Instruction Fetch v0.90

what (really!) is Fetch?

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. Compounding the problem, there is not a generally agreed upon set of terms for describing the newer (post 2000 or so) concepts. This means that this is subject that leads to constant fights, but if you find yourself in the middle of such a fight, try to dial things down by getting everyone to explain their terms and draw diagrams. You will frequently that you agree on the underlying points, but disagree on terms!

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, *taken* branches of one form or another. A useful term for these runs of sequential instructions is a *trace*.
As far as traces are concerned, non-taken branches do not matter, they do not affect the sequential flow of instructions.
Traces can be as short as one instruction, and can, in principle, be indefinitely long; but realistically they tend to have a length of around 10 instructions or so; think for example of the size of a loop body (for a simple loop without branches), since each run of the body is a separate trace.

You should now realize that if you hope to run the machine, as much as possible, at a full 8 operations per cycle, you simply cannot think in terms of fetching one or two instructions at a time; you have to think of fetch in terms of traces. Fetching sequential instructions is easy, handling the breaks between traces is hard.
More precisely, our hierarchy of tasks is something like:
- each cycle we compute a fetch address for the next cycle.
- we then pull in the instructions associated with that trace, and jump to the next trace.
Even more precisely
- at least initially, assume an I-cache line is 16 instructions (64B/4B), so we usually be able to get a good

number of instructions from a single cache line (so initially our I-cache may be single-ported?)
- so each cycle we want to actually predict two values (next trace address and next trace length)
- the next cycle we load in the smaller of (trace length, length of instructions remaining in the cache line, maximum fetch width [which may be, say, 16 instructions])
- the cycle after that we continue as above until we have loaded in the entire trace
- then we jump to the next trace.

Once you understand all this you will understand a few points: (given in summary here, then explained in a lot more detail in later sections)
- you want a deep queue between Fetch and Decode because you want the times when Fetch goes well (a long trace, perfectly cache aligned, so you load more than eight instructions in one cycle) to build up a good buffer for all the times when Fetch goes badly (short traces of just a few instructions, traces that start right at the end of a cache line, so you only get to load a few instructions from that cache line; let alone when we miss in the I-cache and have to burn 15 cycles or so going to L2).
- the next step in boosting this system is to make the I cache double-ported (via multiple banks) so that we can fetch traces that run past the end of a cache line. In fact Apple was already Fetching traces that crossed two I-cache lines as early as 2011.
- the next step after that (rather more complicated, as we will see) is to predict the next two traces (start+length) so that two separate traces can be fetched in the next cycle. As far as I know Apple hasn't had to move onto that next stage yet.

- if we're not going to lose half our performance in Fetch, we have to essentially predict the next trace every cycle. On many platforms this is the job of a structure called the BTB (Branch Target Buffer). Apple has something equivalent to the BTB (the primary Fetch Predictor) but this is augmented with at least multiple additional Fetch Predictors.

Importantly Fetch Prediction and the BTB is far from the end of the story.
Because the BTB has to operate so fast (new prediction every cycle) there is a limit to how large and how sophisticated it can be. But, as long as most of the BTB predictions are correct, we have some opportunity to deal with incorrect predictions without too much drama. The point to realize is that instruction flow is in-order from Fetch, through sitting in the Fetch Queue, through Decode, through Rename, up till Issue. At early points in this in-order flow, we can re-steer the instruction flow (throw away instructions), fairly easily and without affecting the later machine.
As long as instructions are in the Fetch Queue, this is essentially painless (except for some lost energy, and possibly losing a cycle or two).
We can even re-steer instructions in Decode (at Decode ROB slots have been allocated, but they are allocated in-order, and de-allocating them is not too painful). But at Rename all manner of resources have been allocated and at that point it's basically hopeless to drop instructions and re-steer.

What this means, in practical terms, is that we have a sequence of tasks
- [before Fetch] when instructions come into L1 they are scanned and marked as branches (of various

sorts)
- [at Fetch-1] the BTB predicts the next trace
- [at Fetch] the I-cache has to be accessed with the trace info (address, and number of bytes to load)
- loading the instructions will take two or three cycles
- during this period, after the BTB prediction, but before the instruction actually reach Decode, there are a few cycles during which more sophisticated (larger and slower) predictors can make the same prediction as the BTB made (eg the direction of a conditional branch, or the target of an indirect branch). These sophisticated predictors can override the BTB prediction (perhaps after one cycle, perhaps after three cycles) and re-steer, so we only lose a cycle or two or three.
- at the end of Fetch we *might* (unclear) get a second round of checking, to ensure that the trace boundary points actually make sense. (That is, we know where the branches are, and if the predictors had us breaking a trace at a point that is not a branch, then clearly prediction has gone wrong. Recovery may be messy, but at the very least we can stop Fetching and not waste energy until the Back End flushes and tells us where to restart.)
- at Decode we have a final possibility of catching a failure, at least for simple branches (unconditional, known target, like a goto or a direct call) and re-steering from Decode.

So to understand prediction, you need to appreciate that multiple predictors exist, for multiple purposes. The BTB is the first line of defence, and generates most of the successive fetch addresses, but predictors specialized for conditional branches, indirect branches, and loops will kick in and correct if necessary a cycle or three later; and Decode (and perhaps even the Fetch Queue) will catch whatever they can.
We don't want to let a mispredicted branch through all the way to the Out-of-Order machine because that will require a flush, which will directly cost perhaps fifteen cycles or so (and wasted energy), and in fact it's worse than that because the first few cycles after the flush are substantially less productive, because we don't yet have all the queues (Fetch Queue, Scheduling Queues, Load Store Queues, etc) filled, and so much of our parallel machinery to allow finding and processing 8 independent instructions each cycle does not have much to work on.
At this point you may want to look at https://blog.cloudflare.com/branch-predictor/ . I wish it structured its analysis in terms of traces rather than this old-fashioned analysis in terms of single instructions, but it's an easy intro to some actual numbers. (Though the numbers that are easy to measure are not necessarily the numbers that matter much for performance, or understanding...)

two non-trivial observations about Fetch

One observation to bear in mind (which is obvious when you hear it, but was only made very recently) is that, apart from all the other distinctions, there are two types of branches: "local" and "global". Local branches are `if()` statements within a function, or perhaps messy variants of this like the use of indirect branches to handle large `switch()` statements; global branches are jumps from one function to another (or perhaps some of the messier versions of `longjmp` and exception-handling).
For a long time Fetch and Branch Prediction ignored the difference between these two, but much

modern work is based on exploiting this difference. One obvious path is to allow for smaller prediction tables for local branches (since the jump distances are small); but there are second-order differences, like Global jumps are more likely to result in I-cache misses (and to land in a block of code for which we now have no prediction information), so maybe Global jumps should also be hooked up to some sort of L2 branch predictors, which can be loaded into the Fetch unit in parallel with the probable I-cache misses?

A second observation to bear in mind is that Fetch (once you see what's involved) is crazy expensive and complicated because of branch prediction and the fact that all this work has to be done speculatively. But branches serve multiple tasks. Can we not get rid of them for the cases where they don't actually need to fiddle with the flow of instructions?
One version of this is using predication (in ARM64 this could be done, for example, through `CSEL`) rather than minor branches that simply toggle between two instructions or so, things like
`if() a=X else a=Y;`

But that's not the only version. A different type of "structured" branch is the control of loops, obviously counted loops but even many structured (`while` or `until`) non-counted loops.
The thing about loops is that the instruction flow, in a sense, ie the Fetch part of the problem, does not need the full complexities of generic Fetch, and ideally this should be exploited.
One way to do this is via packed SIMD instructions (eg NEON or SVE), where each instruction is, in some sense, a mini-loop. But that's a problematic path, resulting in an exploding number of opcodes. An alternative path, which has not been explored very much in commercial ISAs, is to add some sort of "hardware loop" to the instruction set, enough information that the front-end can decouple the instruction sequencing required for the looping from the generic sequencing requiring the full fetch machinery. An example of this as a suggestion for how to expand POWER, is described in https://ftp.libre-soc.org/simple_v_spec.pdf.
On the Apple side, Apple have a *long* series of patents under the name Macroscalar, beginning in 2005 but going through till at least 2014, an example of which is (2005) https://patents.google.com/patent/US8578358B2 *Macroscalar processor architecture*, best summarized, as far as I can tell, like the POWER proposal as a way to describe a "hardware loop" more explicitly, so that the loop can execute repeatedly without involving the front end.

For the most part what we actually see in current hardware is the sub-optimal solution of trying to detect loops in hardware as a particular type of instruction pattern, and then treat the loop body specially, eg by sequencing the instructions repeatedly out of a loop buffer rather than via the full Fetch machinery. This works to some extent, but seems sub-optimal.
For example this sort of machinery still can't do a great job (and has to "learn" what it is doing for each loop) in handling the issues we have already described:
A "normal" loop consists of local instruction sequencing. Even if it incorporates branches (eg to do one thing when a value is even, another when the value is odd) these branches tend to be small displacements. It's likely that they are better handled either as predicates or some sort of "speculated predi-

cates" that tries to execute the speculated path at high speed, but with enough state saved that a misspeculation can be treated as a low-cost Replay rather than a high-cost Flush; and the loopback to rerun the instructions(even for `while` and `until` loops) is probably better handled as a special case, not as generic Fetch.

decoupled Fetch

Now the *important* point of all the above 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 accurate 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 in 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 trace being fetched is not very long).
To make up for this, when long traces are encountered, Fetch should be capable of fully exploiting them, moving sixteen instructions or more from the I-cache to the Instruction Queue.

How large is the Instruction Queue? I've no idea! Claims on the internet are that Zen2's Instruction Queue is 20×16B which we can consider to be about 80 instructions, and that Sunny Cove's Instruction Queue is ~50 instructions. I'd guess Apple try for at least 8×15=120 instructions, to cover 15 cycles or so of latency in the event of an I-cache miss that hits in L2.
(Note the obvious fact that such a Decoupled Fetch architecture, if connected to a deep enough Fetch Queue, can act as somewhat of an I Prefetcher. Some of the papers we will later refer to are based on this observation, modifying/optimizing the Prefetch Predictor to also act as the L1I prefetcher.)

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.

It appears that this sort of design was first proposed by Glenn Reinman in (2001) https://cseweb.ucsd.edu/~calder/papers/UCSD-CS2001-676.pdf *Hardware Optimizations Enabled by a Decoupled Fetch*

Architecture.

“fully” decoupled Fetch

We can in fact, improve on this! What we have described so far consists of a Fetch Engine connected via an Instruction Queue to Decode and the rest of the CPU. Ideally when one or other of these two is forced to pause, the other can keep going via the queue.

But what we have described as the Fetch Engine has two parts.
One part is the generation of a stream of Fetch addresses (plus widths);
the second part is actually loading the instructions from cache.
We can split these into two machines, a Fetch Address machine and a Cache Access machine, connected via a queue that holds a stream of (Fetch Address+width)!
So the design now looks like
[Address Generation] → address queue →
[I Cache Access] → instruction queue →
[Decode] → [rest of the CPU]

Once again the hope is that if some part of the machinery loses a cycle or two the other part can keep going via the queue. This idea is known as FDP (Fetch Directed Prefetch) or FDIP (Fetch Directed Instruction Prefetch) but it actually wins on two dimensions.

The obvious win is that if Cache Access misses in the L1I, we can keep generating Fetch Addresses for the Address Queue (as well as continuing to Decode out of the Instruction Queue) and so this all acts as something of an L1I prefetch scheme (in that neither Address Generation nor Decode are slowed down by an I-cache miss). On understanding this, we can devote area that would have been used for I1-prefetch to better Fetch Prediction and come out ahead on both area and performance.

But if you think about it, we could have achieved the same thing (I Prefetch) just by having an even deeper Fetch Queue.

What we get with this second decoupling and second queue is
- each entry in the Fetch Queue (one address+width) is smaller than the equivalent trace entry (perhaps up to 16 instructions) in the Instruction Queue, so we save some area/energy with the same degree of “runahead”.

- we now have scope to allow Fetch Prediction (occasionally, not often!) to take two cycles rather than one. Meaning we can have a 1 cycle Fetch Predictor coupled to a 2-cycle L2 Fetch Predictor. Representative numbers in the literature are L1 Fetch Predictor holds 128 entries, L2 Fetch Predictor holds 8192 entries. The hope is that if we build up a pool of entries in the Fetch Queue (because occasionally the L1I will miss, and we'll keep generating Fetch addresses over the 15 cycles or so to go out to L2), then we can occasionally miss in the L1 BTB and lose a cycle of Fetch Prediction to be made up

by the queue.

- we can easily convert this system into a more aggressive Prefetch system. Right now the Prefetch is happening as a side effect of the Fetch (process each address in the Fetch Queue in-order, and store the instructions loaded from the cache into the Instruction Queue) occasionally missing in L1.
But suppose we create a second machine that scans the Fetch Queue and compares the addresses against the L1I tags. This can be done in parallel with the main Fetch as long as the tags are in sufficiently many independent banks (you'd expect the 192KiB L1I probably uses at least 6 independent banks). This second machine can scan addresses as rapidly as possible (perhaps four per cycle?) and can generate prefetch requests to the L2 independent of (and in advance of) the main Fetch machinery.

- if we want to be aggressive, we can do even better. The idea is to load in as prefetch not just each new line that's placed in the Fetch Queue, but some number of lines near that line (perhaps six lines ahead and two behind). This removes a few cycles from the prefetching of lines that will probably we useful.
Of course there are details necessary to make such a system work well (you want a lightweight filter to remove the necessity for most of the tag checks which are mostly lines that have already been checked and are already present in cache) but overall this sort of far in advance BTB Fetch Address prediction along with a smart Next Line prefetch can be made to work very well.
The main, very different, idea that (on paper) works even better is Temporal Replay schemes, but these require truly massive amounts of prefetch metadata, enough that you need to start carving out dedicated areas in L3 to hold it all; and it's unclear that for general purpose machines this is a good investment (for dedicated database machines with large and constantly varying instruction footprints it may be, but it's a lot of work and a lot of design changes, still very much an academic experiment not an industrialized product).

- this all sounds realy good, but we can possibly do even better!
Remember that we have Fetch Prediction generating a stream of addresses. These exact addresses (not prefetch guesses based on them) can be queued up as multiple successive addresses all fired off to the L2 and serviced in parallel or in rapid succession.
Thus we can achieve a good degree of Memory Level Parallelism for instruction fetch.

This is as opposed to when Fetch Predict immediately becomes an ICache access, with no decoupling queue between them; in that case only one Fetch address is known, until the ICache access returns and the next Fetch Address is predicted, and there is no opportunity for any sort of Memory Level Parallelism. by firing off multiple I-requests to L2.

so what's actually implemented?

Apple very clearly are using Decoupled Fetch and deriving some value from the Prefetch that you get for free in this case (like pretty much every modern CPU).

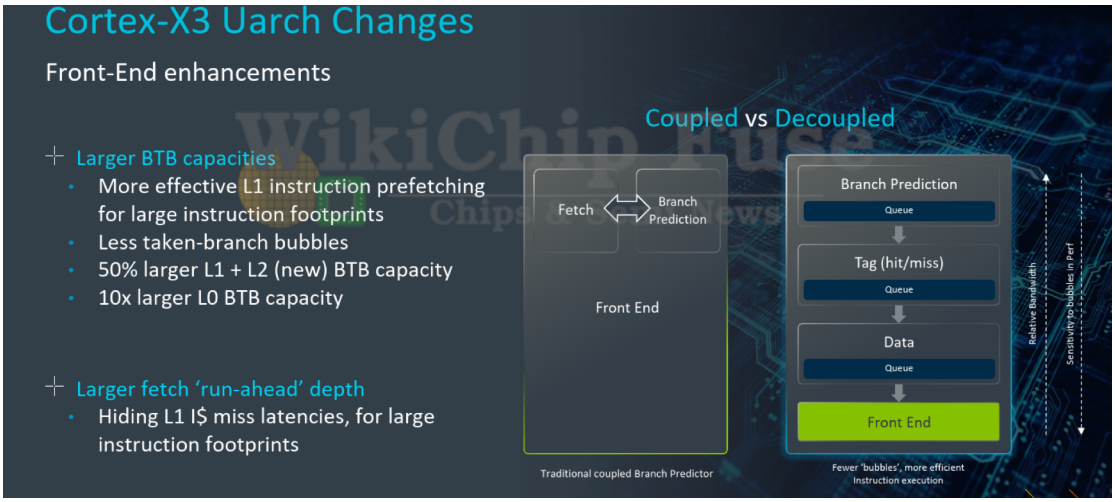
It's unclear whether they have moved to the next stage of Fully Decoupled Fetch Prediction, ie generating a stream of Fetch Address predictions decoupled via a non-trivial queue from Cache Access.

Now, Fully Decoupled Fetch sounds even better, doesn't it? What's the catch? Why might Apple not be using it?

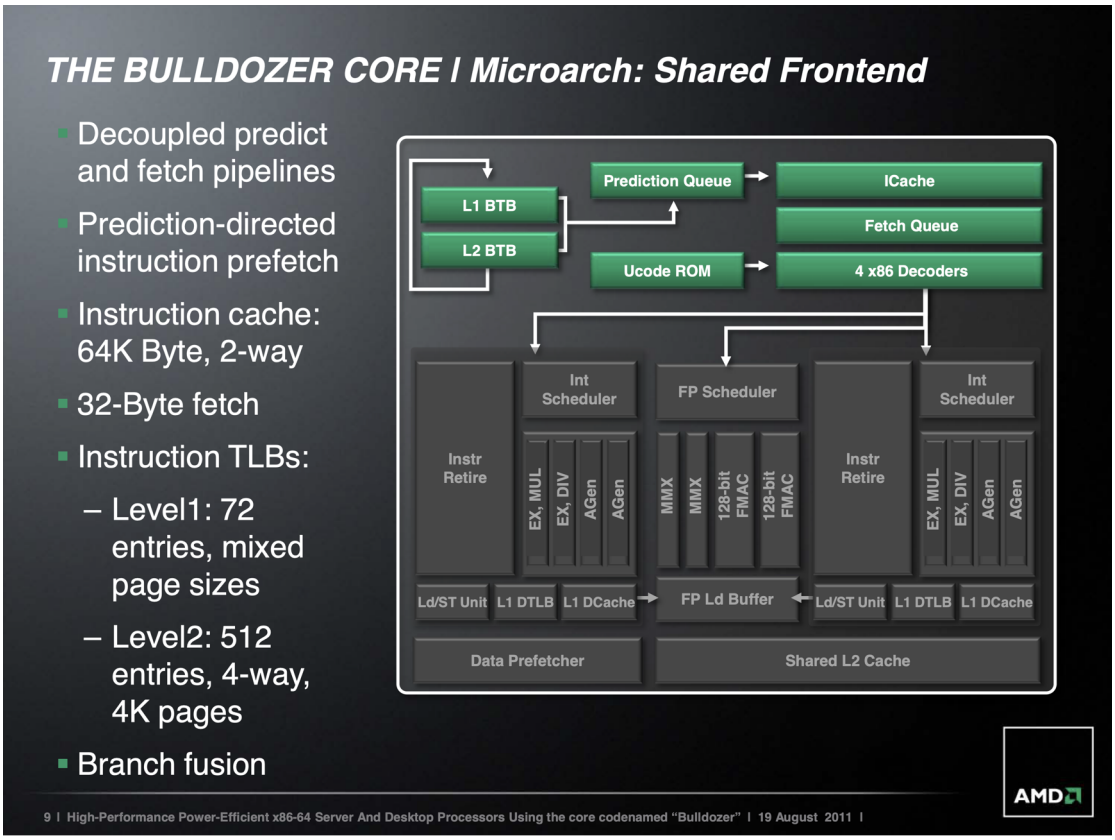
The catch is that this all implicitly assumes we can Generate an Address *per cycle*, or at least at a faster rate than Cache Access can pull in line from L2. This turns out to be a somewhat delicate point that I'll discuss in detail a few pages down.
The basic problem is that it's fairly likely that, by the time your *instruction footprint* is large enough to exceed the L1I, your *trace footprint* is large enough to exceed a single cycle BTB :-(

It seems like current x86 and ARM Ltd designs (small L1I) are just on one side of how this plays out, whereas Apple (large L1I) are fairly far on the other side. If Apple can figure out a way to grow the BTB substantially while still at single cycle access, then Fully Decoupled Fetch makes sense, but otherwise not.
I think we're at the point where we need simulations, but my intuition is no longer nearly as strong as it once was that Fully Decoupled Fetch is always worth doing.

