. If the correction happens via these predictors, it just involves editing the instruction stream before it even hits Decode, so it doesn’t cause much pain.
How can we improve this situation? Well, even with the best predictors known to man some branches (especially indirect branches) are simply impossible to predict. Something like an interpreter that dispatches via a procptr is generating an essentially unpredictable stream of of indirect targets. What can we do about this?
In this case the Indirect Branch Predictor can’t give a useful branch target prediction, but it can say “this branch is unpredictable”. What can we do with that information? What we can do is halt Fetch while the instruction stream proceeds through the core. At some point the indirect branch will be executed by the core and the actual target value will be known. It can be passed back to Fetch, which can resume fetching.

This probably saves some time (whatever we guessed for the branch target is likely wrong, so going

ahead blindly would require the cost of a mispredict pipeline flush once the error was detected. But more importantly we save energy -- if execution for a few cycles is unlikely to achieve anything useful, better to wait out those few cycles till we can get back to useful work. As usual there's a patent (2010) https://patents.google.com/patent/US8555040B2 I*ndirect branch target predictor that prevents speculation if mispredict is expected*. The main thing you need is some sort of usefulness indicator for the branch; the patent described the simplest possible usefulness indicator (a non-prediction!) in the context of a rather simpler predictor than the 2013 predictor described above.

Obviously this same idea could be used for hard to predict directional branches (captured, eg, by their having a low confidence), but the payoff for a random direction guess (50% chance the subsequent code execution is useful) is probably worth the gamble.
However just a few hard to predict branches are a real barrier to further progress in ever deeper speculation, as described in a paper we have already seen, (2019) https://arxiv.org/pdf/1906.08170.pdf *Branch Prediction Is Not A Solved Problem*. In principle, with a well-functioning checkpoint and misprediction recovery system, a hard-to-predict branch is not a catastrophe. Mainly what will be lost is the work between incorrect Fetch past the branch, and the resolution of the branch, so ~10..20 cycles of work.

Where this becomes a catastrophe is the pattern of: a branch that depends on a load that misses to DRAM, because now the entire machine state, the entire ROB and a few hundred cycles worth of speculative work will be flushed on mispredict. This fact suggests a few possibilities

- can we detect this particular circumstance (not just a hard to predict branch, but that its resolution will take a long time)? This seems feasible, and if so, should we perhaps halt the machine until the branch is resolved? In the past it was argued that code that proceeded down the wrong path for some time was not as much of a waste as might appear, because the code frequently touched I- and D-addresses that would be needed by the correct path, and so it did useful work by pulling those lines in advance. It's unclear to me, with the most modern I- and D-prefetchers the extent to which this is still valuable.
This option is best if our primary concern is to save energy.

- alternatively, if the pattern in real code is that these problematic branches (difficult to predict, and take a long time to resolve) are sparse in execution time (ie usually only one of them is active at a time), one could just detect these cases and split execution at that point to run down both paths! This is essentially a very simplified version of SMT. One needs to tag the two instruction streams, the registers they touch, the stores they queue up in the store queue, and other paraphernalia, with a one-bit indicator, and one needs a way, when the branch is resolved, to flush all the allocations associated with the other bit. Technically, probably, the trickiest piece would be something I discussed earlier, making sure that all the speculative state that could pollute branch and other predictors, is segregated appropriately and the incorrect half is flushed when the branch is resolved.
I'm unaware of a CPU (or even a paper) that does anything like this, but I have seen it occasionally suggested on the internet. Of course it relies on these problematic branches being sparse; the scheme I

am describing does not scale well to multiple successive problematic branches! (The point is not that these are rare; it's that they must not cluster together in time.)
On the other hand, is it worth the effort? Suppose we can store 1000 instructions worth of speculative state. In the baseline case, half the time a problematic load gives us 1000 instructions of progress (until we halt waiting for the load to return from DRAM), half the time it gives us zero. In the SMT-like case, every time we get 500 instructions worth of progress. Same consequence on average.
Is that actually better (even from an energy, not just a performance, viewpoint)?

- what if we could somehow detect the load far in advance and prefetch it? If this were easy, the basic prefetchers would do the job, but maybe (given how rare these problematic branches are) it's possible to build a specialized predictor that's tailored to them? The *Not A Solved Problem* paper did not mention this, but to me it looks like a more promising approach than their suggestions.

- there is a concept called CPU Runahead, eg (2020) https://users.elis.ugent.be/~leeckhou/papers/hpca2020.pdf *Precise Runahead Execution*. The idea is that, under certain circumstances (varying depending on the exact design) the CPU switches to a mode where it tries to explore the future execution stream as rapidly as possible, specifically trying to prefetch as much data as possible. So it drops certain instructions (like FP) that probably won't affect future loads, and may guess at various values (like the values of loads that have missed to DRAM, and will affect future loads).
One could fuse this idea with the earlier SMT idea and imagine a design that, on encountering a problematic load, switches to runahead mode where there's no expectation that 50% of the time these instructions will be retained; rather the (tagged) instruction stream is mangled and partially executed in the expectation that it and all the state it touches, will all be thrown away; but will ideally prefetch a useful amount of material while it executes. This is the sort of choice that probably never makes sense for a battery design, but may well make sense for an AC design.

So we have three immediate options, neither of which is especially appealing! Well, it's a problem for the future, we still have tricks to implement today within the constraints of existing predictors.

Now that we understand the general outlines of Fetch and Branch Prediction, there are a few interesting details to consider. (There are also many less interesting technical details I'll omit!)

One problem with the Fetch Predictor as described is that it can't handle alternating branches well. Consider code that alternately (depending on whether a counter is even or odd) goes down two different paths. This is an easy pattern for a sophisticated history/path based predictor to catch, but what can we do using a simple single-cycle predictor?
If the Fetch Predictor is always representing what we did last time, then it is consistently 100% wrong! We can at least improve this to only 50% wrong by implementing hysteresis, ie some sort of delay, eg we have to see a change from the current prediction twice before we'll insert a replacement. The details of one way of doing this are in (2011) https://patents.google.com/patent/US20130151823A1 *Next fetch predictor training with hysteresis*, but are less important than the idea.

(50% still sounds like not a great success rate!
Fortunately
- if this alternating case is within a loop, it will be caught by a loop predictor and handled in a better way, as we will see when we get to loop predictors, AND
- the more recent Fetch Predictors, at least as of 2016 or so, are tagged not only by PC but also by a few bits of recent branch history. This allows the 2-way set associative Predictor to hold two different Fetch Predictions for the same PC, but different branch history, so trivial alternating cases can also be captured.

However (there's always an however!) now that we have hysteresis, it will take longer to flip a Fetch Predictor element that really should be flipped. I *think* this is the problem our next patent is trying to solve, but this one is written at such an abstract level, with no helpful examples, that I can't be sure. I think the idea with (2017) https://patents.google.com/patent/US10613867B1 *Suppressing pipeline redirection indications* is: suppose we have a situation of a very tight loop containing in the loop body a conditional branch. And suppose that the large TAGE predictor detects that its (high quality) prediction doesn't match the Fetch prediction. What to do?
The hysteresis answer, essentially, was redirect immediately, and suppress the Fetch Predictor update for this iteration.
This 2017 patent's answer is, suppress the redirection and temporarily delay the Fetch Predictor update. The idea, I think, is for very short loops (ie every cycle is a new pass through the loop body) allow the temporary Redirection Circuit to make a decision about how best to train the Fetch predictor for this branch (is is alternating? is it a permanent change to the previous case?) By suppressing the redirection yes, we get Fetch looping once or twice pulling in bad data each time, but we can flush that in time. Meanwhile we manage to run the loop through TAGE and Training a little faster than if we had redirected the loop in response to the bad prediction, so we're faster in training the Fetch Predictor to match to the ongoing loop.
To justify this somewhat, consider that Apple suggest the time from start to branch predictor output value is around 5 cycles. This means five cycles are lost on each round of redirect and restart; as opposed to just one or two extra cycles being lost if we hold off on redirect for one or two cycles once we have noticed a mismatch, to figure out an optimal value setting for the new Fetch Predictor entry.
But hey, if you can understand the patent and provide me with a better explanation, be my guest!

The Return Address Stack (RAS) seems sufficiently obvious that there's nothing much to patent in the main idea, but this changes when you start to think about it!

Here's are some issues that you ignore at first, then realize are important:

- Fetch prediction, as we have described it, won't work for returns. Using the mechanism described, we can do an adequate job, that will usually pull in the correct fetch stream, for conditional branches and

for function calls, but not for returns. If returns are treated as just a “follow this run of instruction by going to the same address as you did last time” they will frequently be mispredicted.
This means we need three things
+ a marker in Fetch Predictor entries that says “the jump that ends this Fetch Group is a return so treat it appropriately”
+ a miniRAS for the Fetch Predictor that’s not necessarily too large or too fancy, but which usually is able to push function return addresses when a Fetch Group begins with a function call, and pop on predicted return.
+ which in turn tells us we also need a marker in the fetch group saying indicating when a function return address needs to be pushed

This means in turn that we will have at least
- a MiniRAS being used by Fetch Prediction
- an “Execution” RAS being used by the proper Return Address Predictor. This one will be larger, and will try to keep things accurate in the face of various mistakes.
- the absolute truth RAS which is what is stored on the physical, in DRAM, stack at any given time, and which will be resorted to in the event that things go hopelessly wrong and the predictor makes no sense (for example when we revert to a thread after having context switched to a different thread).
The MiniRAS is not just smaller than the Execution RAS, at any given time it holds different contents. For example the MiniRAS may push, then pop, a return address over two successive cycles, before the PC’s associated with those two Fetch Groups have even been processed by the Execution RAS.