It's worth noting that ARM Cortex X3 seems to engage in a fully decoupled Fetch design (note in the slide below the queue between "Branch Prediction" [what I am calling Fetch or Address Prediction], and the cache access stages).
But the X3 design does not (at least according to the slide) take the last FDIP step of also running an I-prefetcher in parallel with Fetch (ie loading lines based not only on the fact that they are predicted to be in the Fetch stream, but based on other heuristics like a line or two adjacent to the next Fetch line).
There's a discussion of X3 here, https://fuse.wikichip.org/news/6855/arm-unveils-next-gen-flagship-core-cortex-x3/ which you may find interesting to compare with M1.



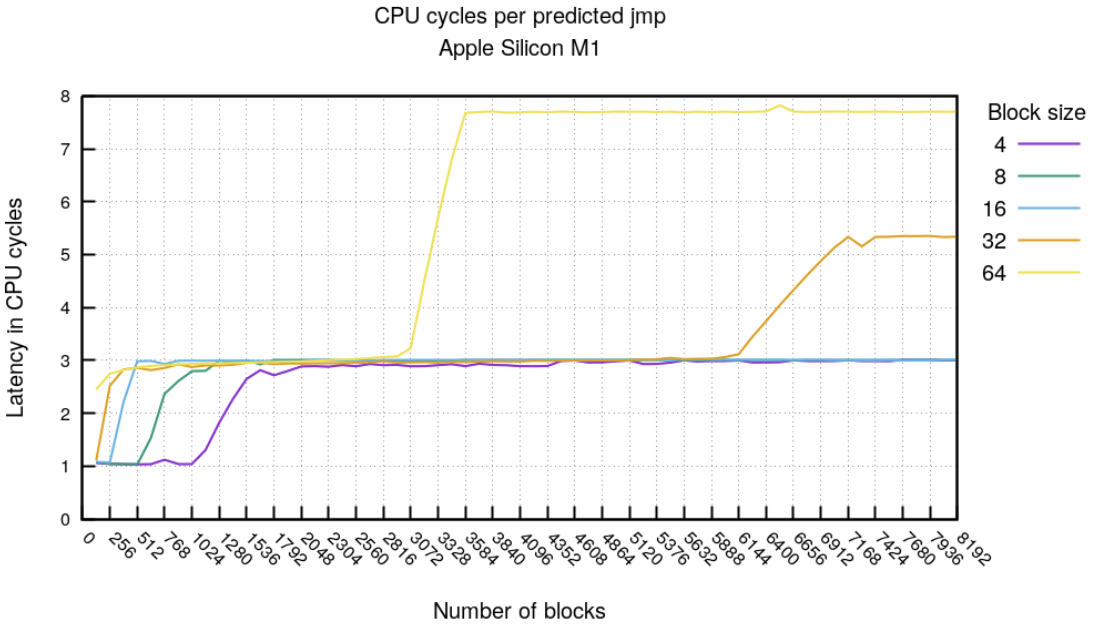
AMD have apparently in the past run a fully decoupled system, as in this slide from 2011



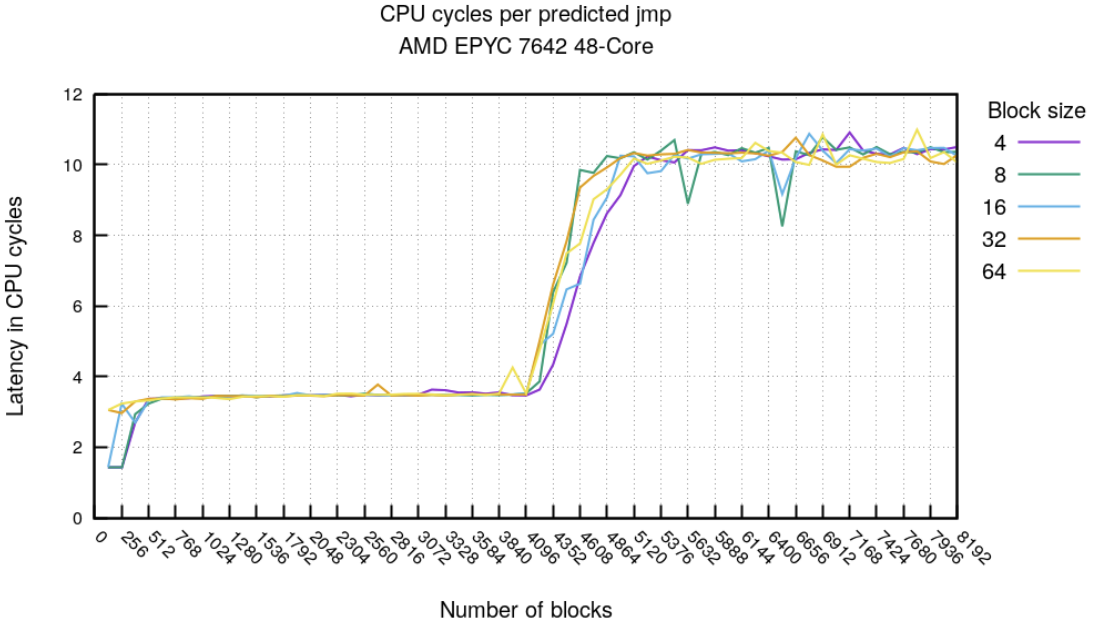
from https://old.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.19.9-Desktop-CPUs/HC23.19.940-Bulldozer-White-AMD.pdf. You would assume they still do so today, along with Intel; but more recent diagrams never show the address queue between prediction and I-cache.

More generally I am told that all the modern x86 designs use Fully Decoupled Fetch, in part because their I-caches are so much smaller than Apple, so it's more important for them to get the prefetch win that's enabled by running Fully Decoupled Fetch.

At this point, maybe it's worth discussing the Cloudflare results:
https://blog.cloudflare.com/branch-predictor/
I no longer have access to an M1, so I cannot independently verify these but assuming the code is what I think it is, let's proceed.



Compare with



Note the big jumps for Apple at 3072 and 6144, and for AMD at 4096.

In both Apple cases 3072 traces that are each 64B long, and 6144 traces each 32B long comes to 192KiB. But the AMD curves all jump at 4096.

To me this suggests that

- AMD's BTB (second level BTB) holds 4096 entries. Doesn't matter how long the traces are, things go bad once we exceed the BTB capacity and cannot accurately predict the next trace to fetch.
- 4096*64B=256KiB. This is much larger than the AMD L1I size. So AMD are successfully prefetching I-

lines into the L1I. This could result from a simple sequential I-prefetcher (notice that three successive I-cache lines have been touched and prefetch the next one, like a data sequential prefetcher); or it could results from FDIP.

Note the block size/trace size is 64B but contains one jump, so one cache line (64B) is processed in what looks like about 3.5 cycles.
That means Address Prediction can possibly run ahead of Fetch and Decode (eg if some BTB accesses hit the faster BTB1, with the remaining accesses taking 3 cycles in BTB2), and runahead address prediction can be used for Fetch far in advance of when the instructions are decoded (so acting like a prefetch, like FDIP).
The timing works because the BTB2 address generation time (3 cycles) is slightly longer than the block processing time (3.5 cycles). This is not true for Apple hence we get back to the argument that Fully Decoupled would not be a win for Apple

What about Apple? The Apple pattern is very different in that, at the high end, we start paying more cycles at the point where we exceed L1I, ie based on the *length of the traces*, not the *number of traces*.
This suggests that
- the BTB infrastructure can hold at least 8192 traces
- for this type of code pattern Apple is not prefetching a line into the L1I, so that as soon as we exceed L1I size we pay the cost of an L2 access.
This in turn implies that Apple has neither a sequential I-fetch predictor (unsurprising; this is not a great design for an I-prefetcher) nor a fully decoupled fetch (which would be able to queue up the relevant fetch addresses in advance of cache access and thus have them act as FDIP).

The second difference between Apple and AMD is that AMD's cost per cycle (ie cost per trace) is the same at the high end, regardless of the trace length. This again tells us that the cost is a *trace* related cost, eg the cost of a mis-predict then flush.
For Apple the cost at the high end is different for larger traces vs shorter traces. This tells us the cost is associated with the amount of data being moved.
Assume Apple is moving two lines of data from L2 to L1I, and the L2 access delay is about 14 cycles.
Now assume 64B blocks. Then that 14 cycles is amortized over two fetches (one that missed to L2, one that was able to hit the successor line that was next-line-prefetched into L1I) and so the average cost per block is 14/2+ some BTB overhead.
Likewise the same case but with 32B blocks. Now that 14 cycles is amortized over four fetches, so the average cost per block is 14/4+ some BTB overhead. That essentially matches what we see.

The second interesting question is what happens at the lower end.
For AMD we know that Zen2 has an L1 BTB of 512 entries and and L2 BTB of 7168 entries, according to https://en.wikichip.org/wiki/amd/microarchitectures/zen_2#Block_Diagram
I'm not sure why the 7168 BTB2 seems to give us an effective size (~4000) that's so low; maybe we are seeing terrible aliasing effects because the blocks all begin at power-of-2 address offsets, so we land up using only half the ways? The same sort of thing seems to hold for the L1 BTB; we're hitting that and getting a throughput that looks like 1+some overhead, but only for about 256, not the full 512 entries.

Once again the Apple story is very different. Once again there isn't an obvious number of traces (ie number of blocks, to use the terminology of the graph) at which we jump, rather we jump from 1 cycle to 3 cycles when the total size of the loop (number of blocks × block size) exceeds 4096B.

So an initial hypothesis is that this is again a cache size issue.
As we will eventually see, again in great detail, Apple has a tiered set of Instruction "caches" depending on the exact details of the code. The simplest short loops are held in a loop buffer; while more complex loops(longer and/or requiring branch prediction because the branch outcomes are not constant), but which fit, are held in an L0I.

Suppose we are seeing performance within this L0I vs performance in the L1I.
+ This fits with the effect being linked to a constant loop size regardless of the block size.
+ On the other hand, I can't see any obvious reason why performance should drop when a loop exceeds the size of the L10! The L10 loop buffer exists for energy reasons, not performance reasons. And we don't see any sort of data movement penalty (cost dependent on block length) like we saw when M1 was missing L1I and had to go out to L2.
So I think this hypothesis fails.

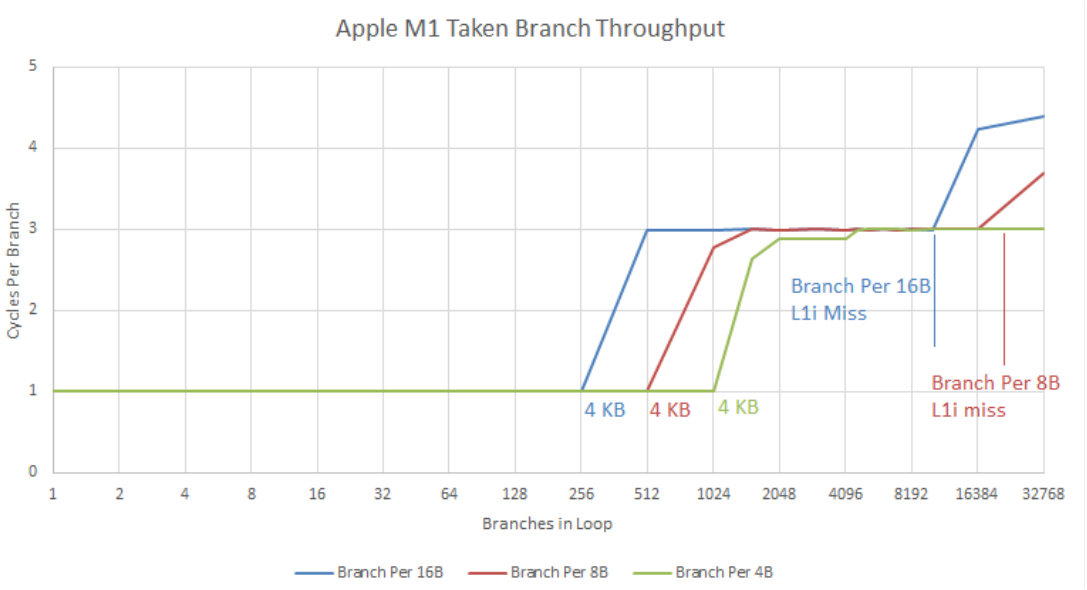
A different interpretation of the low end of the Cloudflare graph is that it reflects a cleaner version of what we saw with with the AMD results: aliasing of address indices into the table, ie as the blocks get larger successive block addresses match at intermediate address bits, and so we use fewer and fewer sets.
This sort of thing can be fixed with a fancier BTB indexing hash, but maybe there isn't enough time for such hash?
If this were my code, I'd try a few runs with different block sizes, like say $2^n - 1$, to try to avoid this sort of aliasing.
If this interpretation is correct, we can summarize that:
+ Apple's primary BTB (we will see that there are other specialist BTBs) is 1024 entries in size, and uses a very simple indexing hash that is easily broken by powers-of-two addresses [which admittedly are extremely fake and not present in real life]!) AMD likewise probably doesn't care about such fake address aliasing possibilities.
+ with an L2 BTB that holds at least 8192 entries.

It is striking that Apple never mentions an L2 BTB in their BTB-related patents, but it's possible that this was considered an irrelevant distraction; or that the split from a single level medium-sized BTB to a larger, but split, BTB was implemented after the most recent BTB patent I have seen (2017).

One question you may wonder is why was AMD, at the high end, only able to use half their L2 BTB? If this is indexing related, why didn't we see the same sort of staggered pattern of graph jumps, like we saw at the low-end for both Apple and AMD?
My guess is that
- some sort of hash of address bits is being performed for the L2 BTB (we've already accepted that that doesn't have to be single cycle, probably for both Apple and AMD) but

- AMD is passing through the second lowest address bit unchanged and unhashed as the lowest index bit (perhaps used as an energy saver, so only one of two halves of a tag SRAM are woken up), and since that second lowest bit never changes (our addresses are always four aligned) we only have access to half the indices. In more realistic code, fetch addresses would probably be randomly scattered at this 16bit granularity, so this would not be a real hardship to performance.

Other people have suggested that, even though AMD claimed the L2 BTB was 7168 entries this is a mistake. Alternatively, perhaps a misunderstanding? Perhaps ~3000 of those entries are dedicated to special cases like function calls and returns) so the “normal” BTB2 only has 4096 entries?
We will see that Apple in fact has a specialized BTB for function returns, so this is not a crazy idea.
We can get more insight from a different versions of this graph (from the RealWorldTech blog), which looks like:



We see this version extends out 16384 entries before jumping, for traces that are 8B long.
This suggests that the BTB holds at least 16384 entries. (Because if the BTB were unlimited in size, we’d expect the next jump point (for 16B traces, once L2 is exceeded) to be at 2×6144; likewise twice that for 8B traces). These earlier jump points suggest that we are now hitting the limits of the BTB before hitting the limits of L1I capacity.)

It looks like there is a mildly fancy hash in place for BTB2 indexing, enough so that the 16B traces (which are missing one bit of entropy in their address compared to the 8B traces) are capable of using about $\frac{3}{4}$ of the 16384 cache slots (rather than using just 8192, as we’d expect if just the lowest address bits were used with indexing).

The rapid rise at the high end of the M1 curves (consider especially the Cloudflare graph) tells us that the line that is being fetched once we exceed L1I is a miss to L2.
This in turn suggests that no sequential prefetcher is being used for I.
The cost of the 64B case suggests that a next-line prefetcher is being used (so that the L2 cost of about

16 cycles is being spread over two or three lines, not just one line).

As promised, let’s look at the numbers in terms of whether Fully Decoupled is ever a win for M1:
Finally consider this: Decode can handle 32B per cycle (eg one branch and 7 NOPs).
Consider the 64B block length case. Suppose we have a Decoupled Fetch design.
Then every three cycles we
- generate a new Fetch address
- Fetch 64B into the Instruction Queue
- Decode 32B (taking two cycles)
We are throttled by the three cycles to generate the Fetch Address.

If we are Fully Decoupled nothing changes because the slowest part of the pipeline remains address generation. We can generate an address per three cycles, fetch its line per one cycle, and decode per two cycles.
This means we cannot see the presence or absence of Fully Decoupled Fetch with 64B blocks (let alone shorter).

Now suppose we use 128B traces (and that the BTB stores maximum length traces that are at least 128B long, which patents suggest is the case, but who knows for sure).
Then in that case

We can Generate an Address every three cycles.
We can Fetch 128B every two cycles.
We can Decode 128B every four cycles.
We are now throttled at Decode.

In this case:
- if we are Decoupled, we will Generate Addresses and Fetch until the Issue Queue is full, then we will run at the speed of Decode.
Now what happens when we exceed L1I capacity?
Fetch is running some distance ahead of Decode, and can start accessing L2 while Decode works through what's already enqueued in the Instruction Queue. This will give some degree of protection from the L2 latency as long as the Instruction Queue still holds entries.
As long as
+ we can generate an address every three cycles AND
+ this address translates into pulling in 4 cycles worth of work AND
+ the Instruction Queue is deep enough (at least around 15×8 instructions in size)
then I think we are fully insulated, it will look like we now have perfect I prefetch.

So even this case doesn't definitively tell us whether we are Fully Decoupled vs just Decoupled!

For Fully Decoupled to show value relative to Decoupled requires that we be able to generate addresses faster than we can Fetch.
But as we've seen, once we start accessing a large number of traces we're facing
- BTB2 access results in one address generation/three cycles
- but Fetch can handle the maximum length reasonable trace (16 instructions, one cache lines) in two cycles, even if it’s in L2.

You can view this from a different angle.
It’s fairly obvious (and queueing theory will confirm) that a queue is in equilibrium when the average input rate equals the average output rate. But in the cases where the Address Queue might conceivably be useful (either for I prefetch, or because we are constantly accessing the BTB2) it takes us three cycles to generate an address and two cycles to utilize (ie fetch from L2) that address.
This difference is just too large for the minor effects that change the average (the occasional hit in BTB1 or L1I, or extremely long traces that result in multiple cache line accesses) to matter much.

Either we are operating withing BTB1 and L1I and the point is moot, or we are operating in BTB3 and L2 and the gap is unbridgeable. For traces that are shorter than 64B (ie most traces) the gap becomes even worse! The usual case is to have to use BTB2, but for the next trace to be within the same cacheline as previously fetched, or a small jump into the next cache line, fetched via the Next Line Prefetcher.

In a way we are concentrating too much on the Prefetch/L1I miss problem, when the bigger problem is the slowness of the BTB2! So we need to figure out a way to improve that.

One thing that might change the relative calculus is the following: Do we have to accept that the BTB2 always takes 3 cycles? Could we not detect that the code patterns have changed and pre-emptively "prefetch" (ie move) appropriate prediction data from the BTB2 to the BTB1? As mentioned, in fact IBM's z15 mainframes do precisely this! This is a lot of non-trivial work, but one day Apple may also reach that point.
The IBM scheme is somewhat convoluted and I'm not sure quite why it's done as it is done; perhaps a simpler scheme based on I-prefetch type patterns (which detect when control is being transferred to "a new area" and prefetch in response) might be good enough, and simpler to implement?

And, of course, if this *can* be achieved to a significant extent, then at that point we are back to all the arguments why Fully Decoupled Fetch is a great idea.
BTB-prefetch – perhaps an idea whose time is coming?

But that's not the only possible solution, and perhaps not even best. Consider this alterative viewpoint:
the BTB (at L1 or L2) is a machine that, when given an input PC spits out a trace (length and target PC).
What makes it impossible to run the BTB2 faster is that (for a large lookup table) it takes three cycles to get from the input PC to the output. All true.
BUT the output is fed into the same BTB machinery to produce a second trace, and then a third. Assuming the predictions along the way are correct, you can build a table that, given a single PC, spits out two or three successive traces. If we built such a table, we're back to generating (on average) a new address/trace per cycle!

Two points that are important, but not deal-breakers.
First is that this pleasant scheme breaks down when a trace ends with a return. To predict the return, we also need to access the RAS to get the return address, and that return address is dynamic, meaning we can't statically encoded the most likely next trace into our table. We have to accept that when we hit a return, that row of our BTB has to encode only one or two traces, not the full three traces; the row has to end with the return.

Second is that you could also do this with the L1 BTB, but the performance tradeoff makes it less worth doing. The second trace is a little less likely to be accurate, the third one even less so, and there's little value (for *now*; to avoid tangents I won't expand this) to generating more than one Fetch Address per cycle.
But with the L2 BTB the calculation changes. Now there's a lot of win in generating mostly accurate Fetch Addresses in advance (for, as we have seen, prefetching purposes) while the fact that there are going to be two dead cycles anyway between every address generation means there's little real downside to generating slightly inaccurate traces. To the extent that they are locally inaccurate, they can mostly be patched up in the usual ways by the Branch Predictors before Decode, meaning the loss of only two or three cycles – which are otherwise dead anyway, so who cares, we're certain to come out ahead overall!

Overall I'm more enthusiastic about this solution than a BTB-prefetch type solution, but I'd love to see both academic and industrial work in this space.

multi-cycle branch prediction

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 the invalid instructions and re-pack the Instruction Queue starting from the correct address (from which we are now fetching). We may lose a few cycles, but if we had enough instructions to keep us busy, queued in the Instruction Queue ahead of where we had to flush, Decode and later may not even notice the flush and resteer.
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.
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 a diagram (from (2016) https://patents.google.com/patent/US10747539B1 *Scan-on-fill next fetch target prediction*). Ignore the details, consider the big picture.
Pipe Stage 1 decides the next Fetch Address. This is derived from
- various single cycle predictors
- one or more multi-cycle predictors 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), or a Return (which triggers the RAS [Return Address Stack]). Normally these would have been correctly handled earlier, by Fetch Prediction, but this may be new code that's being executed for the first time, before the Fetch Predictors have been trained.
- 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.

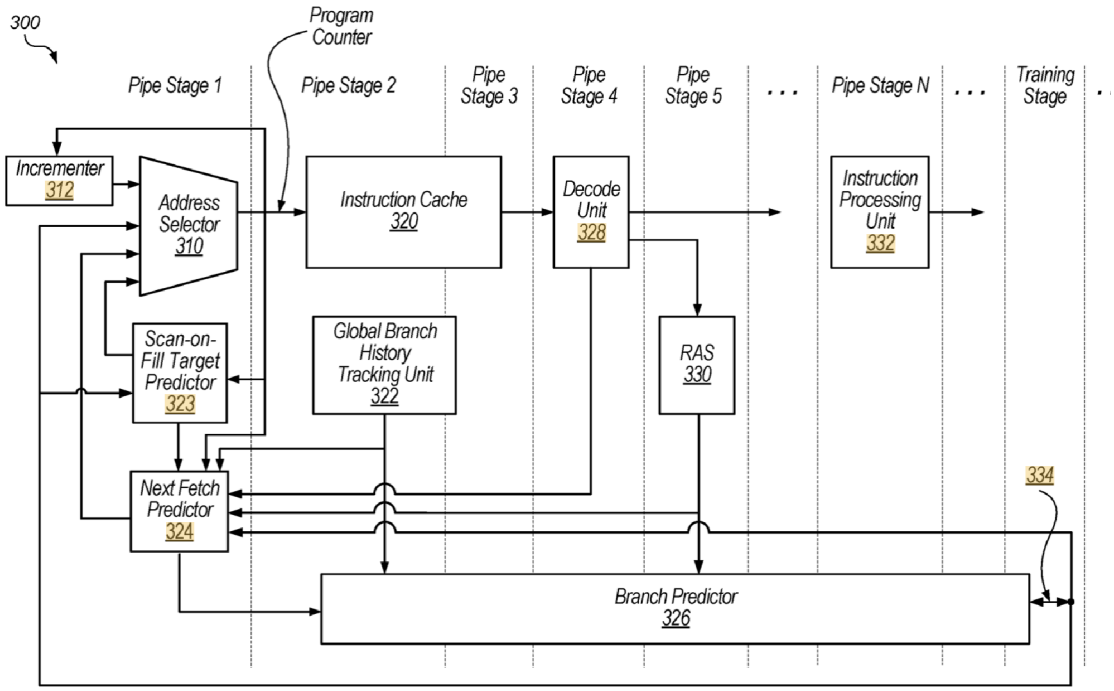


FIG. 3

predict traces, not branches

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 is not branches but break-points in the sequential 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 then, as already mentioned, we have to consider cache line boundaries: If we’re using 64B lines (16 instructions) then a fair number of times a trace will cross cache line boundaries.

On the plus side, this is very easily solved by having multiple cache banks with sequential lines in different banks. In fact Apple appear to have been fetching a maximum of sixteen instruction, and fetching traces crossing two cache lines, for a long time, since at least 2011 or so.

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.
Alternatively, you could argue that most work happens within loops, and loops should be handled separately from general Fetch code. It may be easier to make a loop “wider” so that it’s feeding a steady stream of ten or more instructions into the core per cycle, even if the mainstream Fetch code remains limited to a single Fetch Prediction per cycle.
Later we will see how Apple was already doing a simple version of that for loops in 2012.

avoid branches when possible

Before we try hard to improve branches, it’s worth asking, do we even need a particular branch?
Many short branches take the form of `if(test){single instruction}`. To ensure that just that single instruction is conditionally executed, we lose the “instruction stream coherence” provided by sequential instructions, and have to add an element of unpredictability to the system.
A solution to this is to provide *predicated* instructions, instructions that (depending on the condition codes) either execute or act as NOPs. Note that their is no branching going on here, Fetch is not affected by these instructions; the decision as to whether the instruction executes or NOPs happens later in Execute.
Predicated instructions have fallen in and out of favor. ARM32 used them aggressively, ARM64 provides only limited predication (the `CSEL` and `CCMP` instructions). Unfortunately Intel shipped a few chips with poorly performing predicated instruction (`CMOV`) which taught a generation of Intel fans to hate predication. But it’s a concept with value especially for hard-to-predict branches.

One way you can use the idea of predication is suppose you have an instruction stream that looks like

```
branch conditional +8
some instruction
following instructions
```

which matches our single predicated instruction pattern. The Decoder can detect this pattern (a conditional branch+8) and convert the branch to a NOP, and the `some instruction` to a predicated instruction. IBM have a patent on doing this (2000) https://patents.google.com/patent/US6662294 *Converting short branches to predicated instructions*, but that patent is expiring soon, and it's not optimal. A better solution would be to only perform the conversion to a predicated instruction for the few (but troublesome) branches that are known to be unpredictable.

Apple surely know about this sort of technology because they have a patent for the reverse!
The ARM-32 Thumb instruction set has an instruction `IT` (`If-Then`) which is basically a predication of up to four following instructions.
Now suppose that one of those following instructions is an unconditional branch. If you treat it purely as a predicated instruction (and thus only "execute" it, or treat it as a NOP, at execution time) then

there's a long bubble if the branch has to be executed, meaning fetch has to be re-steered once the branch is executed. (This is one of the problems of an ISA carelessly allowing *any* instruction to be predicated...)
The Apple solution (2013) https://patents.google.com/patent/US20140244976A1 *It instruction pre-decode*
- detects the presence of the IT instruction and the unconditional branch at Pre-decode
- "marks" the unconditional branch as a conditional branch (based on the predicate)
- treats that conditional branch like any such branch for the purposes of branch prediction and instruction fetch.
This patent also confirms the (unsurprising) fact that Apple's I-cache used pre-decode.

Predication has received a bad rap because (*when branch prediction works*) the speculation shortens the critical path of code, whereas predication appears to not allow such a shortening.
This has meant that either the compiler has to guess whether a particular branch might be poorly predicted, or the programmer has to indicate this; and in the past both compilers and programmers have done a sub-optimal job.
But this is deceptive! There is no reason the flags governing predication cannot be speculated, like any sort of Value Prediction; and Value Prediction, like Load Aliasing Prediction, can, in principle, be unwound using Replay rather than Flush, making a mis-speculation a lot less painful.
I expect that over time Apple will begin to implement Value Prediction in its cores (we have seen patents to this effect [the strided load patent], though no evidence so far of implementation), and once enough of this machinery is in place, Value Predicting the flags for predication is the next obvious target, after load addresses.

control flow (branch) prediction

Now some general points about predictors, before we get specific.

don’t pollute training data

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/when 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 all 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 if the speculative instruction stream turns out to be mispredicted!
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.

context switching

How should we handle context switching? Traditionally this has just been ignored, with an expectation that, sure, after you context switch things will suck 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 now have to add data paths 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, usually, 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.
This is not as crazy an idea as it might at first seem. Some of the various security concerns over the past few years about information leakage between processes have been based on manipulating/exploiting the branch predictor so as to spy on another process. ASID tagging limits that sort of thing.

+ 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.
This sounds good, but it does require substantial new infrastructure in the design because code, as written by programmers, 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 you would need various back-channels of some sort to move the virtual address stream present in the instructions through the ITLB, to convert it to a physical address stream.

how (at an abstract level) do Fetch Predictors work?

So, still operating at the abstract level, think about how you'd implement prediction 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 unit calculated as being the next PC)
- these will be reported to the Fetch Predictor, the Fetch Predictor will record them 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:
+ startOfTracePC
+ branchPC
+ branchTargetPC
- using these, and looking backwards, we can also fill in, for previous entries, how long the Fetch Group should be, and what PC to use for the next cycle of Fetch
- so eventually this table will be reasonably densely populated, by a series of entries that look something like
(tag: tracePC contents: traceLength, nextTracePC=targetPC of branch ending the trace)
which we can use to construct an appropriate series of Trace Fetches.

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 trace, and the length of that trace; 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 can, 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?
This is an interesting question (and it's hard to believe that maximal width Fetch and Execute are optimal while in this unknown case) but I've seen no work investigating the issue.

There are heuristics that can be used somewhat successfully to fix things up/improve things even in this case of limited knowledge. For example suppose we know where branches are in a Fetch Group, even if we don't know whether they are taken or not. (We might know where the branches are because instructions were pre-decoded, to classify them into one of a few instruction classes, as they were placed into the I-cache).
If you see a *backward* conditional branch chances are it represents the end of a loop, and chances are it will be taken (since most loop bodies are executed more than once).
If you see a *forward* conditional branch, you have no real way to guess, but
- not taking the branch will use less energy.
Not taking the branch means your energy cost will be
- always pay the lower energy of not taken
- half the time have to resteer and pay the higher energy of taken.
The alternative is to guess forward taken and your energy cost will be
- always pay the higher energy of taken and
- half the time have to resteer and pay the lower energy of not taken.
Clearly the first option is better, but we can do even better, a lot better!
The most common case of a forward branch is code something like

```
if(condition){one or two lines of code}
```

which will translate into an instruction stream something like

```
branch conditional forward; instruction 1, instruction 2, ..; target of branch
```

Now suppose Fetch does not take the branch, so what's packed into the Instruction Queue is the stream like the above, but a slower predictor (like the TAGE branch direction predictor) predicts two or three cycles later that the forward branch should have been taken. How do we recover?
In this case we do not even need to resteer! We can simply delete the invalid instructions between the forward conditional branch and the branch target from the Instruction Queue!
(I expect "deleting" instructions in the Instruction Queue in this way probably actually means setting an invalid bit to true, not literal deletion; then later data movement of instructions to Decode will just skip over invalid slots.)
This relies on the branch predictor having good data while the Fetch Predictor does not, but that's a somewhat common case because the Fetch Predictor has to be somewhat smaller than the Branch Predictor (to fit in one cycle).

Naturally, even with good heuristics, we want to detect and correct errors as soon as possible.
One way to do that is at Decode, where 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.

The patent (2016) https://patents.google.com/patent/US10747539B1, in the Figure 3 section, as an aside, tells us that Decode forwards branch related instructions directly to Fetch and the Branch Prediction subsection, presumably for the sorts of purposes described above, but does not give details.

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.

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 (all wasting energy) and only one way to be correct.

How might such a Fetch 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 can’t both 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 (shared libraries).

Alternatively we could reduce aliasing by using cache type technology. The bare minimum is attach a tag (say a few more high bits from the PC, all the way to the full 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 validation tag in the direct-mapped case 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.)

adding specialized control flow predictors

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; and when a specialized predictor disagrees, we can kill the Fetch at an early stage, 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 anything 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 to decide when to end, rather than being fooled by the fact that this loop has jumped backwards 1000 times so 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 and most of the Fetch machinery?

branch direction predictors (leading to TAGE)

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.
Such a thing 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. This idea is called *gshare*, and was state of the art as of around 2000.

Actually I think even standard gshare is sub-optimal. If you think about both the string of history bits and the string of PC address bits, in both cases the lowest bits have the highest entropy. xor'ing is an optimal method of combining bits, but it combines high entropy with high entropy, and low entropy with low entropy! I think you could do slightly better by reversing one of these so that the highest entropy address bits combine with the lowest entropy history bits. I've never seen anyone suggest this, but it seems a cute easy tweak that might be worth a percent or two. The same idea, of course, equally applies to hashing addresses for any cache...

(The original McFarling gshare paper, https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf, page 11, appreciates the point enough to suggest xoring a short history with the upper bits of an address, not the lower bits [eg if combining 6 bits of history to create a 10 bit index], but doesn't take the next step of reversing the history!)

Once again you can go wild with variations on gshare, but the current world champion (literally! https://jilp.org/cbp2016/program.html) is named TAGE goes in a rather different direction. It's an extension of the above history vector idea, but implemented in a way that allows for a range of history lengths, from short to 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* (as opposed to just a sequence of taken/no-taken decisions). Such a path also includes the non-conditional aspects of history like function calls.

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, more or less, 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 ideas 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, but not great, even apart from the aliasing that results from using a limited amount of storage.

Implementing this predictor in a practical way means we cannot use the full PC as index, just, say, 12 bits; which means that, 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 all this (!!!) 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 we want is something like: “match the branch history backwards for as many entries as possible before the number of matching paths drops below a statistically significant level”.
So
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 some 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 can we use the essential idea? Yes!
First we implement the branch path, 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.
Second we hash this path vector down to 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 continue doing this using geometrically more (that's the G, for Geometric, in both O-GEHL and TAGE) of the history bits, so 4 bits next time, then 8, 16, all the way to all the 256 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 it’s not really.
- The index based on zero bits of path vector you understand: every time this branch is taken, its counter (as determined by the lowest 12 bits of the PC) goes up; when not taken, the counter goes down.
- The index based on two bits of path vector is essentially the same: for each path that has 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 highly unlikely that two strings will match to the same hash slot; and we catch almost all the (few) cases of aliasing via the tag comparison.

So with the basic understanding out of the way, the remaining question is: we create nine indexes and perform nine lookups. Which one do we use?
- For prediction we use the longest match that is valid (ie matches tag as well as index).
- For training, we use the usefulness bits to decide which table(s) 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 details of how you choose which table to update on retraining, and why.
This same idea (use long history vectors, and pull out the best match from matches at various length) can be used for various other predictions, for example indirect branches, or 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 generally need two separate storage items for these two tasks, eg a storage for the branch target address, and a separate counter for “how often has this address been correct vs failed”.

What matters for our purposes going forward is that humanity knows how to build remarkably accurate branch direction predictors, *but* these are slow enough that, to get real value out of them, you need a Fetch implementation like I have described – something fast enough to generate a prediction every cycle, plus a mechanism that can validate those predictions over the next few cycles.

In particular for any particular Fetch Group, in parallel with the Fetch Group being loaded from the I-cache
- you want to know where the conditional branches are in the Fetch Group
- you want to test each such branch (before the last) to validate that it was not taken
- you want to validate that the last branch in the Fetch Group (if it ends the Fetch Group and is a conditional branch) was taken.

If you think about it, this is not exactly trivial! You can mark conditional branches in each cache line via pre-decode, but that doesn't help at this stage because, ideally, you'd be doing Fetch Group validation in parallel with cache access, to generate a correction as soon as possible, and so you don’t yet have the cache line, you’re doing this while the cache is accessed!
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 (eg 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, while some may have many branches (all but the last, hopefully, not taken). How many conditional branches per cycle are you prepared to look up in your TAGE box?

One possible 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 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 the 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.

However, there is another possible answer used by Apple that is not at all obvious and extremely clever, relying on some non-intuitive properties of hashing, as we'll see when we start to look at the patents.

indirect branch predictors (ITTAGE)

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.

All this sounds like a lot of storage, mostly for TAGE, but also for ITTAGE, the Fetch Predictor (and various other items we will see). And yes it is! The general expectation with a modern CPU is that the area (and hence amount of storage) devoted Branch/Fetch prediction is about the same size as the I-cache, and may well be larger.

the competition (deep dive into Pentium-M)

Either before or after you've read below, you may want to read (2008) http://www.ece.uah.edu/~milenka/docs/VladimirUzelac.thesis.pdf *MICROBENCHMARKS AND MECHANISMS FOR REVERSE ENGINEERING OF MODERN BRANCH PREDICTOR UNITS*.
This is admittedly an old document, but it gives a deep exploration of the precise details of one particular design (the Pentium-M) and, if anyone is so inclined, describes the sorts of benchmarks the author constructed so as to analyze this particular design.

patent exploration as to what Apple

does

As usual, we now see what we can learn (via patents or experimentation) of the above hypotheses. We'll start by considering how we can avoid prediction (if possible!), then discuss the general features of Fetch Prediction, then various specialized predictors.
Remember always the pattern that
- Fetch has to generate a prediction for the next trace, based on the current PC, within a cycle;
- but specialized predictors can take a few more cycles to correct that Fetch Prediction;
- resteering Fetch within a few cycles wastes some energy and a few cycles, but as long as the error is corrected before OoO it's not a catastrophe.

Alternatives to prediction

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. But if the correction is later...

How can we improve this situation? 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.
In this case the Indirect Branch Predictor can't give a useful branch target prediction, but it can say "this branch is unpredictable". In that case, we can at least halt Fetch while the instruction stream proceeds through the core. At some point the indirect branch will be executed by the core, the actual target value will be known and can be passed back to Fetch, which can resume fetching.