Next, the big obvious problem to be solved with a RAS is of the stack becoming corrupted under mis-speculation. Suppose, for example, that the speculation path we travel down at some point involves a function call (push an address on the RAS), but we discover the misspeculation and redirect Fetch to the correct spot, leaving that inappropriate return address on the RAS.
The MiniRAS is small enough and lighweight enough that I suspect it makes no attempt at correction. Suppose the Return Address Predictor notices a mismatch between Fetch has done and what the Execution RAS says. Best response is, at the same time that Fetch is being redirected to the correct address, to copy over the top few (presumably correct!) entries from the Execution RAS to the MiniRAS.

What about the Execution RAS? As usual there is a range of ever more complicated options.
To get started, think about what a stack means as HW implementation. Conceptually a stack is a (for now indefinitely long) array of storage, along with a number which we call ToS (Top of Stack) that tells you the top of the stack. What makes it a theoretical stack is that ToS can only be incremented or decremented by 1.
First level of correction is to ignore that rule. Maintain (somewhere...) the value of ToS. Then, in the scenario described what happens is
- we know the value of ToS pointing to the correct point in the array at the point before misspeculation
- the incorrect function call dumped an extra address on the stack, and incremented ToS but
- recovery means reverting ToS to the value before the mispredicted path that led to the function call.

This means that we need to have one piece of storage associated with the stack called ToS, along with the recovery ToS recorded at every point we might need to revert to. At least approximately (we'll figure out details later) let's assume something like
-ToS can change (correctly or incorrectly) only at the execution of calls and returns
- so in principle we could store just at each call and return what the ToS value was at that point
- then on mispredict recovery we'd run through the ROB looking for the first call/return older than the recovery point to read its ROB
- that works, but seems suboptimal. Presumably we also need to to store the appropriate value of ToS in Checkpoints, and maybe we want some sort of structure associated with the ROB that is dedicated to this job of holding ToS, so that we can find the correct recovery version faster than via a sequential search backwards.

OK, so we have figured out, by logic, that we need a ToS value for the stack, a recovery ToS value associated with each call/return, and somewhere to store these recovery values that's, hopefully, fairly easy to search.

Next complication is that the recovery scheme I described above only fixes half the problem. What I described was the case that
- we mispredict a call,
- the call pushes an inappropriate return address on the stack , increments ToS) (and saves recovery-ToS)
- but it's fine because we recover by finding recoveryToS and setting ToS to that correct value

Consider the reverse case
- we mispredict a return
- the return pops a value off the stack
- we continue down this bad path and mispredict a call
- the call will *overwrite* the storage slot that was being used by the return address

Our games with recoveryToS will not help us now. We can revert ToS to pointing to the correct storage slot, but the value in that slot is invalid and the correct value is gone!

With this understanding, you can now look at Skadron (1998) https://mrmgroup.cs.princeton.edu/papers/micro31_retstack.pdf *Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms,* which describes what I said in more detail.

So we have the problem that we also (in some fashion) want to save the return address, recoveryAddress, at the top of the stack.
Skadron's solution to this is to just save that return value in the same place as wherever we are storing the recoveryTOS.
In principle you don't have to store the recoveryTOS and recoveryAddress for every branch, because it

only changes at call/return, so you can use some indirection to store these values in one structure (associated with recent call/return) and have a short index into that structure associated with every recent branch.
Intel implemented, for the Penryn generation, a complicated scheme(based on (1997) http://esca.korea.ac.kr/teaching/com609_TESII/RAS/Recovery-Branch-Misprediction-1997.pdf *Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution*, that implemented a stack via a linked list and allowed pushing new values on the stack to "bypass" rather than overwrite older values, so that those older return values were still available on the stack if required.

I haven't seen anything that explicitly covers how Apple handle this, but their RAS-related patents suggest that the structure is extremely reliable, so it must be utilizing some recovery scheme. The main BIT patent (to be covered in a few pages) refers to some fields that are suerly related to RAS maintenance and recovery, but which are not explained.

We can deal with this by converting the stack to a "stack-like" structure that never overwrites data. This is fairly complicated, so make sure yo understand each step before moving to the next step.

- Consider a standard doubly-linked-list. Call the pointer to the head of the list ToS. It should be clear that you can implement a stack by obvious easy operations on this doubly-linked list. (ie think what it means to either pop an entry from this stack/list, or push an entry onto this stack/list).
The nodes of this list look something like (`return value; previous; next`) where previous and next are pointers.

- Now, consider an array of these nodes. Now we can make previous and next indices into this array, not generic pointers; and ToS is likewise an index into this array.
Once again, work out in your head what it looks like when you push or pop this stack.

- Now we will make this a "one-time-write" stack. Along with ToS, we add a second index (associated with the array), called Alloc. Alloc, like ToS, starts at 0, but Alloc has the property that it can only ever increase, never decrease.
So we push the first value onto the stack. This value is placed at 0, and both ToS and Alloc increament to 1. new entry looks like (val0; null; 1) at 0
We push a second value onto the stack. This value is placed at 1, and both ToS and Alloc increment to 2.
new entry looks like (val1; 0; 2) at 1
We pop the stack. ToS reverts to 1, but Alloc stays at 2.
We push a third value on the stack. We will need to write, so Alloc increments and ToS is set to Alloc=3.
new entry looks like (val2; 0; 3) at 2
And so it goes. Note that
- this still behaves like a stack. We know how to push, we know how to pop, just follow the linked list links.
- but older values in the stack are never overwritten, they are just snipped out of the linked list.

This has two consequences:
+ The previous scheme we described of storing a recoveryToS with each call/return will now still work! The value at the recoveryToS is still valid, and the links from it backward down the stack are still valid. (Forward links are a random mess, but who cares; they represent state from the invalid path).
+ I've described this in terms of unlimited storage for the stack or the linked list. As far as stacks go, the "frequently accessed" depth of most code is, I don't know, probably covered just fine by 64 entries. Some code will exceed this (leading to mispredicts and generating some slowdown) but the real issue is with a budget of 64 entries, how long will you go between either under or overflow and encountering mispredict?
(I have described this via a doubly-linked list because I think that's easier to visualize and imagine in your head. But once you understand how it works, you will see that there are no conditions where you actually need the next point, only the previous pointer; so you can remove the next pointer and its updating.)

But with this linked-list scheme I described, we use up one element of our storage every time we call a function, not just every time we increase the depth of our function calls. This is very different scaling! So we will use up any practical array for our linked list fairly rapidly.
There are multiple ways one can imagine for solving this, but here's one that somewhat matches what Intel does, simplified and ignoring unimportant details:
- maintain both the stack RAS and the linked-list RAS.
- entries in the linked list each have a color which tracks wraparound. So entries start off red, after we wraparound the end of the list, entries we write are green, then with a second wraparound back to red.
- each time your store a ToS, store both the stack value and the linked-list value, and the color of that value.
- this scheme (or something equivalent) allows you to track when you have overwritten a value in the linked list ToS
- when you need to recover from a mispredict, first look at the linked list recoveryToS. If the color matches the recovery color (ie that value has not been overwritten) we can trust the linked list RAS, so we use that stack; otherwise we use the value from the stack RAS.

The Intel scheme is described in (2008) https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.155&rep=rep1&type=pdf *Improvements in the Intels Coret2 Penryn Processor Family Architecture and Microarchitecture*. If you want to know the full details, look at (2001) https://patents.google.com/patent/US20030120906A1 *Return address stack*.

Before we discuss what Apple does, let's return to an issue we skimmed over. We need somewhere to store the the recoveryToS for each call and return.
I described doing so in the RoB slot, but that matches a pattern we have seen frequently before – don't make a general structure larger, rather use indirection to store data in a task appropriate structure.
The issue we have seen here is one that is actually more general: every branch, until it retires, needs to hold onto some context information. We have seen this context information for calls and returns as being recoveryToS values. But for other branches, we will want to store a bunch of data related to the

state of the machine at the point of the branch, things like the history/path vector, and the target of the branch. These may seem no longer necessary
- we predicted the branch long ago at Fetch. Correct or not, the prediction is done
- we compared the prediction to the appropriate value at branch Execution. Again, right or wrong, that's over. All the ROB needs to know is if we mispredicted, and if so where to redirect Fetch.
No! You are forgetting that we also need to train/maintain the Predictors. And what have I kept saying? We don't want to train them on invalid data!
So we want an additional structure (almost like the equivalent of the History File or Register File) for branches, that holds data associated with each branch through approximately the period from when the branch is Decoded to when the branch is Retired. This data structure is called the Branch Information Table. Think of it like an extension of the ROB, but just for branches; in actual implementation I assume branches in the ROB have an index that points into the Branch Information Table (BIT).
We will get into details but, approximately, there are two large subtables, the BIT and the TBIT (Taken Branch Information Table) each with subsections for different types of branch (eg conditional vs call/return). Entries in the BIT/TBIT are the sort of stuff described –what you need to maintain the prediction machinery. So the call/return slots will, among other things, store ToS values; and the conditional branches will, among other things, store their history/path vectors and targets, whatever is required to train the Predictors. On Retirement these fields will be copied out of the BIT/TBIT and sent to to predictors to be handled appropriately.

Apple's patent is (2011) https://patents.google.com/patent/US9354886B2 *Maintaining the integrity of an execution return address stack*. This patent is extremely confusing until you already understand what it is trying to say. You have to read it with the Intel patent in mind, and even so still read between the lines. This is my best attempt.
As terminology what I am calling the MiniRAS, they call the Speculative RAS. What they call the Execution RAS is also not a great name. It is associated with Execution, yes, but it is still essentially a speculative structure :-(They mention as an aside that they only use these two structures, which may seem obvious except that this is in reaction to Intel, who use about five of these RAS structures! So Apple have one for Fetch, and another that's modified at Execution. In particular the Execution RAS (implemented in the linked list form) has to handle overflow differently from Intel, it can't, as Intel does, either drop back to the stack version occasionally.

Apple tell us one way they prevent their Execution RAS from being polluted, beyond everything we have already covered. The problem (I think) that they are trying to deal with is Replay, though the patent explanation is terrible. We have described how instructions can be scheduled speculatively, on the assumption that the load they depend on will succeed, but if fails they will be re-executed. This means that all execution has to be *idempotent*, it must not ultimately matter if a given instruction is executed one or twice or three times; all that matters is that the final execution does the right thing and everything done by the prior executions is overwritten.
We have followed that in great detail in terms of standard integer (or FP) instructions, and how they can be re-executed multiple times because their values are always written to the same physical regis-

ter, so the final execution after multiple replays will overwrite whatever was incorrectly written earlier to that register.
Now you might not think that there could be any dependencies between a branch and a load, but there is, in the form of either `RET` or `BR Xn`, the latter being an indirect call to a function pointed to by Xn, the former being a return which is an indirect branch through register X30. In both cases register Xn or X30 could be filled in by a load, or a calculation dependent on a load. Thus if the load replays, the RET or indirect branch will replay. Will that execution be idempotent? No, not by itself, because it will result in a second push or pop onto the Execution RAS. Hold that thought, while we rephrase what we said in different words (to make the point), and read below while you look at the diagrams in the patent.

+ Each entry in the BIT is a separate branch execution. It may be the same branch as far as its address in the cache is concerned (think of the branch at the end of a loop that is called repeatedly to test loop exit; every execution *instance* of that branch will have an entry in the BIT).
+ The Index in the BIT is some way to differentiate and select a particular instance in the BIT. The best way to think of it is that it's the same number as the ROB slot that the branch occupies, so we have a connection between a given ROB slot and a given BIT slot.
+ The entry in the BIT also has one more bit, the "first time" bit. What's that? Didn't I just spend ten sentences saying that each BIT entry corresponded to a unique execution of a branch? Yes but language is difficult! This is the bit that handles Replay.
The basic flow is
-- at Decode time, when we allocate ROB slots, we also create an entry in the BIT and mark it as "first time".
-- Things continue well, we execute the branch, we clear the "first time" flag.
-- Ideally that's the end of the story. But maybe there's a Replay of the load, which feeds into Replay of dependent instructions.
-- Do the branch is executed again. It recalculates the address correctly (ie looks at register X30/xn which, presumably, now has the correct value from the load) *but* it does the correct thing with the RAS.
For a push (ie an indirect function call) it *replaces* what it put on the stack last time (a junk address corresponding to whatever random junk was in xn since the load failed).
For a pop (ie a return) it does nothing because the the pop executed the first time and ToS has been moved down to its correct location.

So basically we ensure, under very technical circumstances (Replay of a load that's feeding a return or indirect call) the Execution RAS maintains integrity.

The second question of interest is how does Apple deal with the constant growth of the Execution RAS? Remember that when we left Intel, we were at the point that
- we understood the reason why you want a one-time-write stack (implemented as a linked list)
- but that implementation has the problem that it uses up new entries in the storage for every call (as it must, that's the whole point!)
- so we have this somewhat intricate business of wrapping around the storage buffer, and flipping

color, to try to reuse storage with as little damage as possible.
Here's my guess as to what Apple does.
The Intel version is creating a stack via a linked list, but takes the stack aspect more seriously than the linked list aspect. Forget that, treat the structure as a pure linked list implemented with a finite array of node-sized entries (say 90 or so). Each entry has a "valid" bit; also forget the Alloc index.
Now every time an entry needs to be allocated we can ahead down the buffer (with wraparound at the end of the buffer) looking for an invalid entry, which we use. This means we will soon have entries all over place looking like a random linked list; but the stack structure will be there when we follow the pointers.
The above constantly allocates new entries. When do we free entries? Well, the reason we don't want to overwrite an entry is that it may correspond to the return address of a `RET` that seemed not to matter while on a speculative path, but ultimately did matter. In other words, when a `RET` Retires, we can be absolutely certain that the slot in the RAS holding its return address is no longer relevant to anything; so when a `RET` Retires we can mark its slots invalid. This will now give us a process that is freeing slots as rapidly as we are filling in new slots and overflow is no longer an issue.
If this sounds familiar, it should remind you of how we find free registers in the register file, as discussed earlier...
(There is still the issue of what if you go really deep down a set of nested calls? At that point you have to just cull the oldest value in the stack, and accept the cost when you finally pop the stack; just like with the very traditional RAS that began this discussion. I have hypotheses as to how this might be done – eg how to track what counts as the oldest value – but we're already on such a limb as to this proposed implementation that further discussion makes no sense. Hopefully future patents will come to light explaining how Apple deals with the various types of RAS over/underflow).

One final side issue in the patent is they point out the obvious fact that if the front-end RAS (what I am calling the MiniRAS, what they call SRAS) is corrupted, you can recover adequately by copying values from the Execution RAS to the SRAS. Intel say the same thing in their patent, but they seem to suggest doing it on demand, one entry at a time; Apple do it wholesale whole stack at once. I guess that's different enough for a patent. Don't take this part too seriously because we move on to

(2013) https://patents.google.com/patent/US9405544B2 *Next fetch predictor return address stack*, which covers the miniRAS in more detail.
This patents (two years after the previous RAS patent) clarifies and improves much of the machinery.

The first nice improvement is that the main RAS (what was previously called the Execution RAS) is now maintained at Decode. Logically this makes much more sense. Decode is where BIT entries are allocated and, if you think about it, all you need to maintain the RAS is knowing that an instruction is a call or a return, which you know at Decode time. So Decode can either pop the RAS (return) or allocate a RAS slot, and we no longer have the worry about idempotent executions and ensuring that Replay of call or return pops or pushes the stack too often.

Secondly we have confirmed that some level of predecode is being done. Suppose that Fetch accesses lines in the I-cache that it has not seen recently, and so are not part of the Fetch Predictor. From the predecode bits, it will know that the Fetch Group it is sending downstream includes a call or a return, and it can behave appropriately – a return means generate the next Fetch address from the MiniRAS, a call means, as we saw, pause Fetch'ing until upstream can inform us as to the predicted target of the call. (Note that Decode can accurately fill in the return value of a call, even if it does not know where the call will be directed, eg an indirect call!)

The final improvement is that the MiniRAS is no longer really a full separate stack running in parallel with the primary RAS. The MiniRAS is only required if there is a return following a very recent call, so that that the call has not yet propagated to the primary RAS, so it can be very much smaller than the primary RAS. What's needed is to track how many calls and returns have been encountered as part of a Fetch Group, incrementing these at Fetch and decrementing at Decode, so that you know whether you should pop a return value from the MiniRAS, or use the value at the top of the main RAS.

I'm not much interested in security, but if you are, you might want to look at (2015) https://patents.-google.com/patent/US10867031B2 *Marking valid return targets*. The idea is to foil ROP by tagging calls and returns in such a way that a faked return (by placing an address on the physical DRAM stack and forcing a return to execute to that address) will not have a matching tag at the point where it returns. Mostly this is its own thing, the one interesting part as far as we are concerned is that it uses the RAS prediction mechanism to know when to check tags. Under normal control flow, the control flow happens via the RAS and MiniRAS which have not been attacked; the first indication of an attack is when the executed return does not match the predicted return address. So it's only under those conditions that the CPU toggles to paranoid mode, where it checks for a mismatch between the call site and the return site.

I'm not sure if the above idea was ever implemented in HW. It is followed by (2016) https://patents.-google.com/patent/US10409600B1 *Return-oriented programming (ROP)/jump oriented programming (JOP) attack protection*, which should look familiar; this is the PAC (Pointer Authentication) stuff that encodes the 2015 tag in a few high bits of the return address; so now it's pretty much always there, and always checked as part of logic that reads and writes the RAS.

The third patent in this space is (2018) https://patents.google.com/patent/US20200192673A1 *Indirect branch predictor security protection*. This builds on ideas we have seen before. We've seen that the branch predictor (in whatever form) tags each entry in its prediction table so that, even after the index hash that looked up this table is probably unique, we still compare the tag to a different hash of the PC and/or the path to get a stronger indication that we're dealing with the correct entry, not an aliased entry. The idea of the security patent is to augment that tag with a security tag which holds, among other things, the protection level, the process ID, the VM ID, and some of the high bits of the PC. This security tag is compared with the state of the machine, and if the two don't match, the prediction is not used.

This means that an attacker cannot seed the predictor so as to get the attacked code to speculatively go down certain paths for a few cycles before the speculation is discovered (ie the content of the SPECTRE exploit). Each entry in the tag prevents certain types of attacks, for example the process identifier prevents one app doing this against another app.
To my eye, the most interesting one is using some high bits of the PC. Normally one ignores the high bits of the PC in any sort of indexing or tagging because they carry so little information (ie code tends to hang around a limited region of the address space). But suppose we have JIT'd (ie untrusted) code that wants to attack the host process. As long as we place the JIT'd code somewhere that has different high PC bits from the host code (eg reserve the highest 1/8th of user process address space for JIT'd code), then storing a few of the high PC bits in the security tag is enough to prevent the JIT'd code from being able to create branch predictor entries that will affect the host code.
(Note that this patent also answers the question we raised about reusing branch prediction data across context switches. While such reuse is probably theoretically the optimal choice, security forces us down to the intermediate extreme of tagging branch data by processID.)