(2010) halt fetch if indirect prediction is not useful

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. This is described in (2010) https://-patents.google.com/patent/US8555040B2 I*ndirect branch target predictor that prevents speculation if mispredict is expected*.
This patent gives some background as to the 2010 indirect predictor which is very simple, a direct-mapped table.
So, start with the virtual address PC and hash it down to an index (say 10 bit or so). The easiest hash, as always is the lowest order bits; next step up is to xor in a few higher order bits.
This index is looked up in a direct-mapped table, and the some of the higher bits of the PC are com-

pared against a tag. Hence we get a final value of either
- a target address (the target of this indirect branch last time we took this branch) OR
- invalid (the tag does not match the PC).
This functions as a very simple usefulness indicator; later more sophisticated predictors like ITTAGE can also provide a history-based usefulness indicator.
Regardless of details, if we know that we do not have a useful predicted target address, Fetch simply halts. The patent points out that you don't actually have to halt Fetch at this point, there may be some value (or simplicity) in allowing Fetch to proceed for another cycle or three, just as long as you halt it before the stream after the indirect branch enters the OoO machinery and becomes impossible to resteer.

possibilities for hard to predict conditional branches

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 hard to predict branch that
- takes a long time to resolve (eg 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, so it might be a good choice for an E-core, even if a P-core chooses to speculate, accepting that it will only win half the time?

- 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, once 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 even 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 (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 data prefetchers would do the job, but maybe (given how rare these problematic branches are) it's possible to build a specialized predictor+prefetcher 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.
This becomes an especially interesting idea after you read the paper (2022) https://arxiv.org/abs/2209.00188 *Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction*. This is a load-prefetch paper, but the interesting twist it adds is the following:
Consider the (small) set of hard to predict loads (ie what's left after large caches and good data prefetchers). Much of the time spent in servicing these loads occurs not in accessing DRAM but in checking the cache hierarchy before hitting DRAM.
So imagine we add yet another predictor, this time predicting whether or not a load will miss in all caches. Now after missing a load in L1, we can split the load into two requests, one that goes out directly to DRAM, and one that walks the cache hierarchy. That way the DRAM servicing can mostly happen in parallel with walking the cache hierarchy. We don't usually want to do this for DRAM bandwidth and energy reasons, but if we can mostly trust the predictor...
For Apple this scheme seems especially appealing given the memory-side nature of the SLC. In a way we split a request into two halves, one walks upward from L1, one walks downward from DRAM, and

they meet in the middle at the SLC!

- 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 and as many instructions 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). [Runahead has been present in various forms for at least 20 years, mainly to act as a form of prefetch, but this 2020 version seems a much better fit to current realities than the earlier suggestions.]
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 five immediate options for difficult (non-predictable, long resolution) conditional branches:
- halt (saves energy, no progress)
- guess (runs at full energy, half speed on average)
- split into two SMT paths (runs at full energy, half speed on average)
- switch to runahead (unknown energy cost, unknown prefetch speedup)
- use a fancy prefetcher for the associated loads, to speed up the branch resolution
As I said, the last option looks best to me.
Well, that's a problem for the future, we still have multiple tricks left to implement today within the constraints of existing predictors.

The above discussed unpredictable conditional branches that take a long time to resolve.
Much more common are unpredictable conditional branches that are easily resolved. These are the sorts of things that I have suggested should usually be handled by predication, not branching. The predication can run as fast as branches if it is handled via value prediction; the reason this is overall a win compared to just speculating the branch is that value prediction mistakes can be unwound using the same sort of Replay machinery as we already use for Loads, and Replay is so much cheaper than Flushing.
Even for unpredictable conditional branches that are easily resolved but cannot be treated as predicates, if the machinery bases scheduling on criticality rather than simply instruction age, then these types of branches (and all instructions that feed into them) can be marked as critical allowing resolution (and thus limiting wasted effort) a few cycles earlier.

Fetch Prediction

(2011) fetch predictor with hysteresis

We don't get to real fanciness till 2012, but now that we understand the general outlines of Fetch and Branch Prediction, there are a few interesting details to consider even before then. (There are also many less interesting technical details I'll omit!)

One problem with the basic Fetch Predictor as I described it, simply repeating whatever we did last time with this PC, 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. For example, 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.)

This 2011 patent also tells us that, even in these early days, Apple were already using a multi-stage prediction scheme. The Fetch Predictor is single cycle, as described, while a fancier branch predictor may possibly correct the prediction one or more cycles later. Of interest is that, to keep the feedback loop as short as possible, it is the high quality predictor that trains the Fetch Predictor. Hence we have a cascade where
- retired (so non-speculative!) execution trains the quality predictors (with multi-cycle lag) and
- the quality predictors train the fast predictor (with a one or two or three cycle lag)

(2017) replace hysteresis with a different solution

If you think about it, this 2011 patent is not really about the Fetch Predictor per se, it's about how the Fetch Predictor gets updated/retrained in the face of a misprediction.

(2017) https://patents.google.com/patent/US10613867B1 *Suppressing pipeline redirection indications* is likewise about such retraining.
The concern is when we have a short tight loop, such that the predictions of the loop might change within the maximum of (at the time of the patent) five cycles it might take TAGE to produce a prediction.
Because the loop is short, we expect it to be handled all the way from the first few (loopback) cases to the final (loop exit) case before the loop predictor has been trained.
And because the loop is tight we expect that the loopback condition (`i++<4` or whatever), and possibly also branch tests within the loop, to switch their outcomes (taken vs not taken) before TAGE even produces its determination of the loop conditions.

Now this is not a complete tragedy, it mainly means that the loop resteering happens after a few wasted cycles. But we can prevent it from being even worse by ensuring that the Fetch Predictor is not trained with out of date information. (ie, like the hysteresis case, it's basically better to have the Fetch Predictor stuck in the more common case of predicting the loop going backwards than in wasting energy to retrain the loop, only to have the retraining immediately be out of date).

Ultimately the goal is the same as the 2011 patent, but the mechanism seems to be different. The 2011 patent operated by attaching a bit to each Fetch Prediction entry, essentially marking whether the entry changes frequently; the 2017 patent operates by detecting if the branch to be retrained is within a tight loop.
I'm guessing the change comes about as part of a re-arrangement of the loop detection machinery, once someone realized they could use that machinery for this additional task, and so dispense with the area required by the hysteresis bits?

Two additional piece of data we learn from this patent are that between 2011 and 2017
- the Fetch Predictor is no longer a direct mapped cache but at least two-way set associative.
- some aspects of the Fetch Predictor lookup utilize branch history bits.
We learn these because there's a second, rather different, feature described in the patent:

The slow branch predictor(s) like TAGE may produce a final result with variable latency. For example the slow predictors may produce, in one cycle, that the branch is predicted taken, while only in the next cycle producing the target address (eg by performing the add of the branch offset with the PC, remember all TAGE is telling you is taken vs non-taken).
This means the various different check/correction steps are spread over different cycles. We can compare the taken value produced by TAGE with whether the Fetch Predictor assumed a taken value for that particular branch and, if they don't match, kill Fetch at this point; but then we may have to wait another cycle (at least saving energy) for the generation of the correct Fetch Target.
This multi-cycle behavior also affects training of Fetch because, even though once again we have the correct directional value early, we have to suppress training for one cycle till we have all the data needed for training.

This may seems obvious but there is a twist!

For the simplest possible branch case, an unconditional branch, the direction and the target may be available right away from the slower predictor (and may not be present in the Fetch Predictor because that is smaller, and this unconditional branch aged out), so why not, immediately resteer and retrain? Resteering as fast as possible is a good idea, but retraining requires creating the full cache tag, not just the cache index (so some mix of PC and branch/path history) and apparently that is not available in that same cycle, so we have to delay retraining by one cycle to be able to retrain with the correct tag.

In fact a few more details can be filled in from *Scan-on-fill next fetch target prediction*, which we will soon cover. That patent tells us that the Fetch Predictor is two way, but also that it utilizes *two* tags per entry, not just one. The first tag is the PC address, the second tag includes some elements of branch history.
The way it works is that lookup first compares the PC with the PC tag and, if there is a single match we are done. This is traditional two way caching, and allows us to store in the Predictor two PCs whose index hashes alias (and to catch when an apparent index hit is an alias with a new trace, not a real hit). However, if the two PC tags both match the PC, then we compare the history tags and choose based on the "better matching" history. Unfortunately no details of this are given.
Apple's patents tend to use the term branch history to refer to either a history of taken vs non-taken, or a path history of the PC's of branch points; it's unclear which is referred to here, and with so little to work with it's unclear which would be the better solution. It's even unclear to me how much history is stored – if you store anything more than a single bit, you will have cases where the PC matches but neither history tag matches, at which point what do you do – ignore most of the Fetch Predictor entry and rely on the heuristics to handle the conditional branches of this next trace? It's not worth worrying too much because the 2017 patent https://patents.google.com/patent/US10445102B1 gives further slight changes to the system, which we will see soon.

(2012) fetch predictor

We start seeing the structure of the Fetch Predictor itself with (2012) https://patents.google.com/patent/US20140075156A1 *Fetch width predictor*, which validates much of the intro material, even up to a rough sketch of how the Predictor contents are bootstrapped from an initial state of zero knowledge.

Beyond the explanations given 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 even be 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, so the I-cache is (to some extent anyway) double-ported for read.

- 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 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 for this in the academic literature 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.
Apple appears to be following this traditional answer.

- one final cute little tweak Apple describe close to the end of the patent refers to situations like `if(~variable_but_predictable){one or two instructions}`
Thinking about this in terms of traces, the trace in which this code is embedded will sometimes be shorter (when the test of `variable_but_predictable` results in a taken branch) and sometimes a few instructions longer (when the test results in a not-taken branch).
The Fetch predictor, in this case, is retrained to the longer trace, but not retrained to the shorter trace, so it will always pull in the longer trace. Why?
The thinking is exactly what I described much earlier when consider heuristics for Fetch in the case of zero knowledge. In this case, (short forward conditional branch) the damage, once detected by the slow predictor, can be patched in the Instruction Queue by deleting the one or two extra instructions, without requiring a resteer.

(2016) scan-on-fill predictor

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.

The essential idea [I’ll correct one detail once we have the idea] is that when a line is prefetched into the I-cache it is pre-decoded looking for trace exit points. Suppose that, for example, you see a subroutine call at instruction 7, and no branch of any sort in instructions 0..6. Then a trace that enters this line at instructions 0..6 must exit via that subroutine call.
Using this sort of logic (find branches and work backwards from them) we can create some preliminary Fetch Predictor data associated with this line. So the pattern now is
- from the last Trace predicted I know the target of the exit of that trace
- I look for that target PC in the Fetch Predictor (ie what other CPUs usually call the BTB)
- suppose I don’t find it; then I look in the Scan-on-Fill predictor and hopefully I can use some of the information there to construct the next trace.
If the exit point from the Scan-on-Fill predictor is an unconditional branch (goto or function call) we have perfect knowledge for the trace.
If the exit point is a return, we can get the return address from the RAS (return address stack) and things are still pretty good.
If the exit point is a conditional branch, well then we have to use our zero knowledge branch direction heuristics (which are not bad!), and now we do know where the branch points are for the full Branch Predictor. We’ll see the value of that when we look at how the conditional branch predictor manages to perform multiple predictions in a single cycle.

Fetched Group of Instructions 200

205 Fetch Group Address	210 Instruction Offset	215 Instruction Type	220 Exit Instruction Offset	225 Target Address Indication
0x04A8	0	Non-Branch	3	Unknown
0x04A8	1	Non-Branch	3	Unknown
0x04A8	2	Non-Branch	3	Unknown
0x04A8	3	Unconditional indirect branch, R0	3	Unknown
0x04A8	4	Non-Branch	7	Target-A
0x04A8	5	Non-Branch	7	Target-A
0x04A8	6	Non-Branch	7	Target-A
0x04A8	7	Conditional direct branch, Target-A	7	Target-A
0x04A8	8	Non-Branch	11	Target-B
0x04A8	9	Non-Branch	11	Target-B
0x04A8	10	Non-Branch	11	Target-B
0x04A8	11	Conditional direct branch, Target-B	11	Target-B
0x04A8	12	Non-Branch	14	RAS pop
0x04A8	13	Non-Branch	14	RAS pop
0x04A8	14	Call-return	14	RAS pop
0x04A8	15	Non-Branch	Fall Through	0x04A8 + Line Width = 0x04A8+15+1

FIG. 2

There is one technicality to the above that needs to be corrected.

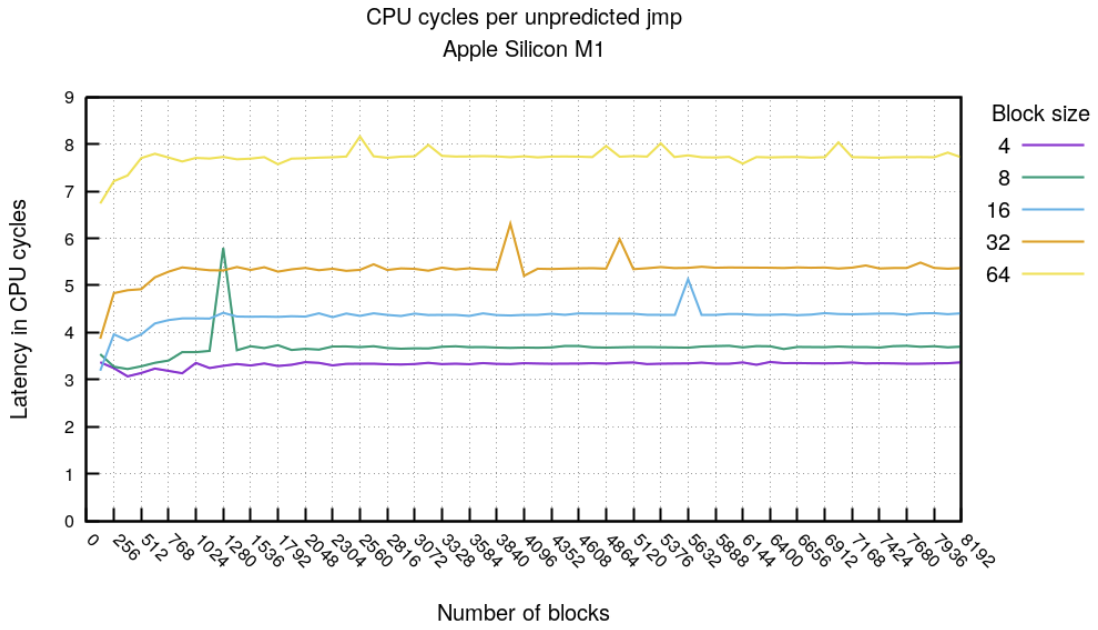
As far as I can tell, basic pre-decode (classify instructions into classes) happens for every line that enters the L1 I-cache. But Scan-on-Fill is more sophisticated and needs to store data not in the cache line but in the Scan-on-Fill predictor structure in the Fetch unit. So the Scan-on-Fill processing does not happen when a line enters the L1I, it happens (hopefully!) some time between when a line is prefetched into the L1I and when the line is actually first jumped to. This scanning happens if there is free cycle available when Fetch (for whatever reason) is not accessing the L1I. This means that it's still possible, though hopefully rare, for a line either not to have been prefetched, or to have been prefetched so

close to when Fetch needed to access it that it was not scanned; and its contents may not be present in the Scan-on-Fill predictor.

(Given that the I-cache is split into at least two banks, to allow Fetch to access a trace that straddles two cache lines, one expects Apple has implemented the obvious tweaks to deal with this Scan-on-Fill limitation as aggressively as possible; for example if Fetch only needs to access one bank, then Scan-on-Fill can opportunistically access the other. I'd expect prefetch addresses are placed in some queue associated with Scan-on-Fill, and split by these two banks, to facilitate this.
You can compare this with the various tricks we saw to try to overlap reads with writes on the D-cache side.)

This idea (the Scan on Fill Predictor) is essentially an implementation of the most interesting part of Boomerang, (2017) https://www.pure.ed.ac.uk/ws/portalfiles/portal/29959353/BoomerangPreprint-_1.pdf *Boomerang: a Metadata-Free Architecture for Control Flow Delivery*. The paper has a bunch of ideas, but the one of relevance is the idea of scanning prefetched I-lines to extract what can be extracted of the control flow breaks in the line.

The previously mentioned Cloudflare blog post, https://blog.cloudflare.com/branch-predictor/ includes this graph



I am not sure exactly what the test code looks like, but if we assume it's the same sort of code as tested earlier against AMD, then the idea seems to be that we (somehow) flush the entire branch/fetch prediction machinery, then start a sequence of short, either always taken or conditional (but taken), loops.

Once again I think the Cloudflare analysis is totally off and what we are seeing is in fact the scan-on-fill predictor doing its job. In broad strokes, what we seem to see is that the first Fetch access into a line has a high fixed cost (around 8 cycles, if we assume the lines are already in L1I?). That's the price we

pay per line when a line holds only one branch (64B blocks).
But when a line holds two branches then the Scan on Fill Predictor is able to extract some data from the line in time to be usable by the second branch, and so the total cost (for two branches) is around 11 cycles.
Once the line holds four branches, the total cost is perhaps 17 cycles (8+3*3).
So it looks like we always pay the fixed cost of 8 cycles, plus 3 cycles if we can extract the data from the line using the Scan on Fill Predictor.

Perhaps, under more normal code conditions, a line will be prefetched into the cache and this scanning will happen in advance of any Fetch access to the cache, so that by the time we need to start Fetching from the line, the Scan on Fill Predictor will take one cycle for each prediction? Or perhaps it always takes three cycles per prediction, same as seems to be the case for the L2 BTB?

(2017) multiple optimizations of the NFP

In (2017) https://patents.google.com/patent/US10445102B1 *Next fetch prediction return table* we take the previously described NFP (Next Fetch Predictor) and optimize it in various ways.
Note that this remains the single “Next Fetch Predictor” even though it’s split into multiple tables accessed in parallel. The Scan-on-Fill Predictor remains as a separate entity, but not relevant to these changes.

- To save area, we realize that any NFP entry that terminates in a Return is wasting the storage used for the Target, because we will retrieve the target from the Return Address Stack.
Hence those entries are moved to a separate table called the NFP Return table.
This separate table for Return-terminated Fetch Groups can be appropriately sized (and probably created as direct-mapped rather than 2-way associative).

Something less obvious is that 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.
Recall that what has to be generated each cycle is (jump to this address, and load this many instructions); that ordering is important. When the rule is (jump to the top of the return address stack), then the number of instructions to load is unknown, because the return address could be multiple locations!
The best one can do is pull in everything from the cache line, then delete as many elements as necessary once the line reaches the Instruction Queue and further data about the branch points in the line is known.

This seems like another area amenable to a small tweak. Imagine, for example, calculating the post-return fetch width *at call time*, and storing it on the RAS, so that it's available for Fetch when we pop the RAS.
This is probably expensive to do dynamically, but could be done at the point where a Fetch Group that terminates in a call is inserted into the NFP.

- in the primary NFP storage, once you have a set associative cache, say two-way, there are now two possible slots into which you can place a new entry. If a Fetch Prediction lookup fails, when we retrain the predictor, which of the two slots should we use? Obviously if one of the entries is invalid we use that, but otherwise? What we want is some sort of indication of the “usefulness” of an entry.
So entries are defined by a few different “usefulness” 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 function 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, and better matching what's actually done) is to consider that we have say a 2-bit field representing both confidence and hysteresis, and 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.
It's conceivable that certain types of entries (function calls?) begin life as very confident, with an initial value of 3, while other types of entries (virtual function calls?) begin life as not very confident.