Now consider branches, direct calls, and even the `TBZ` and `CBZ` branches that test a register. All of these have the property that the target consists of an offset added to the PC. Suppose that when the branch is predecoded (ie when it is loaded into the I1 caches from I2) we calculate the target address there and then and store that in the I cache. This would save the latency of the addition, and the energy cost of many future address target additions and seems like a good idea. It is a good idea except there is one tricky detail. Consider the following set of issues:
- we have a line from a shared library, which includes a branch located at virtual address vA and physical address P.
- we predecode that line, and replace the the branch target address with an address tA which equals vA plus some offset.
- now we context switch to a second app which uses the same shared library, but aliases its placement so that the branch is located at virtual address vb (but still at the same physical address P). If this doesn't sound familiar, you need to go read about Position Independent Code, and Address Space Layout Randomization.
- this means that app B will see the cache line already in the I1 cache (because that cache is physically address), but it will see the target of the branch as tA (ie vA+offset) whereas it should see the target as (vB+offset).
Oh dear!

But the idea can be saved, somewhat. Suppose that rather than calculating the entire target address we calculate only the lowest fourteen bits. This will give us the page offset of the branch which is always the same. Then we actually use the branch we only have to sum the upper bits (a shorter sum, so faster and lower power). It actually gets even better because there are some paths for which we may want to look up the branch target in a predictor or cache, and if that predictor/cache is indexed by just

the lowest 14 bits of an address (or that mixed in some way with other data) then we can perform the first stage of the lookup immediately, in parallel with the sum to find the page number, then compare the page number with the tag of the indexed lookup. (This should sound just like an L1D cache lookup, because it's essentially the exact same trick!)
This is the content of (2014) https://patents.google.com/patent/US9940262B2 *Immediate branch recode that handles aliasing*.

Back to Fetch Predictor

Now that we've seen two interesting things predecode can do (mark branches of various types, compute branch target low bits), how much more can it do?
We've already seen (2014) https://patents.google.com/patent/US20160011875A1 *Undefined instruction recoding* suggesting a not very interesting option – detect all the possible *undefined instruction* encodings in an instruction and replace them with a single "undefined instruction" value.
But once you see the idea, many possibilities open up! Suppose we are willing to widen the "in-cache" instructions to 40 or even 64 bits wide. (Since all instructions have the same size, this does not complicate branching and addressing; with Thumb it might have been more painful.) While AArch64 has a pretty regular instruction encoding, the necessity to fit into 32 bits means some irregularity in the encoding. One could imagine various small cleanups like ensuring registers are always in exactly the same locations, all immediates have the same value, etc. Of course what's worth doing depends on details we don't know about the pain points in Apple's A64 decoder.

Another obvious case is mapping various types of NOPs down to a single canonical NOP.

One thing that is likely is that xzr is removed from many instructions. For example it's cute that the `MUL` instruction is simply multiply-add with xzr being added, but for anything beyond the simplest CPU, implementing `MUL` literally in this way is throwing away performance. I could imagine pre-decode also mapping many of these "use xzr to provide a simplified ISA" techniques to what are literally simpler (lower energy, one cycle less) opcodes. We've seen that xzr behaves differently for some situations, like MOV, and maybe this is a casualty of such pre-decode?

Another interesting direction for this I could imagine is to move the pre-decode unit up to the L2. Most cycles it's unused, and it seems having a single pre-decoder shared across all cores in a cluster is a better use of resources, and justifies expanding the pre-decoder to handle more cases.

Now we get to the more specialist patents.
(2013) https://patents.google.com/patent/US10901484B2 *Fetch predition* [sic] *circuit for reducing power consumption in a processor*. Many of the precise details in this make no sense to me (they may be obsolete relative to the M1 details), but the general idea is clear enough: since the Fetch Predictor contains information about whether the next Fetch Group contains (perhaps untaken) branches, indirect calls, and returns, each of the second level structures used to validate these predictions can be

powered by (probably via clock-gating) for the next cycle or so if it will not be required immediately.
In terms of details, I think what they are trying to say is that
- the Fetch Predictor array is split into two pieces. They are both indexed by the current PC, but one piece holds Fetch Groups that terminate with a target (ie a taken branch of some sort), the second piece holds Fetch Groups that run on to the next Fetch Group.
- this split obviously means some saved storage space (the second piece doesn't require a Target address); but it has an additional implication: the sequential Fetch Group piece can indicate not only what (presumably all non-taken conditional!) branches are present in the Fetch Group, but what branches will be present in the Fetch Group that it flows into. This means that we can know not just what predictors to power down for one cycle, but even for two or three cycles, and that may allow for some additional energy saving.
(We've stated repeatedly branches are dense in code, and that's true. But it's also true that some loops, especially in FP, are unrolled to become a single long loop say 100 instructions or so long without a branch. I think Apple is trying to take advantage of this sort of thing whenever it's encountered, both in the space savings in the Fetch Predictor, and in the energy savings of allowing the branch predictors to sleep for multiple cycles.)

(2016) https://patents.google.com/patent/US10203959B1 *Subroutine power optimiztion* [sic] builds on this above idea by noting that a second common and very stable pattern is of a Fetch Group that terminates in an unconditional call. In that case, just like the previous sequential prediction case, we can make useful predictions about what branch predictors will be utilized over the next two (and perhaps a little further) cycles.
A second small tweak is: suppose a new cache line is pulled into the I-cache. This cache line will be pre-decoded before it hits I1 and the rest of the core, and that pre-decode will note the presence of branches. This information can be conveyed to Fetch which can, once again, use to it determine whether branch predictors need to be powered up or not. (An obvious case is if the line includes no branches, or no indirect branches; a less obvious case is if the first branch is unconditional so no prediction is needed, we know it will be taken, and this will be handled when the branch hits Decode.)
A third way to exploit these new-to-the-cache lines combines the above two; suppose that the first instruction in the line is a call, followed later by an unconditional branch of some sort. Then, even though the Fetch Predictor may not know this line (it's new to the cache) we can know, and store away, that when the call returns to this line, the number of instructions to load from I-cache should not be the rest of the line, but just the instructions up to the unconditional branch. (This may seem like a rare situation, but remember return, or calling a second subroutine, are both unconditional branches...)

Getting back to the 2014 patent, given that you are packing the Fetch Predictor with various energy-saving data, you can go further! For example you can pack both way information and ITLB-translation information into the Fetch Predictor, meaning that now, as long as we are successfully running out of the Fetch Predictor, both the ITLB and tags comparison of the I1 cache can be slept. Very nice!
(This facet of the patent builds on (2010) https://patents.google.com/patent/US8914580B2 *Reducing cache*

power consumption for sequential accesses, which is a much less ambitious version of the idea; essentially suppressing the ITLB and tags lookup in the case where sequential Fetch reads instructions from the same I-cache line as was accessed in the previous cycle; also upon (2013) https://patents.google.com/patent/US9311098B2 *Mechanism for reducing cache power consumption using cache way prediction*, which applies the same sort of idea to sequence the way activation in the correct order when following straight line code or known branches.)

I'm going to out on a limb here and suggest that the Fetch Predictor is defined by "slices". The slice concept seems to be a common feature across many parts of an Apple CPU. The easiest example, given what we have seen so far, is to consider the ROB. We described the ROB as consisting of multiple "rows" where each row can hold up to six "simple" instructions, and one "failable" instruction. If you think of the ROB as a two dimensional table, it consists of multiple rows, and a row consists of six entries of type A, and a seventh entry of type B. The columns, in this structure, are, I think, what Apple calls slices. So for the ROB there one slice (or perhaps six slices?) that hold "simple" instructions, and an additional slice for the "failable" instructions.
The advantage of this setup is that you get to share some of the indexing and control across all slices of the structure; the disadvantage is that the number of items in one slice is tightly linked to (a multiple of) the items in another slice.

So for the Fetch Predictor, I think we have this sort of slice structure. Each row in the Fetch Predictor has
- a first half that holds "Fetch Groups that will end in a branch", so that entry includes a field for "target address" and another field for "number of instructions to Fetch";
- the second half of the row holds a field for "something about the branch structure in the next fetch group or two or three", but doesn't need either the target address field or the fetch width field.
Remember both halves have a separate tag, so even though they appear in the same row, they are unrelated; they just happen to both have the same lower bits in the PC; there is no implication that once a Fetch has used the upper (or lower) half of a particular row in the predictor table, it switches to using the other half of that row.

We will see yet another version of this slice structure when we look at the Branch Information Tables in detail.

To try to summarize some of the above.
We described the idea of a Next Fetch Predictor, and showed
- how it could be bootstrapped from nothing by being continually retrained as its predictions (starting with "always go forward") were corrected one after the other.
- how the sooner these corrections were provided (so, if possible, in Decode rather than at Execute or, even worse, Retire) the better

Even so, it takes up time and a constant stream of mistakes to populate the Next Fetch Predictor.

Can we improve this? That's what (2016) https://patents.google.com/patent/US10747539B1 *Scan-on-fill next fetch target prediction* does.
Suppose that the Next Fetch Predictor jumps to some piece of code that it has never encountered before. This cache line, once it is pulled into the Fetch unit, will be scanned to construct a row in the "Scan on Fill" predictor. Essentially this predictor holds, for a few cache lines, some details that are relevant to Fetch, like where the branches are, the type of branch, and (if possible) the branch target. Now next time a Next Fetch Address is looked up, the lookup will occur in both the Next Fetch Predictor and in this Scan on Fill Predictor. If there's a hit in the Scan on Fill Predictor but not in the standard Next Fetch Predictor, then the Scan on Fill Predictor data will be used. This data will make its way to the standard Next Fetch Predictor and eventually this line of the Scan on Fill Predictor can be removed.