- 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. (eg multiple different histories define different exit targets for this trace).
Rather than accepting this, we define an "overflow table" (in traditional caches, using an overflow cache in this way to deal with low associativity might be called a victim cache; of course skewing is another traditional way to handle low associativity...)
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 (for reasons I cannot see, the patent calls this the NFP Fast Table).
Compare this with the 2016 design. It's hard to be sure, but there's no reference now to two separate tags (one for PC, one for history) or anything similar. My guess is that the history vector and the PC are combined together gshare-style (ie via an xor) as part of the creation of the index hash. The lookup can be validated by having the PC stored in the tag.
This scheme would allow the Fetch predictor to hold multiple different Fetch predictions corresponding to the same PC but with different history patterns, if that made sense.
But this would also result in a constant stream of new patterns (same PC but different history), so to prevent each new pattern from deleting older patterns, we need higher quality usefulness bits attached to each entry, and perhaps the occasional degree of higher associativity.
The design of the overflow table doesn't make complete sense to me. The patent suggests that it's a FIFO, and with no usefulness bits, so even a good, useful entry in this overflow table doesn't have any mechanism to persist beyond its FIFO lifetime, or to be promoted to the main NFP table. Perhaps experience shows that this is good enough, that we tend to have a spot of locality where better than 2-way associativity is useful while we repeatedly call a set of functions (during which, once equilibrium is achieved, we don't enter more entries into either the main NFP or the overflow table), and once we leave this particular equilibrium and move on to new code, it doesn't matter much that the FIFO mechanism ages out a few overflow entries that are no longer valid?

- So in any given cycle, what we will see is the PC is presented to
+ the primary NFP
(As of the patent this was two-way set associative, tag probably gshare with recent history;

As of M1, perhaps two level with L1 perhaps 1024 entries direct-mapped, and L2 at least 8192 entries, probably two, three or four way associative)

+ the Return NFP (probably direct-mapped, probably no history going into the tag)

+ the victim cache Fast NFP (fully associative FIFO of recent overflow entries)

+ the Scan-on-Fill Predictor (some sort of cache but no details given, maybe also fully associative FIFO?)

Note that while the L1 primary NFP has severe timing limitations (has to generate a new result every cycle from a lookup index based on the previous cycle's result) this is a constraint on the index (and using multiple ways); it is not a constraint on the tag.
The tag can still be complex (to test that the value looked up is, in fact, what we want – corresponds to the full PC and some elements of the previous history) with slower, pipelined lookup and testing, as long as their is enough time to kill the Fetch that was generated (ideally before accessing L1I, but even a cycle later is OK, it's just wasted energy and a dead cycle or two.)

Each of these will try to provide a best guess as to the address of the next Fetch Group.
If more than hit gets a hit, the NFP gets highest priority, then the Return NFP, then the overflow table, and finally the Scan-on-Fill table.

One could imagine further subdivision over time. For example, I would guess that direct calls usually don't care about a variable history path and could be moved to their own, direct-mapped, table based only on PC?
Likewise for indirect calls, only they care a lot about history, so their index construction might use more history bits, and be 2- or 4-way mapped (not trying to compete with ITTAGE, but to at least pick up the lowest lying fruit of virtual calls with an easy prediction pattern).
Finally you can probably save a little energy at almost the performance by converting the overflow Predictor and the Scan-on-Fill Predictor from fully associative FIFOs to, say, 4 different, quarter-sized FIFOs activated based on the lowest 2 bits of the index. And perhaps there's some value to marking the most recently used entry, and skipping over that when doing the FIFO updating? (Hardly an LRU scheme [which may well be undesirable], but able to capture a frequent reuse case?)

The patent contains a few intriguing suggestions of further optimizations, but no details.
These include hints that the NFP might operate differently either under low power conditions or under conditions of higher than expected misprediction.

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.

Instruction Prefetch driven by Fetch

Instruction Fetch lends itself to a hierarchy of predictors.

We’ve started with Fetch Prediction which aims at creating mostly the stream of instructions that will be executed by the CPU, but with some overflow, some additional instructions loaded past problematic conditional branches.

More accurate is Branch Direction Prediction which tries for the completely accurate stream of instructions that will be executed by the CPU (which can hopefully be achieved simply by invalidating a few of the instructions in the stream that’s been loaded from the I-cache.

Going in the other direction, we have the FDIP stream which once again aims to be the accurate in-order instruction stream, but with some additional addresses thrown in for possibilities like “nearby” cache lines that even if not predicted as being part of the instruction stream right now, may be so in a subsequent loop through this code.

And then we arrive at prefetch, for which we allow much less accuracy (but we still don’t want to waste

energy!)

some comments on I-prefetch

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. This means that we slightly bias L2 to retaining I rather than D lines (since missing an I line is much more painful than missing a D line).
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, so we won't see it discussed in patents.

Next up is simply ensuring that, like the branch predictors, the I-prefetcher is not tainted with incorrect data. 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.

PIF as an alternative to FDIP

The main alternative to FDIP is called Temporal or Proactive Instruction Prefetch (PIF), (2011) https://compas.cs.stonybrook.edu/~mferdman/downloads.php/MICRO11_Proactive_Instruction_Fetch.pdf *Proactive Instruction Fetch*.
This paper makes two points:
- one is the point I have stressed, that prefetchers need to avoid being contaminated by mispredictions and irrelevant execution (like interrupts);
- the second is that there is a lot of repeated *structural order* in the sequence of PC's accessed by code, if you can figure out a way to compact that structural order (remove the irrelevant parts like loops), and a way to index into it.
This academic research has taken a path that leads to extremely accurate predictors that, however, are also extremely memory intensive, enough so that it seems unlikely that they will ever be productized.

(2016) Callgraph-based (long-range) instruction prefetch

However the PIF viewpoint leads to Apple's second Instruction Prefetcher after FDIP, (2016) https://patents.google.com/patent/US10642618B1
Callgraph signature prefetch, which can be viewed as one level up in the hierarchy beyond FDIP.

Under normal conditions code executes for some amount of time performing loops and making calls within the range of the L1I. To move beyond that code under normal conditions for normal code (and all we care about is normal conditions/normal code) the sequence will be something like a call to a new function A, which will call function B and function C, each of which will call functions D E and F. To some level of accuracy (not the accuracy of something like Proactive Instruction Prefetch, but good

enough) we can summarize this as something like “if we have good reason to believe that function A is going to be called, then prefetch the first line of A, and also prefetch the first lines of B, C, D, E and F”. The FDIP can then fill in a few lines around each of these function entry points, and hopefully the entire packet of required instructions will be in the L1I by the time it’s required. If we’re lucky then, for example, either just before the call of A, or just after but before the call of B, there will be some real work (a loop or whatever) that will take some cycles that can run in parallel with the load of the lines of functions A..F.
Now we’re not even hoping to generate something close to the final instruction stream, it’s more like a rough sketch of the big picture (a sequence of function calls, ignoring loops and movement within each function; that’s to be filled in later, for now just get the lines to the L1I).

To fill in some details, for this scheme we need three major pieces:
- starting at some “event” we need to maintain storage of I-requests that miss in the L1I. At the next “event” we record this set of L1I misses, along with the event.
The idea is that if we choose the event well, then that event defines a context (of what’s in the L1I and what is not) that’s fairly constant, and that is well served by prefetching the matching set of I requests. This ongoing storage doesn’t have to be too large. Of course there will be times (eg when an app starts) where almost every I-request misses, but we only care about the situation once things stabilize.

- secondly we need a set of events to match with these prefetch sets. The events are chosen to be function calls and returns.

- third we need a way to map a the event to the prefetch set. The simplest idea would be to have the PC of the function call (or the PC of the call target) act as the event, but that’s not accurate enough. More accurate is to use some sort of path, like the last four function calls (again either the PC of the call, or the PC of the target) and hash that in some way. This is much like the use of path history for branch prediction, only now
+ we are predicting a set of I-cache misses, not a branch target or direction
+ we are using a slightly different path (based on function calls, not on all branches)

So the basic idea is on each function call:
- we construct a hash based on say the most recent four function calls
- we look up the hash in a Miss Table
- if we get a hit then we start prefetching
- if we get a miss then we record the next prefetch set in the Miss Table along with this hash

comparison with RDIP (2013, an academic equivalent)

This more practical version of PIF looks much liked something called RDIP in the academic literature (2013) https://akolli.github.io/pubs/rdip-micro13.pdf *RDIP: Return-address-stack Directed Instruction Prefetching*. As a very new idea (certainly at the time the patent was filed) RDIP is promising, but many precise details like the best way to construct signatures, the best way to train the predictor, the best

sort of data structure, etc, remain unexplored. The patent reflects this with lost of vagueness!
I'd recommend you read the paper to see explained in more detail what I've just summarized.
The paper suggests a few numbers which at least give us a vague sense of what Apple's sizes probably are.
The paper suggests:
- a perfect I-cache (or alternatively, perfect I-prefetch) is worth about 16% performance improvement over the baseline (32kiB I-cache, no prefetch).
+ a very simple I-prefetcher is worth about 5%, PIF is worth about 13%, RDIP is worth about 12%. These numbers only give order of magnitude (a CPU that goes through instructions faster, like M1, will hurt more when instructions are delayed) but they show that RDIP gets close to PIF (about the best known I-prefetcher right now) at about 1/3 the storage budget, and that there is still some gap between RDIP/PIF and the perfect prefetcher.

- the Miss Table they suggest has 4K entries, and each entry (one for each call signature event) can describe one, two, or three prefetch regions. Each Prefetch region consists of an address and pulls in the line at that address and up to 8 lines around that address.

- overall their storage comes to about 64kB. Remember they're only using a 32kiB L1I!
Does it make sense instead to just use a larger L1I? They simulate that, along with a simpler prefetcher that does not need storage, and it's not good. The larger L1I is substantially slower and uses more power than the alternative of devoting lots of storage to a good prefetcher.

- in the paper, the "events" used to trigger prefetch are both function calls and function returns, and they are based on hashing (simple xor'ing) the top four elements of the RAS stack.

The above details all become more interesting when you compare them with Apple's specific implementation.

The first difference is that academic RDIP does an xor hash of the top few entries of the RAS. Apple seems to construct an "aging" hash where on each event (call or return) we shift the history a few bits, then xor in the new branch target (the call target or the return address). This may work better than the paper, or may be a patent workaround, or may just be simpler to implement since it matches what the rest of the branch prediction machinery is doing.
The paper suggest that using more (or fewer) than the past four addresses is suboptimal, but Apple can easily get that same effect by shifting one quarter of the width of the history before the xor.

A second difference is that the paper suggests associating with each event up to three prefetch source addresses, and loading up to 8 lines from each source. Apple is much less aggessive and much more concerned with efficiency.
Each event can only result in up to two prefetches, of the target line and possibly the target line+1.
Each line to be prefetched has a counter associated with it (so a line in the Miss Table has two counters

for two possible lines) and each counter is incremented or decremented depending on whether the line was/was not useful (ie was accessed before being replaced in the I cache). These counters, along with an indication of how recently the line was used, are used to decide whether to replace an entry with a new possibility.

Finally there are a bunch of more slightly technical details to save energy, like the use of a Bloom Filter before checking the callgraph signature (ie hash) against the Miss Table.

There's also an interesting size optimization: how do you store the address of the line to be fetched associated with a particular callgraph event? The obvious solution uses some large number of bits (perhaps 40 or so) depending on the exact details of how many virtual address bits you support, and only using enough low order bits to define a cache line address, not a location in the cache line. But most of those bits are somewhat redundant, in that the code only comes from a few places – the app itself and various shared libraries. A better solution is something like to split the address into lower bits and something like a page number which can be referred to by a short index. So what is stored in the primary Miss Table is something like a 10bit offset address and an 8 bit index, where the 8bit index refers to one of 256 "superpages" that can be discrete locations for code. I've made up these numbers, but they give a feeling for the idea.

(2020) JOLT, state of the academic art

If you find all this interesting, you will probably enjoy (2020) https://research.ece.ncsu.edu/wp-content/uploads/sites/19/2020/05/D-JOLT.pdf *D-JOLT: Distant Jolt Prefetcher*, which is essentially a systematic investigation of the RDIP idea, varying parameters like the hash used, the type of call history used (eg a stack vs a path history), and when in time to generate the prefetches relative to the current call. They also introduce one clever new idea, namely they split the job of prefetching into two different tables (each using different parameters); one optimized for long distance prefetching many functions in advance (the paper has this as fifteen functions ahead), for the cases where this sort of long distance prediction works well, along with a mid distance prefetcher (the paper suggests four functions in advance) for those cases that cannot be predicted far in advance.
These optimizations allow them to get substantially closer than RDIP to the perfect I-cache performance while using perhaps a little less than half RDIP's storage.

This is part of (2020) https://research.ece.ncsu.edu/ipc/ *1st Instruction Prefetching Championship*, which is a generally interesting collection of papers.

Branch Direction prediction

(2012) initial branch direction predictor (O-GEHL)

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.
The Branch Direction Predictor (as of 2012) was the O-GEHL version of the Perceptron predictor, which was considered to be the best predictor at 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).
Once again the patent mostly matches my description of TAGE (except for the last, Perceptron, part, which we will ignore as no longer relevant).

The basic O-GEHL scheme was well known in the academic literature, so most of the patent is about specific implementation details

These include
- The Branch Predictor tracks its *confidence level* and uses this, among other things, to indicate whether further training is needed.
(For example most direction predictors calculate a number, say maybe between -15 and 15. It's obvious that one end of the range vs the other end corresponds to taken vs not taken; but one can also consider how far the number is from the extremes, as a degree of confidence. A value of +2 might indicate that we guess the branch will be taken, but are not very confident in that guess.)
So, after a branch is eventually resolved, what sort of retraining should happen?
- Obviously if the branch was incorrectly predicted, then retraining should occur.
- What if the branch is correctly predicted? In that case, transferring the branch resolution back to the branch predictor seems just wasted energy.
EXCEPT
what if the branch was correctly predicted, but at low confidence? Then we need to retrain the predictor not about the direction, but to increase its confidence.

So think about this. How exactly do you handle this communication? If you simply report the outcome of each branch resolution back to the Predictor it's not that easy to convert something like (PC, branch result) into retraining data, for reasons we'll see.
So what the Branch Predictor actually feeds forward, along with the predicted branch outcome (all attached to the Branch instruction) is
- an indication of whether the branch was confidently or weakly predicted
- the index values (or something that can be used to recreate them) used to lookup each of the various history length tables.

This way: if the branch was correct and is confidently predicted, the ROB does nothing, whereas if the ROB determines retraining is needed (either incorrect prediction or the weak prediction bit was set), the indexes of all the entries to be updated in the various tables can be made available. If we didn't do this, how could those indexes be recreated? They are based on path history, and that path history has long since moved on by the time we are retraining the predictor!
A slightly lower power way to handle this is to save the indexes locally (in the predictor) in a circular

table that's "large enough", and just attach an index into that table to the Branch instruction that' sent forward.

- Next, what about the issue of how to handle multiple branches in a Fetch Group? A given Fetch Group may have many branches in it (in principle even as many as the number of instructions), so how do we handle the issue of performing multiple branch direction predictions in a single cycle for all those (possible, but usually not present) branches without blowing the area and energy budgets? The Apple answer (at least as of 2012) looks clumsy and inelegant, but is actually way better than it first appears.

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 alternatively assume that the table is indexed by the low bits of the *cache line* in which the PC sits. In other words, suppose we were using 14 bits of the PC. If an I cache line is 128B, that's 7 bits. So strip off the lowest 7 bits, and our table now has an index of 7 bits (ie 128 entries).
Each entry is now a row of counters – one for each instruction in the cache line, so 32 counters.
When we want to access one particular counter, we would use the appropriate seven address bits (bits 7..14) of the branch's PC to find the appropriate row in the table, and then use the five bits 2..6 to identify the appropriate counter of the 32 counters associated with this line.

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 hearing 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? Wrong wrong wrong!
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 *actually* 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. In both we're relying on hash table randomness (up till a table is about 2/3 full, negligible collisions will occur) to limit aliasing damage.

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 still 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). It works because the path information that's x'ored into the address is based only on the address of the *taken* branches before a predicted branche, and all the branches in a trace before the last one are not taken, so all the branches in the trace (including the last one) are based on the *same* path history!
Isn't hashing magical?

There are a few minor additional details beyond the above, for example the actual storage is split into two halves, so that one can always access a full line's width of storage, either by accessing the upper and lower halves of a line in parallel, or accessing the lower half of a line and the upper half of the next line. This can also be exploited in the obvious way by not firing up the other half of the storage bank if the width of this particular Fetch Group fits within only half a cache line.

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.

(2015) switch to TAGE

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*.

Consider the 2012 design. That has many good features, both as a prediction system, and as a way to, fairly easily store and read the status of multiple non-taken branches, and one final taken branch, in a trace. However the system uses no tags, whatever storage location we hit in the storage is just assumed to be correct thanks to the magic of hashing, and sparsity.
We can improve this (at the cost of some more storage and a little more energy) by adding a tag of some sort to every storage slot.
Imagine exactly the same design as before, but in each location where we simply had a counter of whether the branch (associated with this path xor PC) each time was taken or not taken, now we also add some sort of tag. The tag doesn't have to be full length, it can just be, for example, any 8 bits from the full index of PC xor path history that are not whatever bits are used to create the index.
If we do this, now, after using the index to figure out a slot in the storage, we can compare the tag at that slot with the appropriate bits in PC xor path history, and if they match we can be somewhat more confident (only $\frac{1}{256}$ chance of an error!) that we're not aliasing. If we do alias, then we use whatever zero knowledge heuristics we have chosen [eg backward branches are probably taken because of loops; forward branches who knows, but not taking them is cheaper so might as well assume that, and remember incorrect *short forward* branches can be be fixed up without resteering, just by marking the instructions in the Instruction Queue as invalid].

This is already a slight improvement, but we can make two more improvements.
First is, once we have tags, we are no longer restricted to just a direct-mapped cache in each table; we can for example make a table 2- or 4-way associative.
Assume we have, say, nine tables each using more and more of the path history bits in the indexing.
So the flow for each trace is
- calculate nine "base" indices based on the cache line address of the PC and the path history
- for each index, activate the appropriate line in the storage of the appropriate table
- based on where the branches are in the trace, look at the tags at each location in the line (eg branches at instruction 4, 7 and 15 means look at tags 4, 7 and 15)
+ but if we are two-way associative, there will be two tags at each location, so look at both of them.
- hopefully we get matches for each branch, perhaps branches 4 and 7 math on tag 0, branch 15 matches on tag 1
- based on the way of each tag that matched, look in the data storage for each of branches 4, 7, and 15 to get the prediction from this table
- for each of branches 4, 7, and 15, aggregate (in some fashion) the predictions from each of the nine tables.

It seems complicated, but each step in itself should make sense.
Now that we have associativity added to the mix, we can even choose to have different tables (associated with different path lengths) of different associativity, eg the local table and tables using minimal path history being 4-way, the tables using intermediate path history being 2-way, and those using long path histories being direct-mapped.

Once we have tags, we can do more interesting things with the tags.
An initial possibility is to add something to the tag to help with the context switching problem. This could be to include some ASID bits in the tag, or, better, to use a physical (rather than a virtual) PC as the address that’s xor’d with the path history.
Even more aggressive would be to include security information in the tag (for example two bits for interrupt vs user vs OS vs hypervisor exception level) so that eg user branches cannot interact with OS branches. This security tag scheme is in fact implemented to some extent and will be described later once we have covered the full range of branch prediction.

The second change we can make is in how we aggregate the information from all the tables, from the Perceptron weighted scheme of O-GEHL to the “best fitting” scheme of TAGE.
We also use the TAGE scheme of “usefulness bits” (see details in the TAGE paper) for deciding how to retrain the storage on each failure. (And probably, once again, on correct but “weak” predictions, as per the earlier patent.)

The patent itself is primarily about the different associativities for different tables because that’s novel, as opposed to the basic TAGE mechanism, which by that stage had been published. The patent suggests that, to save energy, there is some degree of sequential testing of the different tables rather than

full parallel testing. The details are not given, but one could imagine something like
- in the first cycle test the first three (of nine) tables. If there are no hits, or the hits all indicate low confidence, then
- in the next cycle test the second three tables,
- and then, if necessary, the final three tables.
This sequential mechanism means that good matches are preferentially written to (and read from) the earlier tables incorporating less path history, and it is because these tables land up serving many more results that they are made more associative, perhaps 4-way for the first three tables, 2-way for the second three, direct-mapped for the last three.
Sequential lookup in this way isn't exactly the TAGE algorithm (which chooses the best match across all lengths, not the best match that's "good enough" within subgroups of the lengths) but it's close enough, at lower power.

We have 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.

(2016) reduced power TAGE

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.

One can imagine other possibilities for handling the above issues.
One obvious possibility is, in addition to having different TAGE tables having different associativity, you

could also make some of the tables shorter: the long history tables are vital for some branches, but those branches are fairly rare.
Another possibility might be to use something like a Bloom filter to record the PC's of the few (but important) hard to predict branches, and based on a Bloom filter hit, jump to immediately looking up in these long history tables, bypassing the short history tables?

Indirect Branch prediction

(2013) first sophisticated indirect predictor

The most trivial indirect branch predictor 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, and we will see an example of this soon, in a 2010 patent.
However note that this scheme 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 (a reminder, this was, a long time ago, and is now 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 a hash of 10 bits of path data and some bits of PC 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.

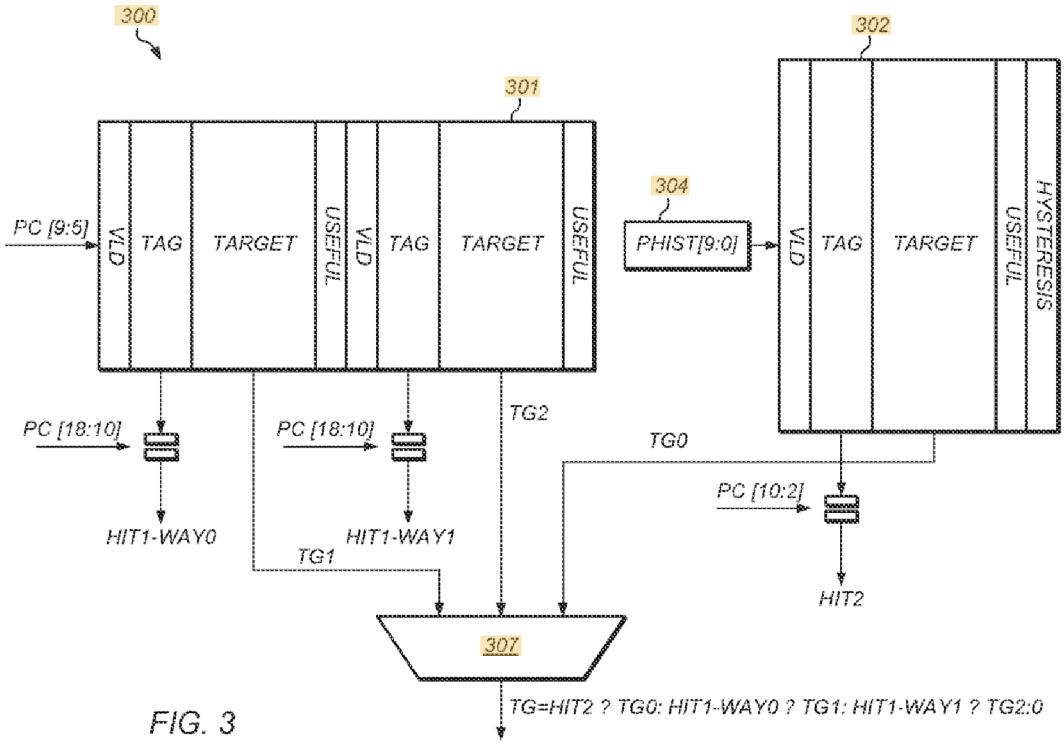


FIG. 3

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..18 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 two indexe (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 real focus of the patent is the training, which is non-obvious. We have two different predictors here, so which one should we train on encountering a new branch, or modifying a branch?

The first idea is that fields have a “usefulness” counter which increments every time they generate a succesful prediction. On the one hand, this is obvious; on the other hand, it’s interesting that this is a feature that’s commonly used in predictor “caches” but is not commonly used in “traditional” (I, D, TLB) caches....

When a new indirect branch is encountered, the initial impulse is classify it in the PC-based predictor. We do this if there is a free slot (invalid, or marked not useful); if there’s no free slot, then we place it in the history-based predictor. If a branch already in the PC-based predictor mispredicts, we mark its usefulness as 0, correct its prediction, and place an entry in the history-based predictor.

So essentially we start off assuming that PC alone will identify a particular indirect branch (for example a virtual function call that's always to the same type, at least in this call location); and the cases for which this works will increment their usefulness and be "locked" in the PC-based predictor.
For these cases where this does not work, we create a history-based entry in the hope that path history might be a useful indicator of the target (eg virtual function, or switch statement).

Now what happens when we want to install an entry on the path history side? Again if the appropriate hash slot is empty (or not marked as useful), the decision is easy.
If the hash slot is already occupied things are more interesting. Rather than just overwrite the slot, we decrease the usefulness, just as we increase the usefulness when we get a cache hit. So we try to balance the possible usefulness of a new entry against the definite usefulness of an existing entry.
The one additional feature on this side is the common Apple feature of hysteresis in a cache. 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.

(2020) objc_msgSend predictor

This patent represents functionality that we know is not in the A14/M1, but appears to be present in the A15/M2. The functionality we want to speedup is Object-ve-C messaging, described here: https://www.mikeash.com/pyblog/friday-qa-2009-03-20-objective-c-messaging.html .
That may seem narrow, but a wider viewpoint is that the problem to be solved is generic to any interpreter or something similar: a message-call "descriptor" enters a SW dispatcher which looks like a large switch statement to send the call to the correct function. To the extent that we can accelerate this, we can accelerate all such interpreters.
The machinery is somewhat flexible, as described in
https://web.archive.org/web/20210530203518/https://opensource.apple.com/source/objc4/objc4-824/runtime/objc-bp-assist.h.auto.html
and it's believed that, apart from appropriate configuration for objective-c, this is also used by WebKit

(where, again, we have an interpreter). I don't know enough about Swift internals to know if it has any sort of interpreter mechanisms like this; obviously most of Swift tries to be more virtual function based ("methods, not messages"), but it seems that Swift still uses selectors in a few places and is not discouraging their use, so???
Presumably at some point (perhaps after they are confident that this HW mechanism is robust and the details don't need to be changes) Apple may make public the configuration APIs given in the source above, and other interpreters (Lua? even Mathematica?) may also be able to use this acceleration.

On the software/language specific side we have:
- Swift's #selector will check various things to make sure that the selector you want to call is legit; Objective-C's @selector() accepts pretty much anything, so be careful!)
- The objective-C runtime converts a selector (a string) via hashing and a cache, into a procPtr, as described in the Mike Ash article.

back-end side

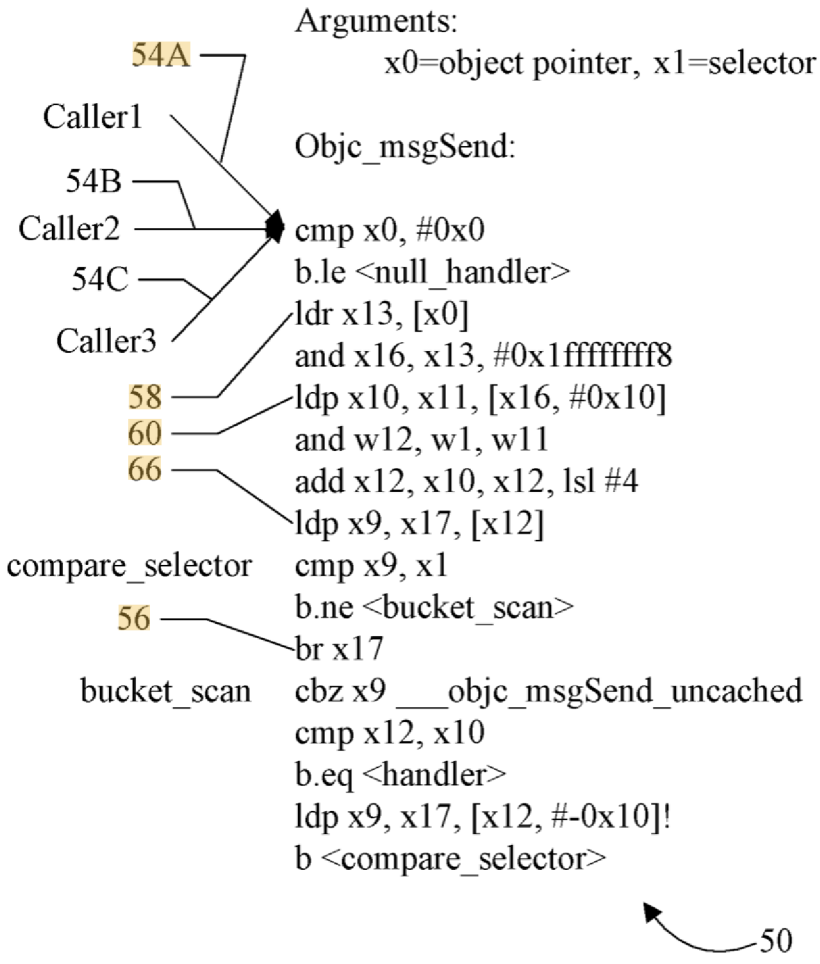
Both of these are not interesting to us; we're interested in the next stage of making sure that the branch prediction to that procPtr is as accurate as possible. That is the subject of (2020) https://patents.google.com/patent/US20210240477A1 *Indirect Branch Predictor Based on Register Operands*.

So if you think about this at an abstract level, given the framework in which objc_msgSend operates, we have a strong link/correlation between the string that is used to identify the selector (that "string" actually being an address in some global table of strings) and the ultimate function that is branched to by objc_msgSend (after hashing, then cache lookup).
So that seems a reasonably promising basis on which to create a predictor.
But it's not trivial given that Fetch Prediction (and everything related, like Indirect Branch Prediction) lives in a totally different part of the CPU from reading register values, and happens many cycles earlier! Let's see how Apple does it.

The first part to understand is that the prediction is split into two parts.
We continue to run Fetch Prediction as usual. If there's a good prediction for the indirect branch available (same selector is always called) then this machinery will work well.
If there is no good prediction available (the selector keeps changing in an unpredictable fashion, maybe the 2010 patent (where we halted Fetch until the target was resolved) is still active? Or maybe we continue with a bad prediction?
Either way the front-end is more-or-less unchanged; instead what's new is we concentrate on resolving the indirect prediction as rapidly as possible. (In a sense, this is like accepting the 1-cycle Fetch Prediction is imperfect, but we try to clean it up as soon as we can with the slower, more accurate, branch predictors). The reason this is valuable is that indirect branches are known to take much longer, on average, to resolve, than other branches, up to a hundred cycles or more, because they tend to be rare, and to depend on chains of loads at least one or two elements of which may not be in L1. For the standard Objective C case, the chain is going to be something like calling objc_msgSend, then 12 or so cycles of sequential loads, hash construction, another load, then the actual indirect jump. If we can

generate a much better prediction for the Indirect Branch at the start of this chain (ie basically as soon as the start of the code execution transitions from Decode to Map/Rename/Allocate) we can save those 12 cycles of wasted work. We still have to flush whatever was loaded between the unsuccessful Fetch Prediction of the Indirect Branch and the point at which we can predict the correct target based on the register value, but 12 cycles saved is 12 cycles saved, even in the best case scenario, and it could be many more wasted cycles that are avoided.

So if we're going to try for a prediction on the execution side of the pipeline we need to know
- what's the register whose value we are going to use as a key to predict the final target address
- what's the PC of the instruction that sets this register (presumably we don't want to get excited every time register x2, or whatever, gets modified!)
- how do we build the table connecting when the register is set to the execution of the BLR (Branch to Register and Link) to the final target register (after the process of hashing the string pointed to by x2, then looking it up in a cache, then finding the target procPtr).
Being more precise the code we want to accelerate is this;



Much of this is actually done by configuration, rather than extraction from the instruction stream! Which may seem disappointing, but is a lot more efficient.

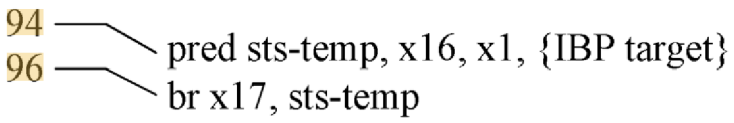
- There are some configuration registers in Decode which are set to
+ the (one) hot PC that represents the final branch-to-register at the end of the dispatch code, represented above by 56
+ the two registers whose values are used to predict the target. Ideally these would be x0 (the object being messaged, and x1, the message selector). But if you think about it, there are a lot of objects in most programs! Which makes x0 somewhat difficult as prediction source. What's much less variable is not the object but the class of the object, and that's essentially what's in x16. So in the current implementation the two dependency registers are set to x16 and x1.

There still seems some inefficiencies here!

If x13 were used as the relevant class identifier, we could get it one cycle earlier, and perform the masking (if that's useful) when constructing the hash.

Alternatively reusing x13 for both the ldr and the next and x16, x13, #0x1FFFFFFF8 would allow us (one day...) to fuse loads with lightweight ALU ops, to be performed in the LSU before the value is dumped on the results bus, saving a cycle in a different way.

Anyway with configuration set up, we now have the following flow.
Each branch-to-register (BR) instruction in Decode is compared against the special hot PC. If there's a match, then a special fake "predict" instruction is inserted before the BR (so the BR expands to two instructions). Ignore the sts-temp for now.



Essentially what the pred instruction does is "push" the required information for the predictor into the predictor. So it "pushes" the values of x16 and x1 (and the guessed target by the Fetch Predictor) into the prediction system. This is only required the first time, but something like this is required so that the predictor knows the values of x16 and x1 that will be linked to this particular outcome.

The final piece we need to understand the system is to remember that we always want to train based on ultimately correct outcomes, but we have to predict based on what just happened in the past few cycles, which is speculated. This governs the training system.

So, imagine the first time we message a particular class/method pair.
- The execution of the pred instruction will create a hash from x16 (the class) and x1 (the method selector), using the usual xor's and folding (and hopefully some bit-reversals) to create something like a 10-bit index, which will be looked up in the MOP (Method/Object Predictor) cache. Since this is the first time, there will be no hit.
- A Training entry will be created.
- At some later point the Indirect Branch will be *resolved*. This is the point at which the Indirect Branch is executed, and the value used by the indirect branch will be trained "into" the created Training entry. That Training entry was created using the target given by the Fetch Predictor, and that will either be "rewarded" as being correct, or "punished" and the new value placed in training.
- But that's not the end of the story, we have to wait until the Indirect Branch is *retired*. That's the point

at which we know that the Indirect Branch was ultimately correct (this whole path of execution could have been speculative based on a predicted branch 200 cycles ago that turns out to be false). If there was such an early misprediction, the Indirect Branch will be flushed, os its *retirement* is when we know everything happened according to plan, and the value stored in the training entry is copied over to the MOP cache.

Then the next time we hit the pred operation, once again we'll convert x16 and x1 into a hash, we'll look it up in the MOP, and we'll immediately redirect Fetch if necessary.
However the training entry is still allocated, even in this case, and the training entry will hold the correct final target as before, so that the correct value can be updated into the MOP if required.
Honestly, I'm not sure this delaying until the BR *retires* is such a good idea! Normally this delay in updating a branch (or other) predictor makes sense for the reasons I've described; but in this particular case, the connection we want to establish, the link between (x16, x1) and x17 almost always remains just as valid regardless of whether the code was or was not on the correct path, and delaying the update seems somewhat pointless.

There are a few other details in the system that are technically interesting but irrelevant to the larger structure so I'll omit them to avoid additional confusion.
In terms of the larger goal, two issues we will return to.
The first is that the timing may seem tight, even pointless, given that the pred instruction is placed right before the br instruction. However remember this is an OoO machine. Each instruction will execute as soon as it can, so the pred will execute as soon as x16 is available, before the two sequential ldp's and their intermediate instructions.
The second is that for the sake of training, we need to ensure that the pred instruction occurs before the br instruction (something that's highly likely but not *guaranteed* since they do not have any sort of direct dependency). To solve this, we create a fake dependency between the two via the sts-register. This is a conditional register that we set (the value is unimportant) in the pred instruction, and read (again the value is unimportant) in the br instruction. Its sole job is to ensure the ordering, that pred always ocurs before the br; and we care about this for the sake of training.

Ultimately this whole scheme has elements of value prediction and of criticality prediction. However when you think about it, it really has to be its own special thing.
The connection between the (x0, x1) or (x16, x1) input registers and the target address is strong, but is separated by so many instructions that no practical value predictor could learn it.
Likewise, while criticality could establish a chain of dependencies backwards from the br through the lookups and hashing, it could not bypass several slow steps in this chain the way the MOP can.

front-end side

The above is all very interesting, but we still waste a whole lot of cycles in that gap between what the IBP (Indirect Branch Predictor) predicts and what the MOP predicts. Can we do better?
Well, if you think about it at the most abstract level, there is a design flaw (or perhaps, better, a design mismatch) between the code and the IBP.
The IBP, like most such predictors, assumes there is a link between the PC of the `BR x17`, and the target of the indirect branch (ie most of the indirect branches that occur at this PC are to the same

target, or at least follow a pattern).
That assumption is true for many virtual function calls, but not for a "dispatcher" type `BR` where every call is funneled through the exact same `BR x17`. ie there's essentially zero predictive information in the PC of that `BR x17`.
But it's not hopeless! The call chain looks something like
PC1 call to objc-messagesend
PC0 `BR x17`
and while PC0 has essentially no predictive power, PC1 may have some predictive power (either directly, or in conjunction with a branch history).
So if we could somehow, for these particular indirect branches, use PC1 rather than PC0 for the indexing, history and tagging of the indirect prediction, we'd be a lot better off!
That's essentially the content of (2020) https://patents.google.com/patent/US11294684B2 *Indirect branch predictor for dynamic indirect branches*.

We begin by confirming that the IBP is a TAGE variant, something like ITTAGE or COTTAGE, but then get to the meat of the patent, which is essentially as I described. We've already configured one special PC as the PC of the hot indirect branch, the `BR x17` branch. So all we need to do in the pre-existing Fetch/Branch Prediction machinery is continually store both the PC current of the current function call and the PC of the previous function call, then usually we use the current PC except when that current PC equals the hot PC, in which case we use the prior PC. That's a quick hack (and some internal modeling and thought may eventually turn this into a more principled design) but it's a good simple solution.
(In theory the branch path should be good enough to handle this.
In practice there are two problems.
First if PC0 is information content free, then we're creating our indexes and tags in a sub-optimal way; ie we are wasting a lot of space; so let's fix that, regardless.
Secondly the construction of a path history is forced to use just a few bits xor'd into the history. This is adequate (and all that is practical) when we are using the path history as one of many such histories; but it doesn't work well when we require the path history to be giving us the *full* information content of PC1, the previous caller PC.

A more principled design might do something like track the PCs of multiple indirect branches that tend to be unsuccessfully predicted, having a table of these (I would guess it wouldn't have to be large, maybe 4 or 8?; even apps that might have more than one such dispatch point won't have too many). You could perform this comparison cheaply with a simple Bloom filter.
Then, at the point where the current design switches to using PC1 rather than PC0, the new design would use PC1 (the prior call point) for any PC that matches in this small table of troublesome PCs.
This is basically a bet that even when the current PC has no useful content for the purposes of IBP, the PC of the prior call site may have some useful content.
It's possible that this scheme could land up confusing prediction of the PC1 branch with prediction of the PC0 (using PC1 as index) branch. This could be handled via something like not'ing the value of PC1 when it is used as the "earlier" PC, which would allow predictors for both PC1 (directly) and ~PC1(indirectly, as an index for PC0) to co-exist in the prediction machinery.)