Even this, as I have described it, is slightly reactive. So the Scan on Fill Predictor takes one extra step. Suppose a line is moved into the I-cache but not by Fetch (ie the prefetcher has pushed it into the I-cache). Then when the I-cache has a cycle free, the Scan on Fill Predictor pulls in that line and builds an entry for it. Assuming prefetch is working optimally, this means that by the time Fetch will be ready to generate an access to the line, not only will the line be in cache, but it will also be sliced up by the Scan on Fill Predictor into Fetch Group units.
This mechanism can obviously handle, as embedded in a newly encountered cache line
- direct calls
- returns
- indirect calls (by the "Halt Fetch when it encounters and unknown indirect call, until the target is resolved" method we've already seen).
- what about conditional branches? The patent does not tell us. It suggests these are always either considered taken or not taken. Another alternative might be to probe the Branch Predictor to see if it has a suggestion. A full probe might takes too long to be feasible, but what about a simple probe of the base Branch Direction Prediction table, the one that's just a local 2-bit counter?

The most recent version of the NFP is described in (2017) https://patents.google.com/patent/US10445102B1 *Next fetch prediction return table*.
The following strands are unified:

- entries are defined by a few different "strength" bits.
One of these is a hysteresis bit which tells the system to hold onto the entry even in the face of an early misprediction.
A second such is confidence bits that indicate the target address of the entry (ie where to jump next at the end of this Fetch Group) is trustworthy. If this goes down, we may retain the entry but change the target (eg a virtual call that ends the Fetch Group has changed its target).
Another way to think of this (different from the way Apple describes it, but I think more helpful) is to consider that we have say a 2-bit confidence field, but the field is initialized not at 0 but some higher value. Depending on the initial value we set (1, 2, 3) the entry gets up to three chances at early misprediction before it's considered hopeless and is a prime candidate for replacement.

- now that we are tagging entries not just by the PC but also by a few history bits (as we said, now allowing for a few variant paths to this Fetch Group) two-way set associative is not always enough. Rather than accepting this, we define an "overflow table". If both entries for a particular index in the primary Next Fetch Predictor table are valid and have high strength, then we allocate this additional entry into this overflow table (so essentially a victim cache, though for reasons I cannot see, the patent calls this the NFP Fast Table).

- additionally we realize that any NFP entry that terminates in a Return is wasting the storage used for the Target (the PC to go to after this Fetch Group is loaded). Hence those entries are moved to a separate table called the NFP Return table, relieving some pressure on the primary NFP Table.
(This table can also omit the Fetch Width value because that can't be exploited – returns to different locations will result in the next Fetch Group having different widths, so best one can do is pull in everything from the cache line and splice it out as appropriate once it reaches the Instruction Queue. This seems like another area amenable to a small tweak, like calculating the post-return width at call time, and storing it on the RAS. This is expensive to do dynamically, but could be done at the time that a Fetch Group that terminates in a call is inserted into the NFP. A second reason to have a distinct table or two for NFP entries that terminate in calls, as I suggest below – such tables could have an additional "post-return fetch width" field...)
This separate table for Return-terminated Fetch Groups can be appropriately sized (and probably created as direct-mapped rather than 2-way associative).

- So in any given cycle, what we will see is the PC presented to
+ the primary NFP (two-way set associative, also matching recent history)
+ the Return NFP (probably direct-mapped, no history tagging)
+ the victim cache Fast NFP (FIFO of recent overflow entries)
+ the Scan on Fill Predictor (some sort of cache)
Each of these will try to provide a best guess as to the address of the next Fetch Group.
One could imagine further subdivision over time. For example, perhaps direct calls usually don't need the history tag and could be moved to their own, direct-mapped, table with a smaller tag? Likewise for indirect calls, only their table is 2- or 4-way mapped, with longer history tags. Not trying to compete with ITTAGE, but to at least pick up the lowest lying fruit of virtual calls with an easy prediction pattern...

Now let's talk Loop Buffers.
It should be clear from what we've discussed, that there's no real *performance* need for a Loop Buffer on the M1. A Loop Buffer (and related concepts) is a way to avoid paying the cost of the backward branch of a loop; but M1 already has that cost down to zero. Even so M1 incorporates a variety of loop buffers, mainly, but not exclusively to save energy. The reason the different variants exist is because the most aggressive energy-saving techniques only work with the simplest loops, but one doesn't have to give up on more complex loops, one just has to accept a lower energy reduction.

We start with the basic idea, to use a loop buffer for "easy" loops. The characteristic of an easy loop is

that there is no variant control flow within the loop body – we enter at the top, we loop back at the bottom. In the strictest version we could insist on no branches in the body.
Such a loop (if it's short enough, which it usually is) can be captured within a buffer in the Fetch Unit right next to Decode. We can run the loop out of the buffer and avoid the energy costs of branch prediction and the I-cache.
This is covered by (2012) https://patents.google.com/patent/US9557999B2 *Loop buffer learning*, and the main trickiness is detecting such a loop (and recording that other potential such loops should be ignored). However we also see a few additional optimizations:
- The loop buffer lives behind Decode. This means the buffer holds uops rather than ops, and the cost of Decode can also be avoided!
- The loop buffer does allow for a limited number of forward taken conditional branches (ie simple `if(condition){}` clauses within a loop), but no indirect branches. The patent is somewhat ambiguous as to whether simple call+return would be allowed.
The loop body has to be *invariant*, ie the same from iteration to iteration, so conditional loops are allowed – but only if they don't change their taken/not taken status.
The number of branches allowed is surprisingly generous, 8 are suggested in the patent. In other words, even at this simple stage we're capturing loops a lot more sophisticated than a basic blitter or strcmp.

Once we have a loop buffer, what more can we do with it? Well, one minor flaw in the Fetch scheme we have described so far is that only one Fetch Group (ie run of instructions before a branch) is acquired per cycle. This is not ideal if someone writes a really tight loop with the loop body as, say, 3 instructions – now the maximum speed at which we can run is 3 instructions per cycle/regardless of any other details. But suppose we could unroll that loop in the loop buffer... Unroll it three times and we at least have the possibility of running 8-wide per cycle; unroll it more than three times and we may be able to engage in even better optimizations like running multiple dependency chains in parallel.
This patent also provides insight into the size of the loop buffer suggesting (at least as of 2012) that it was arranged as 16 rows of 6 slots (6 slots because decode for that generation of A7-class CPUs had 6-wide decode). To compare, the (somewhat equivalent) structure in Skylake, the IDQ, can hold 64 µOps. (The IDQ is statically partitioned in 2x64 halves, but I'm only interested in single-thread behavior. The point is also somewhat moot because a bug in Skylake means streaming loops out of the LSQ was disabled via microcode update in 2017.)
This is the content of (2012) https://patents.google.com/patent/US9753733B2 *Methods, apparatus, and processors for packing multiple iterations of loop in a loop buffer.* This is of particular interest to Apple because they recommend all code (unless there's a good reason otherwise) be compiled as -Os which, among other things, will not unroll loops that clearly look like they would substantially benefit from unrolling.

But the basic loop buffer is limited to easy loops. Next up in complexity, suppose we have a loop that involves control flow, but nothing hard that needs to be predicted. The most obvious case is a loop that calls a simple function. We do need to predict the return of the function, but that's not hard. This sort of thing can be captured by a

. Depending on the exact design, we may be able to capture the control flow as a Trace in the Loop Buffer, or we may farm it out to a very simple Trace Cache. Either way, we still avoid most of the costs of Fetch.

Finally we have loops that involves some degree of branching. Neither a Loop Buffer or a Trace Cache is a good match for such code, but we can move the loop into an L0 cache that uses lower energy, and we may be able to clock down some structures (eg RAS or Indirect Branch Prediction) for the duration of the entire loop, once we see what is and is not in the loop body. We could even assume (or perhaps test, and validate) that the branches in the loop can be handled by a simple (few entries, just a local 2-bit counter) L0 branch predictor tied to the 0 cache.
(This brings up another issue which is interesting, but I've rarely seen discussed – how to treat loops when training branch predictors. Simply mechanically advancing the history vectors on every iteration of the branch back at the bottom of a loop, or even on a simple branch within a loop, is mainly a good way to remove all useful data from your history vectors... On the other hand one probably wants at least one iteration of the loop to be captured by the history vector, to provide some context of the occurrence of that loop into the history.
So all these different energy saving tricks, even if they were not designed with better prediction in mind, probably help towards that goal! If two or three iterations of a loop go through the normal branch training procedure, and then a specialized loop mechanism takes over which, among other things, powers down branch training and continually updating the branch history vector, that's probably the ideal outcome even if it wasn't explicitly designed that way.)

(2014) https://patents.google.com/patent/US20150205725A1 *Cache for patterns of instructions*, discusses how and why Apple uses an L0 cache (with associated L0 local branch predictor). The patent also refers to a Trace Cache but describes it in only the most minimal detail, so make of that what you will.
The patent also describes how more complex loops (including even nested loops, as long as they fit in the loop buffer/trace cache/L0) are tracked and handled. If that interests you, *Instruction loop buffer with tiered power savings* (to be discussed below in a different context) gives a few more different examples of nested loop patterns that can be detected.

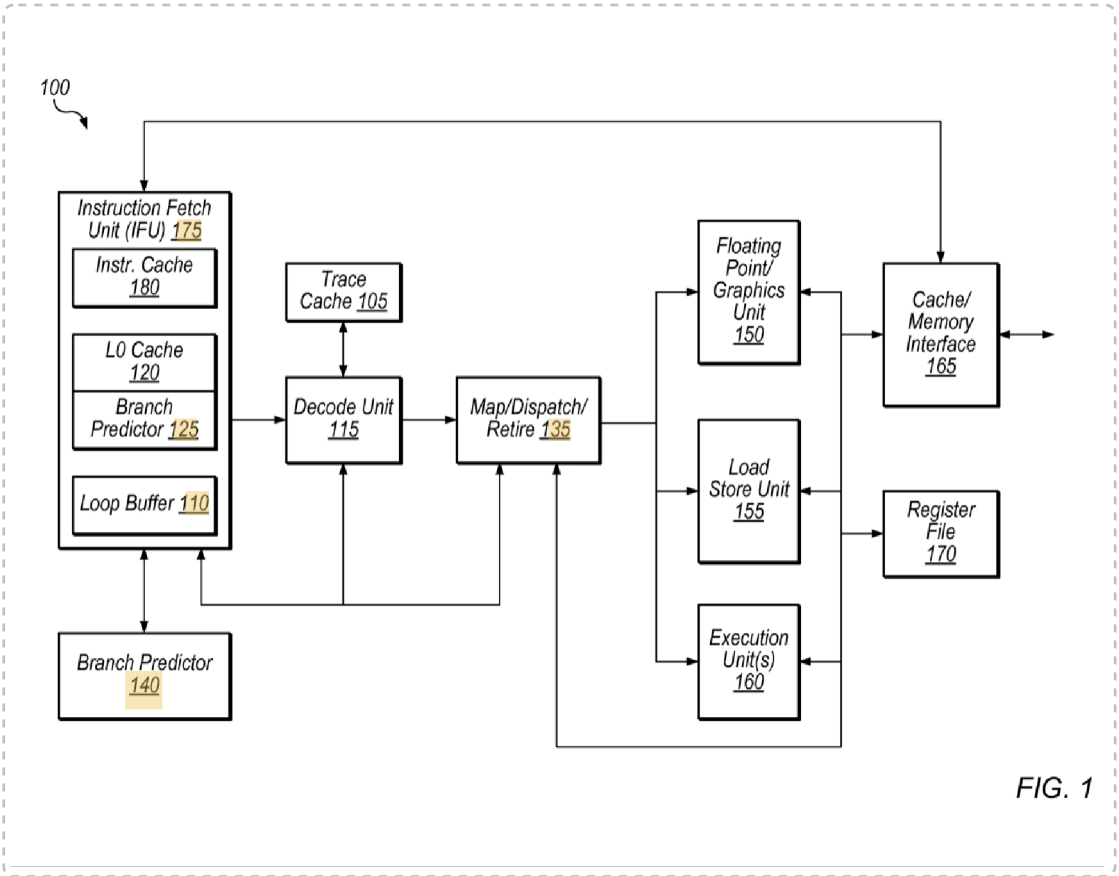


FIG. 1

An issue with the loop buffer is how rapidly do we switch from (higher energy) traditional decoding to a lower-energy loop buffer mode. Something I don't understand is that none of these patents suggest storing what I'd consider obvious, namely a table of "known loops" storing things like the loop start+-size, "number of iterations" vs "unpredictable exit", "types of branches encountered in the loop", etc. With a table like that, you could, much of the time, capture loops into a lower-energy mode every time after the first encounter.

The concern (at the time of the patents, anyway) appears to be that prior iterations of the loop will affect the branch prediction in later loops, and so one wants to iterate the loop enough times to "saturate" the predictors. I've already mentioned that this seems like a terrible idea once you have very long (hundreds of branches) TAGE-like predictors, and I also have to wonder, for most loops, how much useful prediction information is created by repeated iterations of the loop after the first two or three. My suspicion is that some of these details have substantially changed by now.

Regardless, given a loop buffer, another dimension for improvement is reducing energy. (2014) https://patents.google.com/patent/US9471322B2 *Early loop buffer mode entry upon number of mispredictions of exit condition exceeding threshold* provides a slight tweak to the 2012 design. One of the items that is tracked by the table tracking candidate loops is whether the loop exit is unpredictable (think for example of something like strcmp). An unpredictable loop exit isn't great, but whatever you do, there is going to be a mispredict at loop exit. Given this, the logic is you might as well at least still

run the loop at lower energy out of the loop buffer for as long as possible, and give up on trying to train the branch predictor on details of the loop.

Meanwhile (2014) https://patents.google.com/patent/US9524011B2 *Instruction loop buffer with tiered power savings*, attacks the problem from a different direction, rapidly shutting down some elements of the machinery that feed a loop, then gradually shutting down more elements as confidence in the loop's persistence grows. The first shut-down is of the Fetch front-end; the neat conceptual breakthrough here is that the loop is already present in the queue between Fetch and Decode (I described above, when talking about Asynchronous Fetch, why you want such a queue and why you want it to be fairly large). If the loop is present there, then we can at least stop bringing it in from the I-cache every iteration, and so, even though we continue to power-on the high quality branch prediction, we can sleep the Fetch Predictor and all access to the I-cache. This is done as soon as two or three loop iterations are detected. Then after more iterations, we may feel confident enough to shut down branch prediction (and presumably also copy the loop to the Loop Buffer holding uOps and placed after Decode).

At this point we should discuss the Branch Information Table. We mentioned this when we discussed the RAS.
The patent discussing this is (2016) https://patents.google.com/patent/US10175982B1 *Storing taken branch information.*
The problem to be solved is that we want to store all branch information in a tentative state until the branch retires, at which point the state can be used to train predictors. As an additional design criterion, insofar as possible, we want predicted non-taken branches to be as close to a NOP as possible; ideally they do not take up space in the predictor tables, so that when no prediction is found we can just predict "branch not taken". So if a predicted taken branch is not taken, we need to inform the predictor to update. But if a not-taken branch is encountered, I think we don't bother training on it, we're happy to leave it out of the tables forever if it's always non-taken.
This means
- we want a structure that's like the ROB, a large circular queue into which every branch is inserted at one end (the write pointer)
- there's also a retire pointer; all branches between write and retire are pending in the machine
- there's also a training pointer; all branches between training and retire have retired but not yet been fully incorporated into training
- so the layout looks like (write ..... retire .... training)
- to reduce area, there are two tables, a Branch Information Table and a Taken Branch Information Table
- BIT holds information relevant to all branches, TBIT holds additional information relevant to taken branches
- a BIT slot is allocated at Decode, as is a TBIT slot if the branch will be taken

- but what if the branch is predicted not taken, but that's incorrect? Then at the point of branch execution (where the incorrect prediction is discovered) a TBIT slot will be allocated
- but, something very weird, in the example given in the patent, they suggest the BIT holds 60 entries, while the TBIT holds 96 entries. Why make the TBIT (which is supposed to be a subset of the BIT, isn't it) larger? I think what is going on is that while the BIT operates like a simple circular queue as described, entries in the TBIT are opportunistically allocated and deallocated. In particular I think that at or shortly after Retire "ownership" of a TBIT entry is handed over to Branch Predictor Training which holds onto it for a few cycles doing whatever training needs to do. Thus at any given time, up to 60 pending execution branches may be present in the BIT, along with up to (but probably few less than) 60 associated pending taken branches; along with an additional up to 36 records of taken branches that have retired but their branch information has not yet been integrated into the Predictors.
(Or the patent simply got these numbers backwards?)

- with all this now in mind, most of the fields in the BIT and TBIT make sense. For some of the less obvious ones:
+ a few of the low order bits of every branch address are recorded in every branch, taken or non-taken, conditional or not. I think these are used to update the branch history vectors at the point the branch is Retired
+ for branches that will be predicted (and thus will be looked up in a predictor) we also need some of the higher address bits because they will form the tag to check that the value looked up in the predictor actually is the branch we want (ie check that aliasing has not occurred)
+ for all taken branches we record the branch history vectors at the point the branch executed. We want these to restore branch history state if we need to recover from a mispredict at this branch.
+ the remaining fields are related to the precise details of how either the Indirect Branch Predictor or the Call Stack are updated. Some look familiar'ish, some not. They suggest that the Indirect Branch Predictor uses elements of TAGE (like multiple tables and U bits), but also unfamiliar elements like "BTP hit table". Likewise the RAS pop pointer field is familiar (recover the RAS when additional elements have been pushed on top) but the "RAS branch" stuff only suggests to me that Apple are doing something different from Intel in how they prevent return addresses from being overwritten by speculative code.

- as mentioned, I think the slice stuff means eg one branch per predicted branch (alternate unconditional and conditional; also call and ret as "unconditional"
and maybe 1 target, 2 "all branches", 1 call/return in the BTIC?

need to run lots of tests here, especially with pairs and triplets of different types of types of branches, to figure out the exact details

We have talked glibly about Fetch feeding an Instruction Queue which then feeds Decode.
Naturally the details of this become ever more complicated the closer you look. For example in any given cycle, Fetch has to pull in some number of instructions from up to two cache lines, and those

cache lines can in fact be in any of an actual cache line, or a prefetch buffer, or delivered on demand from L2 or higher. Beyond that, if the Instruction Queue is empty, then the Fetch'ed instructions should be delivered directly to Decode without buffering and, in a worst case, some of the instruction to be delivered to Decode will come from the Instruction Queue, some will come from Fetch, while the remainder from Fetch will go into the Instruction Queue.

This sounds like nightmare. Fortunately we have a patent (from 2016, but it's clearly describing the A6, so it's much simpler than any modern implementation). The patent shows us both the naive solution, and a somewhat neater solution.