Loops

Now let's talk Loops.
A naive view might be 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 less energy reduction.

However, before we discuss details, let's think about loops in more detail, informed by Macroscalar and the similar POWER proposal.
Consider, on the one hand, a simple loop that adds together two arrays of four integers, vs doing the same thing using a SIMD instruction set. What's the difference (ie what does SIMD, as a "miniloop" get you over using a normal loop?)

- we save instructions. SIMD handles
+ incrementing a counter (1..4 over the items to be added)
+ initializing and testing that counter
+ the messiness of pumping the same instruction through the pipeline multiple times

- this translates into saved energy (Fetch energy, register file energy for the counter, pipeline control energy)

- it's also trivially clear that the separate operations (add, or load or whatever) are independent, which makes scheduling trivial, and uses many fewer OoO resources (registers, ROB, etc)

Suppose we use technology like a loop buffer. How much of these desirables in principle can we claw back (for short SIMD length loops, or in general)?
- we would expect even the most basic loop buffer to store the loop body in special storage, and to apply an appropriate predictor for the loop test, so we can avoid Fetch energy, and misprediction of the loop ending condition
- we can have the loop buffer placed *after* Decode, so we avoid the costs of Decode
- we could even move some of the instruction interdependency analysis performed by Rename/Map into state held by the loop buffer. Ideally the simplest sort of loop (equivalent to a SIMD instruction) has no such dependencies.
- *in theory* we could imagine a smart "loop analysis" that can see that a variable is being used as an "index" variable within a loop, and could use a special type of register for this purpose, rather than the costs of general register renaming.
- but at this point we hit the reality that we cannot index registers. This is where the dream falls apart, we can't write a loop that's the equivalent of loading physical registers r[0]..r[3] and r[4]..r[7] as two miniloops, followed by a miniloop that adds r[i]+r[i+4]. That's where SIMD has an insurmountable advantage, unless we augment the ISA with something like the POWER suggestion.

OK, so much for that dream. If we can't do that, what can we do with loops?
We'll see that there are two waves of patents a first set in 2012 that are "adequate", with substantial improvements in 2014.
The first thing we have to be able to do is detect a loop.

(2012) loop detection, basic loop buffer, basic in-loop branching

Let's start with "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) 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.
Detecting this is simple.
- When we encounter a backward branch, we see how far back we branched.
- If this is small enough (so the loop could fit in the buffer) record the instruction count and start detection mode
- Next time through check that
+ there are no branches in the loop body
+ the loopback distance is unchanged,
and increment detection mode, otherwise reset everything.
- If we increment detection mode successfully enough times (say three times) then we assume we have a loop and start treating it specially.

This basic scheme is covered by (2012) https://patents.google.com/patent/US9557999B2 *Loop buffer learning*, and is doubtless no different from any other CPU.
Note that (as I've described it, and as the patent describes it) there is no learning/memory of any sort across multiple re-encounters of this loop.

The main slightly nonstandard points are:
- The loop buffer apparently lives *after* Decode. This means the buffer holds μops rather than ops, and the cost of Decode can also be avoided.
(It's unclear if this is retained in later designs. Later patents seem to show the loop buffer placed before Decode. One advantage of is that you can run the loop out of the instruction queue sitting before Decode, ie you can avoid moving the instructions to a dedicated Loop Buffer. So you save that energy; but you pay more energy for the repeated Decode.
So???)

- 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.
However this is not as powerful as it sounds, because the the loop body has to be *invariant*, ie the instructions (and so conditional branch outcomes) must be the same from iteration to iteration, so these conditional branches are allowed.
These details result from how the loop is discovered.

(a) On the backward branch we start counting the instructions till we hit the backward branch again. (As opposed to recording the target of the backward branch.) This is an initial filter that the loop instructions do not change (since if conditional branches were taken in different ways in the loop body, then this instruction count might come out differently).
(b) While tracking the loop, we have a table that can record up to 8 conditional branches, recording specifically how many instructions from the start of the loop till the branch.
This is a second (more powerful) way to check that the conditional branch outcomes did not change during the loop execution.
(c) We only allow forward branches, because we're using a backward branch as the indicator for the loop termination point.
Honestly the details look very much like prior art workaround rather than being especially clever or interesting in themselves.

So we can allow for some degree of conditional in the loop but they must have an unchanged outcomes. Essentially we can handle cases like `if(exceptional condition){...}` where we expect the condition mostly not to occur; but not even a basic "do this if index is even, otherwise that if index is odd".

The patent is somewhat ambiguous as to whether a simple call+return (ie a short function call inlined by the hardware rather than by the compiler) would be allowed. The way it's written, legally I think that would be covered, but it doesn't seem to fit with the way the rest of the patent works. Certainly (for obvious reasons) indirect function calls disqualify a loop.

The Loop Buffer is basically acting like Fetch Prediction, in that it is generating a stream of instructions that flows through the OoO engine. So, just like Fetch Prediction, if there's a misprediction (ie one of the branches that was assumed unchanged in fact changes its direction) this will be caught when the branch actually executes. Recovery will, as usual, flush the machine and resteer Fetch to the correct address, but in addition the Loop Detector will flag the backward branch as not to be associated with a loop (for at least some period, details not given; perhaps only until we switch to detecting a different backward branch?)

(2012) loop packing

Even with this basic loop buffer, we can get some slight performance benefit from it.
One minor flaw in the Fetch scheme we have described so far is that only one Fetch Group (ie trace of instructions up to a taken 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.
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.
You might expect the compiler should be doing this, and yes, perhaps so. But memory is always an issue, and Apple recommends all code (unless there's a good reason otherwise) be compiled as -Os which, among other things, will not aggressively unroll loops.
So, if you can fairly easily get the benefit of unrolling in HW, why not?

The name of the patent gives it all away: in an advance on the traditional basic loop buffer, once a loop is detected, and if the loop is short enough, the loop body will be replicated in the buffer as many times as will fit. This bypasses the Fetch limit.
Imagine, for example, that the loop body is 4 instructions long, with the fifth instruction being the backward taken branch. Then the loop buffer contains a stream that looks like
I0 I1 I2 I3 B, I0 I1 I2 I3 B, ...
And we can just dispatch instructions from this as wide as the machine will allow (6 at a time in 2012, 8 at a time now). So we can dispatch (I0 I1 I2 I3 B, I0 I1 I2) as one unit, whereas Fetch would have had to limit the Fetch Group to (I0 I1 I2 I3 B) each cycle.
This is nice, and seems like a good idea, but what, sadly, isn't covered (in this or the previous patent) is the question of how we decide to exit the loop! To judge from the patents, we just obliviously loop continually out the loop buffer, until execution signals a misprediction.

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); so able to hold about 96 ops (well, 96 uops, but usually there is a 1 for 1 equivalence).
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 IDQ was disabled via microcode update in 2017: http://gallium.inria.fr/blog/intel-skylake-bug/ SKL150 bug fix.
The hardware on Intel that *checks* for a loop is called the LSD, Loop Stream Detector; but the instructions flow from the IDQ.)

2014 widespread improvements

The basic loop buffer is limited to easy loops, and we've described various real or at least apparent problems with the 2012 scheme including
- no memory of previous loops
- no intelligence applied to the loop exit condition
In 2014 we work on these, and on much more difficult loops (either variable branching, or difficult exit condition).
We solve these, in part, by creating multiple instruction feed mechanisms that are intermediate in complexity between the basic (no varying branches allowed) Loop Buffer and the full Fetch Prediction+Instruction Cache.

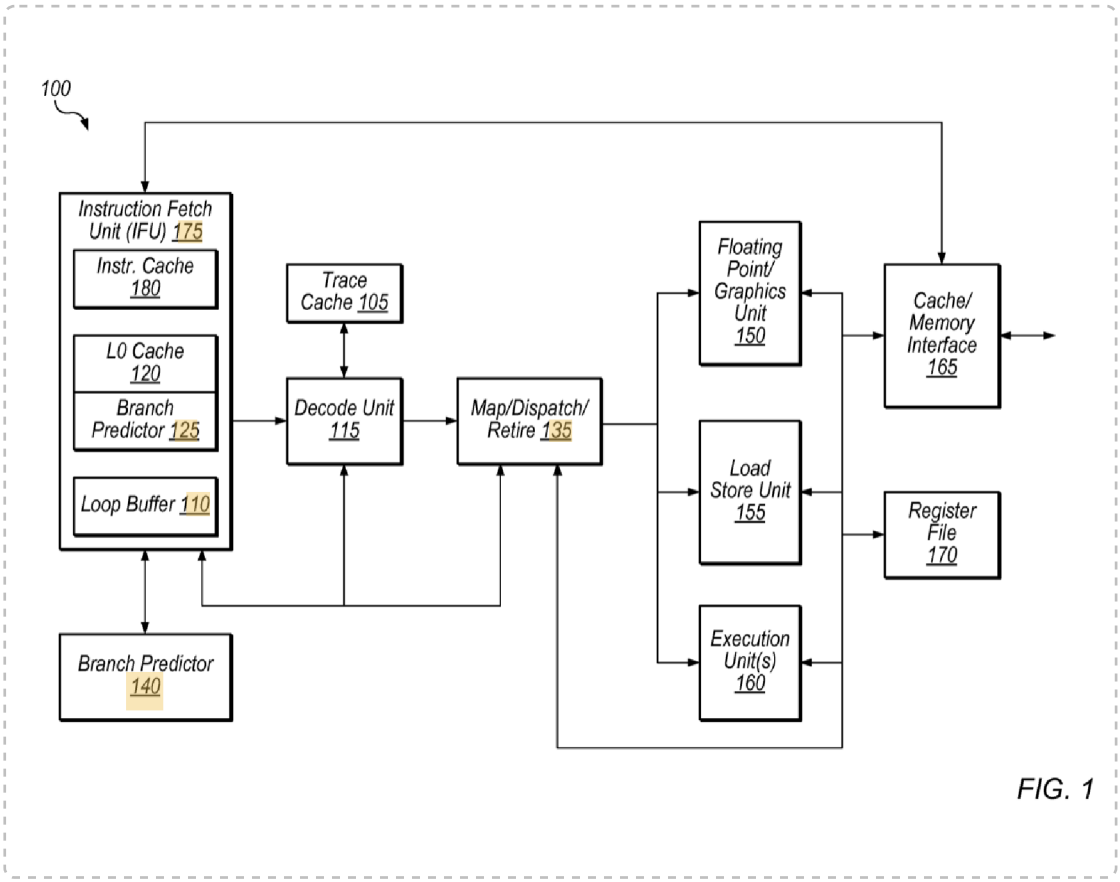
(2014) L0 I cache

Suppose we have a loop that involves some degree of branching within the loop body.
A Loop Buffer has no prediction mechanism and so is not a good match for such code.

A Trace Cache might work, but will need an associated predictor.
Most complicated (and powerful) is to use a simplified version of the full Fetch machinery (simpler Fetch Predictor reading from a small L0 cache).

(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 envisages being able to sequence instructions from four different sources
- the full Decoupled Fetch+I Cache+complex Branch Predictor scheme
- an L0 cache (smaller and lower power than the I cache, probably direct mapped) with Fetch Sequencing controlled by a simpler Branch Predictor (probably just a small local predictor with minimal history, something like gshare)
- a loop buffer
- a trace cache



As already discussed, the placement of the Loop Buffer seems strange. Should it be below the Decode Unit, somewhat analogous to the Trace Cache, or even connected directly to 135?

- It's clear that normal Fetch happens when executing new code, when jumping to far away functions, and basically for all "non-local" execution.

The other three seem only relevant to loops, ie “very local” execution.

- We are not given details as to when the Trace Cache is used.

- The Loop Buffer is for loops of the form we have already seen, ie “small enough”, and with no, or only “stable outcome” branches.

- The L0 cache is for loops that involve some degree of (mostly predictable) branching.
The loop learning scheme appears to be much the same as in the 2012 patent, but if the loop fails some of the 2012 criteria (too many branches, variable branches, too large) it may still be eligible for the L0.

The patent does not give many details as to how the L0 system works.
The L0 Fetch scheme seems to be something like the original (direct-mapped) Fetch scheme, ie essentially a table of target addresses, only shrunk down not just in obvious ways, but also using fewer bits to, for example, indicate the address of targets, since all targets are required to be in the L0.
As we’ve seen with the normal Fetch, there is some attempt to detect errors as early as possible, by marking entries in the L0 cache (and, as I understand the description, also in the Fetch Prediction table) as invalid, and reverting to normal Fetch if these are encountered.
The scheme is fairly general (it can handle things like nested loops and strange backward branches within a loop [assuming these cases can be detected]) but, as a consequence of this generality saves limited energy.
We are not told the size of the L0, but other patents in this collection suggest that it’s much the same size as the Loop Buffer, which is much the same size as the Fetch Queue; so maybe 8 to 12 lines each holding 16 instructions, something like that.

If we’re willing to limit ourselves to “normal” loops, with just forward flow control we could have the loop storage structured something like a long trace (including even function calls and returns), with another simple gshare-style branch predictor attached to it.
Then prediction would consist not of predicting a sequence of PCs, but of predicting *which elements in the conditional runs of this trace* (the parts in between braces of an `if(){stuff}`) *should be marked as invalid instructions*. These invalids can be predicted every cycle then used, in the next cycle, to decide which instructions are (and are not) moved from the loop storage to the next stage in the pipeline.
This seems to be worth considering as either yet another instruction source mechanism, or as a way to make the Trace Cache system (however that works...) much more powerful, able to cope better with conditional branches, while also using a lot less storage.

The Intel paper (2014) https://citeseerx.ist.psu.edu/viewdoc/download?-doi=10.1.1.1069.9568&rep=rep1&type=pdf *Improving the Energy Efficiency of Big Cores*, for example Figure 5, gives an interesting description of this same sort of tiering a loop through various possible buffers and caches as it grows larger, though the precise details relative to M1 are, of course, very different.

(2014) loop table and tiered loops

(2014) https://patents.google.com/patent/US9524011B2 *Instruction loop buffer with tiered power savings* tells us that a "previous loop table" has now been added to the system. This table records some obvious things like the loop, start, end, and iteration count, and something about the types of branches in the loop body and how variant they are.
The table is both rich enough to record some degree of nested loops, and dynamic enough to update itself if features of a loop change (eg loop count changes, or what appeared to be a constant branch pattern changes).
The language of "table" suggests this holds multiple entries, which in turn suggests that we can immediately shift to loop mode when a loop is re-encountered, and can know which type of loop mode to switch to, but strangely the patent does not even mention this possibility. It's unclear whether "table" refers to multiple loops, or the fact that data about multiple branches is stored so as to detect nested loops.
One would also hope that the iteration count is persistent, in the sense that it can be used on future encounters to know when to exit the loop (if this count remains unchanged) but again the patent only discusses using this in the context of training the first time the loop is encountered.
I honestly don't understand why they're working so hard to capture weird and unnatural loop patterns before optimizing the (apparently) low-hanging fruit of basic, nested, count-based loops.

The strategy now, is,
- as soon as we see an apparent loop, perhaps after just two or three iterations, we switch to the most basic loop mode. This puts Fetch and the I-cache to sleep (but Branch Direction Prediction stays awake) and we run the loop out of the Instruction Queue. (This is the same as Intel does, as we described above with Skylake.)

- ideally we run that way for 20 iterations or so (if branch prediction requires, we may have to break out of this mode for an alternative fetch, but as described, minor short forward branch mispredictions can be handled by marking instructions in the Queue as invalid, without requiring Fetch to wake up). After 20 iterations or so, we can then decide whether to move the loop to the lowest power Loop Buffer, or to the lower power L0.

- presumably we keep gathering statistics as the loop executes; and next time we encounter it we can immediately respond appropriately using the values in the table; but the patent is silent on that aspect of things.

Many aspects remain unclear! For example, while the system supposedly can handle nested loops, it's not clear if it does so optimally (for example storing the inner loop in a persistent fashion in the Loop Buffer, while running the outer loop from within the L0).

An interesting technicality stressed by the patent is that under normal conditions, Fetch will deliver instructions directly to Decode, if the Instruction Queue is empty, or to the Queue (if it is not empty). However if we wish to enter loop mode, we need to modify Fetch slightly to force the loop body instructions into the Instruction Queue even if they would naturally have bypassed it to go straight to Decode.

So this gives us definite progress on complex and nested loops, and possible (but somewhat grasping a straws) progress on remembering loops from one encounter to another, and remembering the loop exit condition.

(2014) complex exit conditions

We get some progress on complex exit conditions (no word on easy exit conditions) with
(2014) https://patents.google.com/patent/US9471322B2 *Early loop buffer mode entry upon number of mispredictions of exit condition exceeding threshold*.
The target is cases where the loop exit is unpredictable (think for example of something like strcmp); and the concern is both saving energy and the training of the branch predictor.
How should we handle this training when executing a loop?
Some basic points are
- if we execute the loop while running the standard Fetch + branch training machinery, we are soon going to flood our path/branch history with repeats of the fact that the backward branch was taken, and whatever branches occcur inside the loop body. That seems bad.
- if we segregate the loop to separate control (either the loop buffer or the L0) the loop buffer is decoupled from Branch Prediction, and the L0 uses its own predictor with its own storage. That seems good.
- but the standard Branch Predictor is probably capable of predicting the loop exit condition under many circumstances (certainly for small counts that repeat from one loop iteration to the next, although this is not the most efficient way to do such a prediction; and even for cases where the loop count is perhaps correlated with an earlier branch). That seems like something we'd like to take advantage of.

The resolution Apple has chosen, at least as of this 2014 patent, is
- we run the loop out of its special storage (the loop buffer, or L0) and shut down what we can
- but we keep the Branch Direction Predictor alive, to (hopefully) predict the exit condition
We do this for as long as it makes sense, which means for as long we have not saturated the Branch History vectors. (Saturation is essentially when we've iterated enough times that the information in the history vector stops changing from one iteration to the next.) After this many iterations, further training is pointless, and the prediction of the exit condition will not change, so we might as well shut down the Branch Direction Predictor.

We can do slightly better than this if we know that the loop has an unpredictable exit condition (as I said, something like strlen). In that case, tracking the Branch Direction Predictor to help with the exit condition is pointless, so we might as well shut it down right away. So the fact that a loop exit is unpredictable (ie was mispredicted by the Branch Direction Predictor) is noted, and if this happens a few times, then right from the start we shut down Branch Direction Prediction.

As usual there are unanswered questions!
This patent requires persistence of some information associated with the loop from one loop instance

to the next. This is done by something called the Early Loop Buffer Mode Table. But it's unclear if this table stores anything beyond what's necessary for this early shut down for the Branch Direction Predictor; in fact it's even possible that this persistence only exists within the context of an unpredictable inner loop within an outer loop that's also being handled by special loop control.

The patent also suggests that we use the entire might of the full TAGE Branch Prediction Machinery to handle the prediction of the branch exit condition which, OK, will work to some extent, but seems a whole lot less optimal than various schemes one might imagine for a loop exit predictor specialized for the task and based on common cases like, eg decrementing or incrementing an index. I'm guessing that branches within the loop body do not attempt to retrain the predictor (that seems just horrible) only the loop exit branch; but honestly it's not clear.

Unfortunately the trail of loop related (and trace cache related) patents runs dry at this point! It's hard for me to imagine that the loop machinery has not been substantially improved; there are many academic ideas in this space; and it's clearly (done correctly) the single biggest hammer you have in continuing to reduce the cost of Instruction Fetching and Decoding. But for whatever reason, the group inside Apple working on loops appears uninterested in filing patents.