In[3]:=

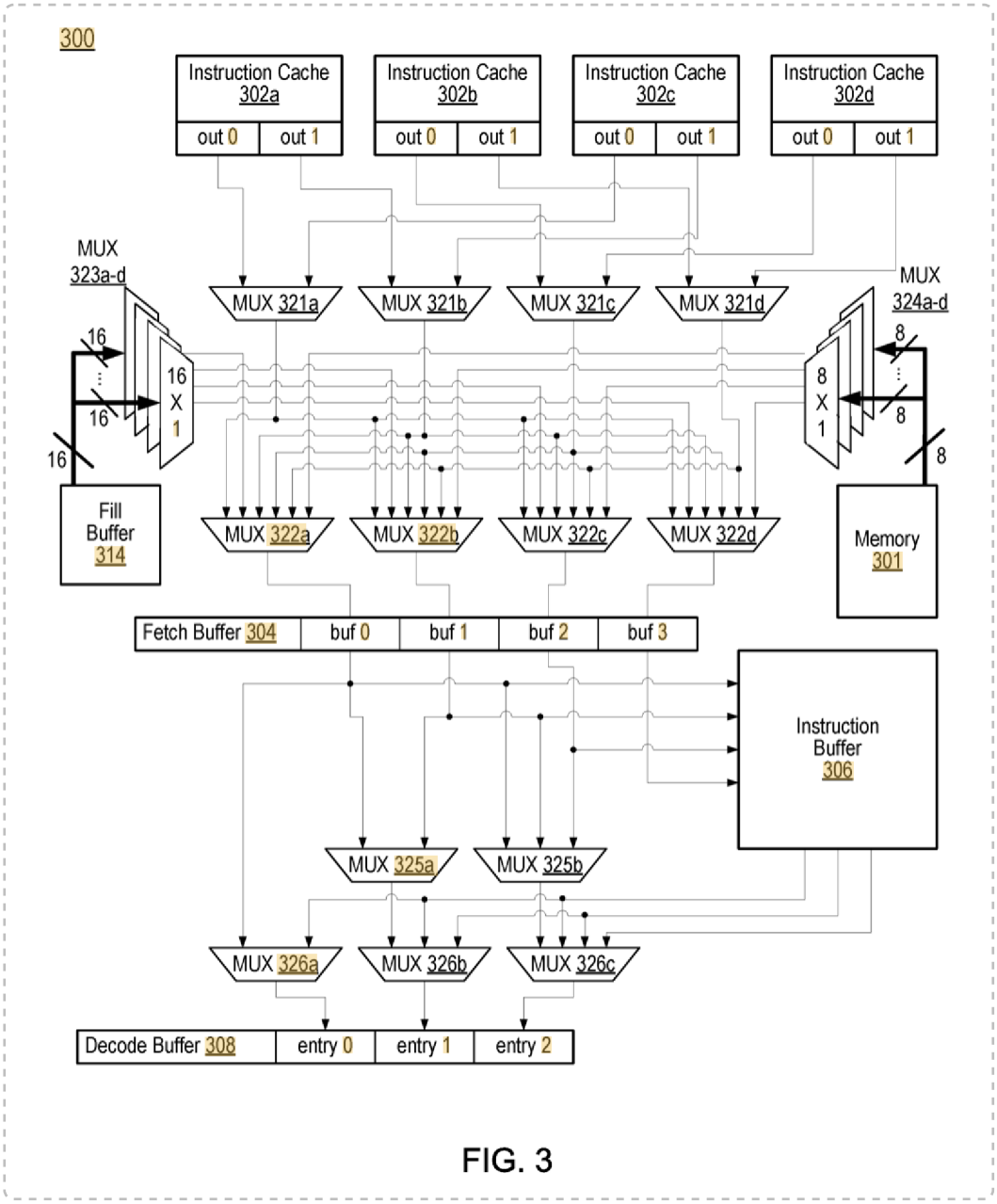


FIG. 3

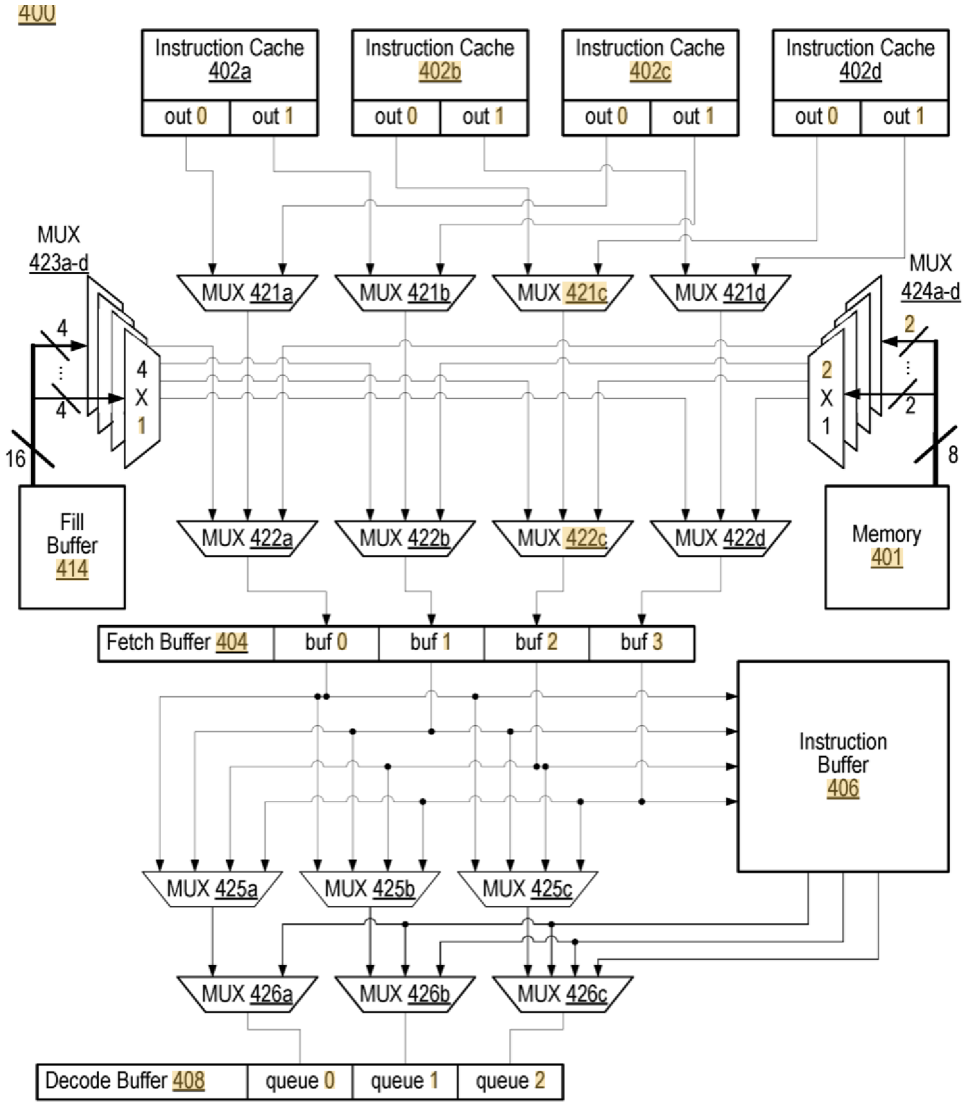
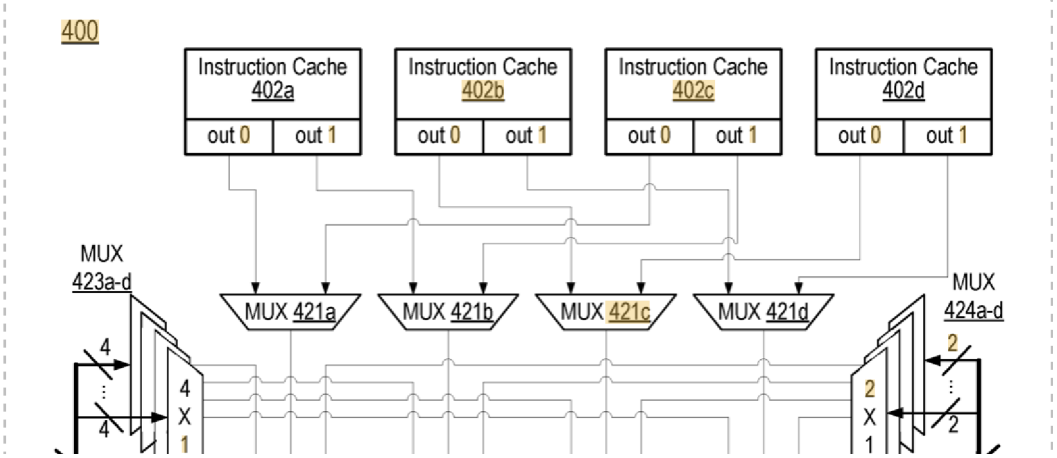

400
Instruction Cache 402a
Instruction Cache 402b
Instruction Cache 402c
Instruction Cache 402d
out 0
out 1
MUX 423a-d
MUX 424a-d
MUX 421a
MUX 421b
MUX 421c
MUX 421d
4
4 X 1
2
2 X 1
16
8
Fill Buffer 414
Memory 401
MUX 422a
MUX 422b
MUX 422c
MUX 422d
Fetch Buffer 404
buf 0
buf 1
buf 2
buf 3
Instruction Buffer 406
MUX 425a
MUX 425b
MUX 425c
MUX 426a
MUX 426b
MUX 426c
Decode Buffer 408
queue 0
queue 1
queue 2

FIG. 4


400
Instruction Cache 402a
Instruction Cache 402b
Instruction Cache 402c
Instruction Cache 402d
out 0
out 1
MUX 423a-d
MUX 424a-d
MUX 421a
MUX 421b
MUX 421c
MUX 421d
4
4 X 1
2
2 X 1

Out[3]=

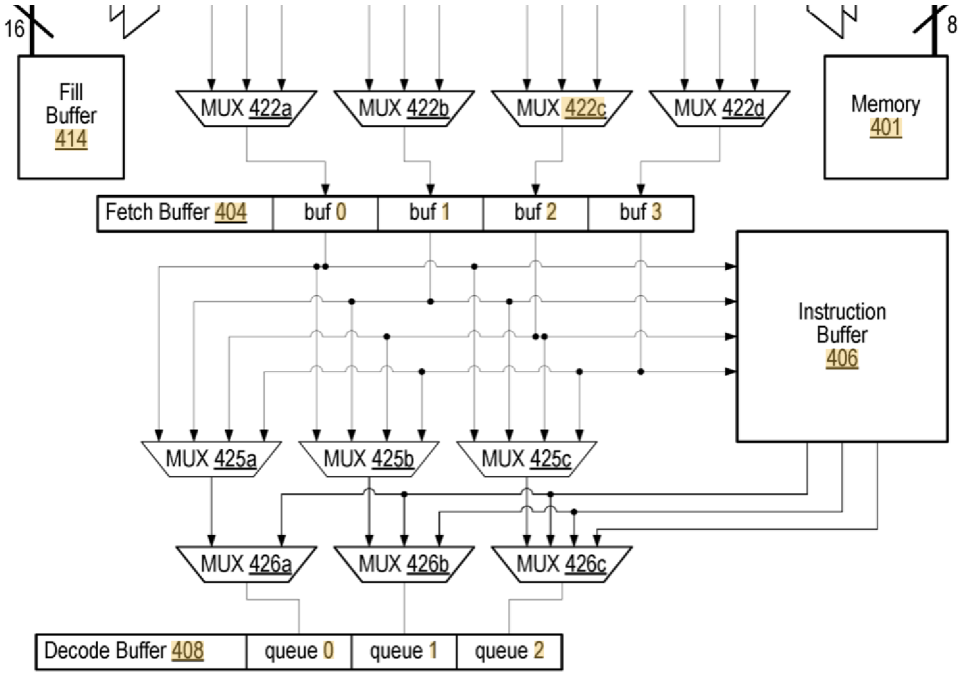
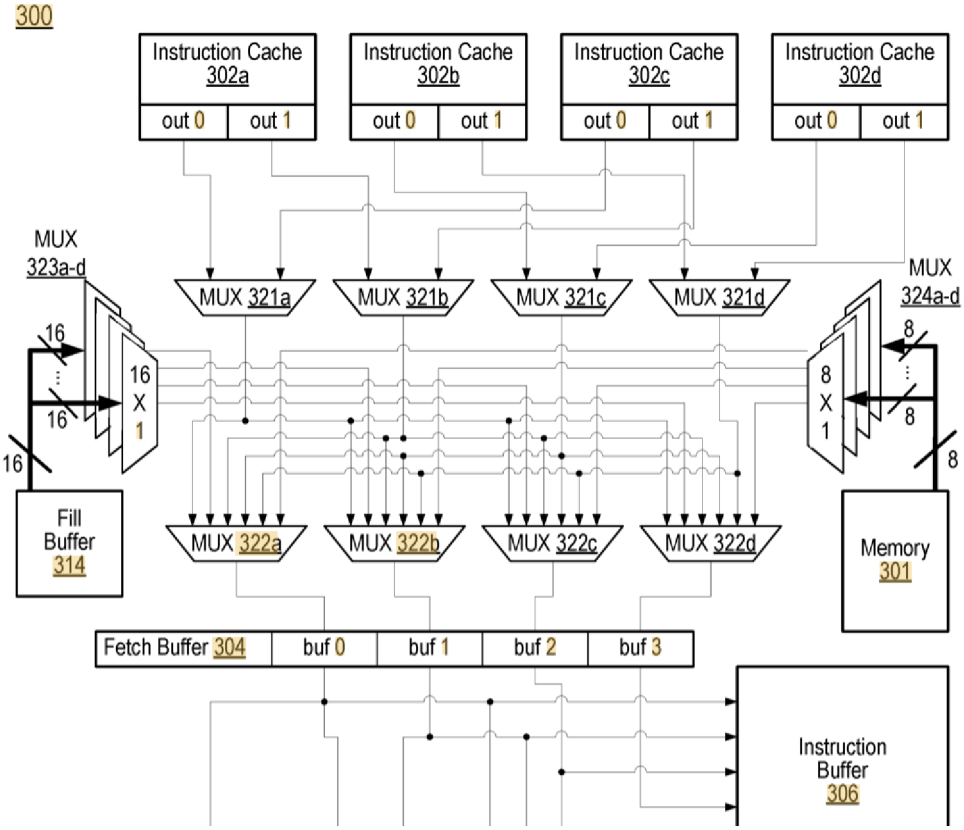

16
8
Fill Buffer 414
MUX 422a
MUX 422b
MUX 422c
MUX 422d
Memory 401
Fetch Buffer 404
buf 0
buf 1
buf 2
buf 3
Instruction Buffer 406
MUX 425a
MUX 425b
MUX 425c
MUX 426a
MUX 426b
MUX 426c
Decode Buffer 408
queue 0
queue 1
queue 2

FIG. 4


300
Instruction Cache 302a
Instruction Cache 302b
Instruction Cache 302c
Instruction Cache 302d
out 0
out 1
MUX 323a-d
MUX 321a
MUX 321b
MUX 321c
MUX 321d
MUX 324a-d
16
16 X 1
8
8 X 1
Fill Buffer 314
MUX 322a
MUX 322b
MUX 322c
MUX 322d
Memory 301
Fetch Buffer 304
buf 0
buf 1
buf 2
buf 3
Instruction Buffer 306

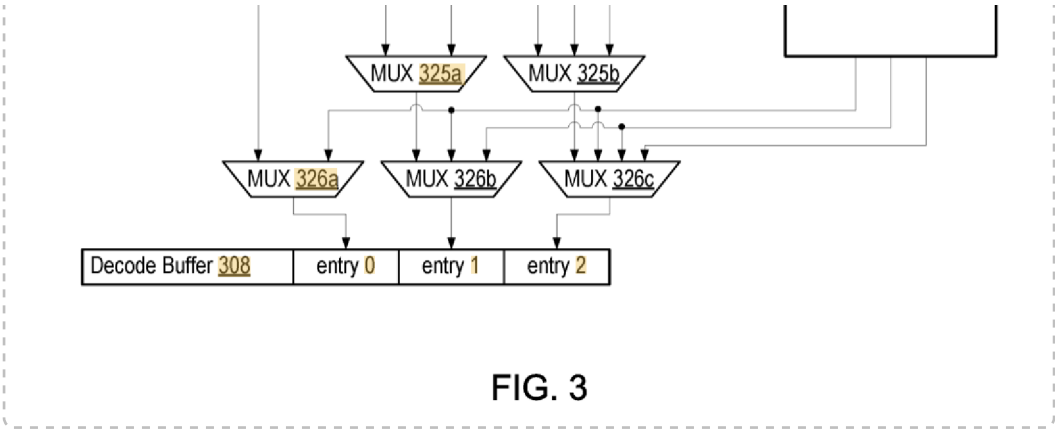


FIG. 3

You probably don't want to puzzle through these in full detail, unless you;re really dedicated, but Figure 3 is the naive solution.

The Fill Buffer is the Prefetch Buffer, so we have that eg the 16 instructions are replicated to 4 16-wide muxes. Each of these buffers chooses up to one instruction which gets fed to the 322a..d muxes. Likewise each sequential pair of instructions delivered by the four instruction cache subarrays, and likewise for memory. The point of these complicated nested muxes is not just to choose the next four instructions we want to Fetch, possibly from more than one of these sources; it is *also* to store the instructions in the correct order in the Fetch Buffer.

Note also how the Fetch Buffer then feeds into the Instruction Buffer and the Decode Buffer so that, once again with a correct selection of controls to the 325 and 326 muxes we can pull some number of (appropriately ordered) instructions from both Fetch and Instruction Buffers, while also dumping other instructions into the Instruction Buffer.

The insight of the patent is that no-one actually cares how the data is placed in the Fetch Buffer. So what Fig 4 does is remove some of the logic from Figure 3. The instructions are still placed sequentially in the Fetch Buffer, but treating it as a circular buffer, ie the sequence of instructions can be placed at any location.

This allows a substantial reduction in the number and complexity of the muxes above the Fetch Buffer, while only requiring a fairly simple rewiring and slight increase in the complexity of the muxes 425a..c (and a similar slight modification to the queueing logic in Instruction Buffer 406).

Of course even before Fetch, we want an I-prefetcher to be ensuring that, as much as possible, every I-line that Fetch touches is already in the I-cache.

I-prefetch is, in a sense, even more important than data prefetch because the machine can do less work while it is waiting for an I-cache miss. It can work through the instructions in the Instruction Queue but that might, best case, cover fifteen cycles or so; enough to get to L2 but no further.

The first helpful technique, not even really a prefetch, is to slightly prioritize I-lines in the L2 cache over D-lines. One can imagine a few ways to do this, and I expect Apple is doing so, but this is well known and probably nothing patentable.

Next up is simply ensuring that, like the branch predictors, the I-prefetcher is not tainted with incorrect data. Obviously this means training it with a sequence of PC's generated at Retire, not the (admittedly

more easily available) sequence of PC's available at Fetch. But it also means filtering out PC's generated by interrupts and exceptions...

At this point we're at the prefetcher itself.