One thing that becomes clear from looking at these loop patents is just how much small ISA modifications can improve (or hurt) attempts at performance and low energy.
We've already discussed how Macroscalar type loops can help, but another sort of example is having the ISA encourage a certain type of "structured" loop. For example POWER's use of the count register and the `bdnz` instruction make it a lot easier to see the outline of a loop, and to know when the loop will end.
There might be value in doing this for ARMv8 in a simple compatible way. An expert would have to look at the details, but two related options that spring to mind:

- You may know that there is an ARM HINT opcode takes an immediate value between 0..127 that describes the "type" of hint. These are used, for example as the standard NOP, or YIELD or WFE.
What if, as the first instruction in a "standard" loop body, we used a new sort of LOOP_START hint, so that the Fetch/Decode machinery would know to immediately slide everything from this point until the loop exit point, into special loop handling machinery. Might that be worth doing?
(Ideally this hint could carry extra data, like the loop length, but there's not much immediate space available; I think the best that's feasible is to have the first backward conditional branch be the indication of the loop exit point.
Of course this might not cover everything that could be called a loop – this is a performance optimization, and we only need to cover 90% of useful cases, not everything imaginable.)

- We could also rule that the presence of this hint is a promise (broken at the cost of wasting energy and reduced performance once the predictions fail) that the loop follows one of a single or a few standard, easily detected, patterns; for example the simplest pattern might be the loop makes use of

+ a single counting register, always the same, say r2,
+ always counting downward so always exiting at r2==0, (so similar to the count register)
+ possibly with rules about the r2 modification, then the r2==0 test, are always the last instruction [eg `SUBS`] before the loopback branch, which always loops back to the address of LOOP_START+4 (so similar to `bdnz`).

This would be backward compatible, but could allow both power and energy savings for CPUs that understood what the LOOP_START hint was indicating and could
- immediately switch to low power branch storage, and
- make use of a low power branch exit predictor (ie a simple counter)

The future of loops and branches

Right now (and basically for the past 25 years or so) branch prediction has been driven by control flow. That is, branches are predicted based on previous branch history (either at this PC or earlier in time). This has clearly worked very well but our discussion of loops has shown some of its limitations.
For example if you are tracking history 32 branches back, and you have a loop that is taken 25 times, that can be captured in the prediction machinery; but a loop that is taken 37 times cannot be captured by this machinery.
A second example is the *Branch Prediction Is Not A Solved Problem* paper, pointing out the dire performance consequence of branches that are "unpredictable" and depend on a long latency load so are slow to resolve.

In both these cases, the issue is branches that are clearly "data-dependent", in the sense that predicting them is poorly handled by using previous branch information, but could perhaps be well-handled by using data available in the CPU. This gets us to the most leading-edge ideas for how to handle branches, namely (2021) https://hps.ece.utexas.edu/pub/PruettPatt_BranchRunahead.pdf *Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches*.
The biggest idea of this paper is to allow the branch prediction machinery access to the abilities of the rest of the CPU, rather than being confined to just looking at tables of previous branch history. In slightly more detail, the idea is that
- we provide a full computation engine for branch prediction. This could be a little side mini-CPU (probably easier) or it could involve sharing the main CPU (and appropriately tagging instructions and results so they don't confuse and pollute the rest of the CPU).
- secondly we need to detect branches that are constantly being mispredicted
- third we try to capture the "important" instructions that ultimately generate the branch decision, and "pre-execute" them in our side CPU.

Obviously this is not going to solve every difficult branch but it solves about half of them (for the exact configuration of the paper). For example the case of a loop count that is large but stored in a register that just counts down is trivially captured (though there are more energy-efficient ways to do so).

Likewise for the case of a branch based on a load that we could perhaps execute early.

Just as obviously, this requires a huge change to existing designs! I expect it'll be ten years or so before we see it in a CPU, as other academics look at the idea from different angles and consider optimizations or easier ways to integrate it into existing designs.
On the plus side, the sort of machinery required to detect the "important" instructions that result in a branch decision is pretty much the same machinery required to track the criticality of instructions, so there is a path from today, through criticality (CPU core only), to Branch Runahead (propagate some of the criticality info to a separate branch pre-executor).

Return Address prediction

introduction

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 might 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 indicating when a function return address needs to be pushed onto this miniRAS

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 slower, 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 after a context switch).
The MiniRAS is not just smaller than the Execution RAS; at any given time it holds different content. 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.

The big obvious problem to be solved with a RAS is of the stack becoming corrupted under misspeculation. 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 then discover the misspeculation and redirect Fetch to the correct spot. That will leave the inappropriate return address on the RAS.
The MiniRAS is small enough and lightweight enough that I suspect it makes no attempt at correction. Suppose the Return Address Predictor notices a mismatch between what Fetch has done and what the Execution RAS says: The 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, as a quick fixup.

What about errors in the Execution RAS? As usual there is a range of ever more complicated options. To get started, think about what a stack means as a 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 *stack* is that ToS can only be incremented or decremented by 1.

The first level of correctability is to ignore that rule. Maintain (somewhere...) the value of ToS. Then, in the misprediction scenario already described, what happens is
- we know the value of ToS pointing to the correct location in the array at the point before misspeculation
- the incorrect function call dumped an extra address on the stack, and incremented ToS and so
- 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 ToS.
- 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 have described so far 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 recoveryToS)
- 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
- this 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 background, 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 have 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.

If you really want to understand the Intel scheme (which may now have changed...):
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 increment 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 recovery-ToS 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*.

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.

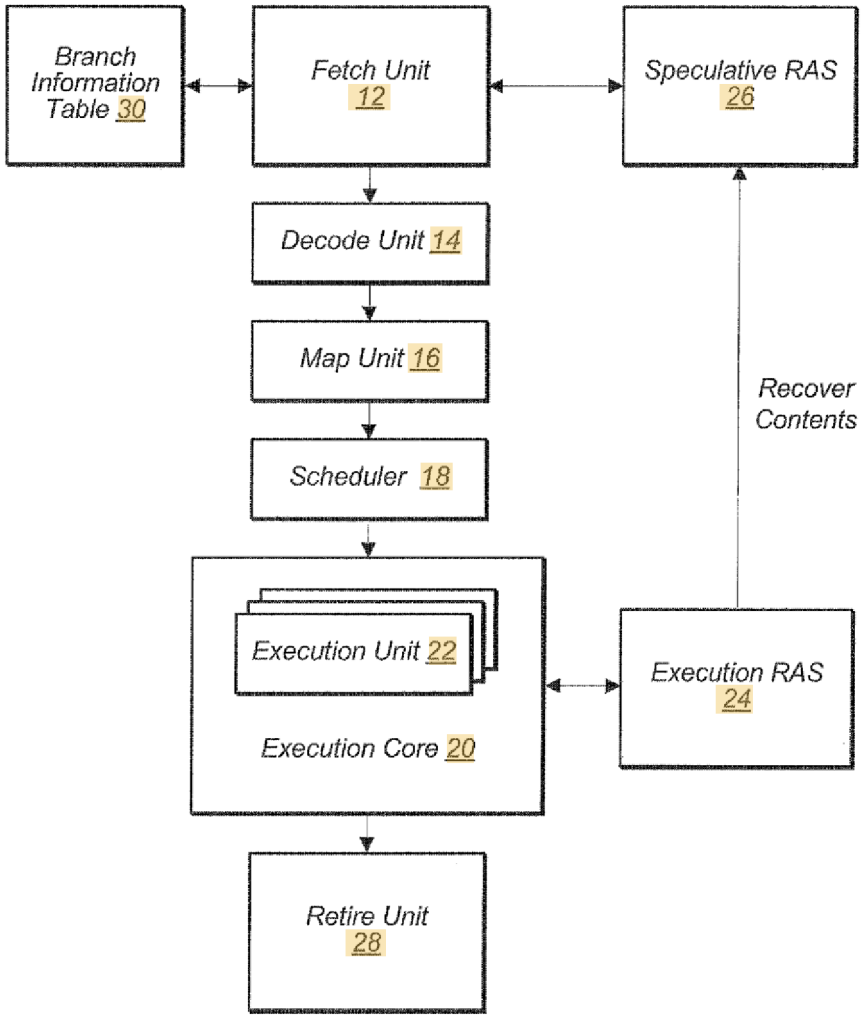
Before we discuss what Apple does, let's return to an issue we skimmed over. We need somewhere to store the the recoveryToS for each call and each return.
I described doing so in the RoB slot, but that’s not ideal; rather us a pattern we have seen frequently before – don't make a general structure larger, use indirection to store data in a task appropriate

structure.
The issue we are seeing 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.

(2011) initial RAS design (somewhat like Intel/1990s ideas)

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 regards 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, 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 they are trying to deal with is Replay.

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 described this in great detail for standard integer or FP instructions, including how they can be re-executed multiple times because their values are always written to the same physical register, so the final execution after multiple replays will overwrite whatever was incorrectly written earlier to that register.

Now you might not think that there could be any dependencies between a branch and a load, but there are, 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 by 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 into 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 a load, which feeds into Replay of dependent instructions.
-- 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) *and* it *also* 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 when 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 that, under very specialized 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 deals 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. Apple

says: Forget that and forget the Alloc index;, treat the structure as a pure linked list, implemented with a finite array of node-sized entries (say 90 or so), where each entry has a “valid” bit.
Now every time an entry needs to be allocated we can work our way down the buffer (with wraparound at the end of the buffer) looking for an invalid entry, which we use. Of course 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 slot as 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 in part 1...
(There is still the issue of what if you go really deeply down a set of nested calls? At some point you just have to 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, the whole miniRAS-sized stack at once. I guess that's different enough for a patent!
Don't take this part too seriously, however, because we move on to:

(2013) a substantial redesign

(2013) https://patents.google.com/patent/US9405544B2 *Next fetch predictor return address stack* clarifies and improves much of the machinery.

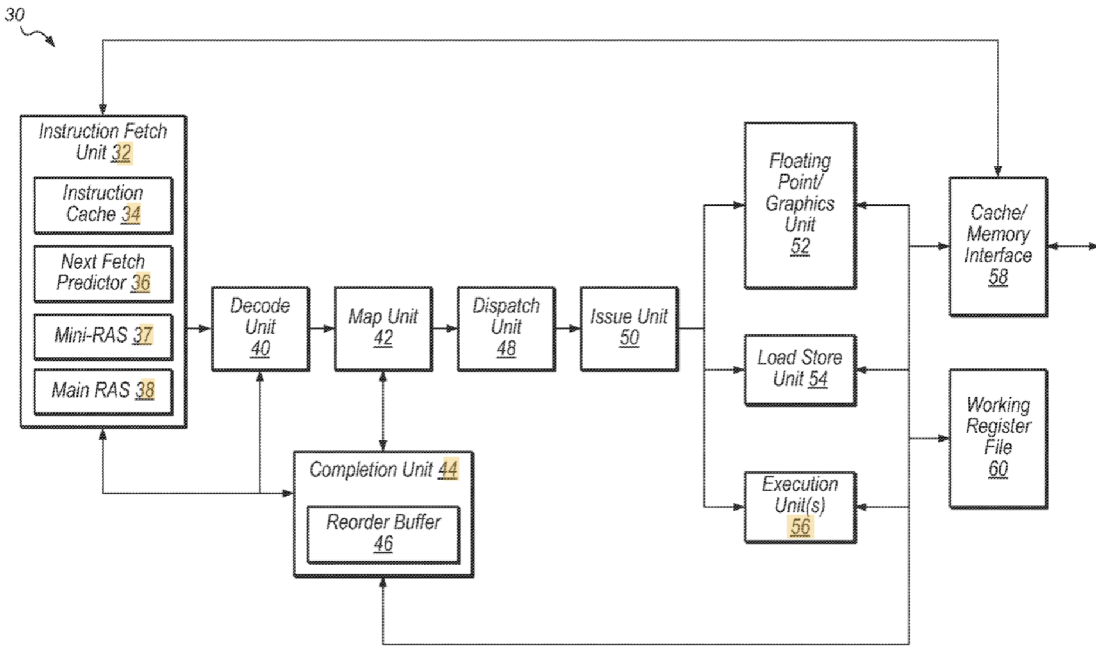


FIG. 2

The first nice improvement is that the main RAS (what was previously called the Execution RAS) is now maintained at Decode rather than at Execute.
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 does not pop or push the stack too often.

Secondly we see a use of the predecode bits. Suppose that Fetch accesses lines in the I-cache that it has not seen recently, and so are not part of the Fetch Predictor. Even so, 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 (a predictor, or execution) can inform us as to the predicted target of the 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 really 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 just use the value at the top of the main RAS.

In a way it's like the 2012 design was a somewhat stripped down version of the Intel design, which was an industrial implementation of late 1990s ideas.

The 2014 design reconsiders and drops many of the Intel/1990s elements, rethinking the problem from scratch in light of the particular details of ARMv8, as opposed to x86. (x86 has all sorts of crazy concepts that, even if rarely to never used, have to be handled, like far calls and calls through task gates; this may mean that much of the streamlining that's feasible on ARM may not be feasible for them.)

Security aspects of branch prediction

(2015) first, somewhat clumsy, version of PAC

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; and so 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.

(2016) final version of PAC

I'm not sure if the above idea was ever implemented in HW; it requires some compiler modification and somewhat increases the size of code.
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 (rather than being placed [somehow...] in NOP-type instructions before or after the call instruction); so now it's pretty much always there, and always checked as part of logic that reads and writes the RAS.

(2018) security tags to limit adversarial training of predictors

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 found this table entry 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. Ultimately this means that an attacker cannot train the predictor so as to compel 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. This is unfortunate, and perhaps there is a way to avoid this for most shared libraries, while retaining it for particularly sensitive libraries?)

These same ideas are used in subsequent predictors, for example the 2020 MOP cache says that the cache includes a security tag.

Ways of using pre-decode

(2014) store in cache partial sum of branch offset with PC

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.
- we now 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. This can happen because shared libraries (like most ARMv8 code) are compiled as position independent code; and because for ASLR or other reasons, the shared libraries are placed at different address offsets in different processes.
- 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. 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. When 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 hashed 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*.

(2014) undefined instruction recoding

So we've seen two interesting things predecode can do
- mark branches of various types
- compute branch target low bits
Is it good for anything else?

We've already seen (2014) https://patents.google.com/patent/US20160011875A1 *Undefined instruction recoding*, which suggest a not especially interesting option – detect all the possible *undefined instruction* encodings in an instruction and replace them with a single "undefined instruction" value. This simplifies the real Decoder, which only has to detect (and generate a trap for) a single "undefined instruction" encoding.

additional possibilities for pre-decode

simplify irregular decode

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 format, etc.
Of course what's worth doing depends on details we don't know about the pain points in Apple's A64 decoder, but this does suggest a way to reduce some of the energy costs of Decode.

A version of this is to remove xzr from many instructions. For example it's cute that the ARMv8 `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 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, and unexpectedly, for some situations, like MOV, and maybe this is a casualty of such pre-decode?

improved fusion

Another possibility is to mark predecoded instructions by "class", so that later Decoding can more easily fuse instructions that belong to compatible "classes".

compressed code

A further 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 (but more powerful!) 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. Superficially this would seem to require a separate I L2 (since I cache lines would be wider), and this is not necessarily a bad thing; but another option would be to have this predecoder kick in on the transfer of lines between L2 and I L1.

An even more extreme version of this might be to have this predecoder capable of expanding compressed code, so that compressed lines of instructions are stored in DRAM, SLC, and L2, only expanding to full lines in L1. This would be something like IBM's CodePack: (2002) http://web.donga.ac.kr/jwjo/Lectures/Papers/(Avishay%20Orpaz)Co-Design.pdf *A study of CodePack: optimizing embedded code space*. CodePack seems to be able to shrink PPC code to about 2/3 the original size, and is essentially performance neutral (1999) https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.78.1091&rep=rep1&type=pdf *Evaluation of a High Performance Code Compression Method*.
On the other hand, binaries for POWER tend to be about 20% larger than for ARMv8, so maybe there is less redundancy to squeeze out of ARMv8, and so less of a win?

Power saving for Instruction Fetch

There are different opportunities for saving power on the instruction cache side than on the data cache side. The primary difference is that there's a lot more "structure" in instruction fetching.

(2010) sequential access within one line

The easiest way to exploit this is when instruction fetching is sequential and within the same cache line. We see this with (2010) https://patents.google.com/patent/US8914580B2 *Reducing cache power consumption for sequential accesses*, which suppresses 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.

(2013) sequential access across cache lines

But that's of limited applicability once the maximum Fetch width is the size of a cache line, and once we start loading a trace across two cache lines within a single cycle.
So next we have (2013) https://patents.google.com/patent/US9311098B2 *Mechanism for reducing cache power consumption using cache way prediction*. This imagines not a way predictor but a *Sequential Way Predictor* that, when given a Fetch PC, gives the way information for say the next four sequential cache lines. This will allow looking up and precharging only the appropriate way(s) for the next trace (which may be spread over two lines for this lookup, and then perhaps even two more lines for the next lookup if we have a long run of sequential code). This allows us to avoid the tag and precharge costs once the sequential way predictor is properly populated, but what about the TLB costs? My guess is that the most recently used TLB translation is stored in a register in Fetch and that's used rather than the TLB whenever a match allows for that. (Of course this is one more thing that has to be wiped whenever a TLB control command is sent to the CPU...)

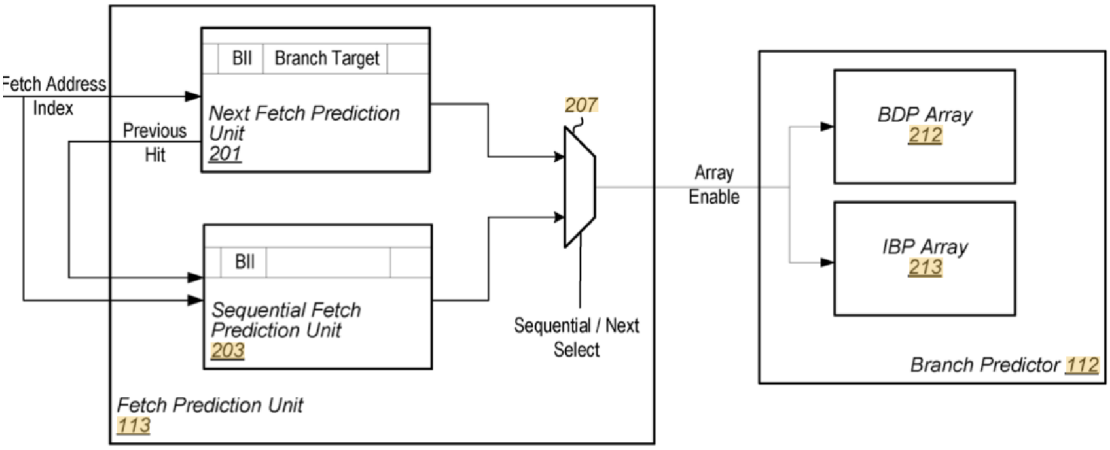
(2013) access via the Fetch Predictor

The previous (early 2013) patent is reasonable, but why are we using a different predictor, and one that only holds some of the info we would like to have?
An obvious improvement is to move as much of this stuff (predicted way? perhaps even the ITLB lookup?) into the Fetch Predictor entry...

(2013) https://patents.google.com/patent/US10901484B2 *Fetch predition* [sic] *circuit for reducing power consumption in a processor*, suggests aspects of this, while saving Fetch energy in other ways.
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 conditional 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 branches are present in the Fetch Group, but what branches will be present in the Fetch Group that it flows into (via the Sequential Fetch Prediction Unit). 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. Of course, once such branches are captured by different structures that no longer make use of the Fetch machinery [eg loop buffer or L0] then the design tradeoffs change.)



Along with the above branch prediction energy savings, the second part of the patent has to do with avoiding ITLB and tag lookups. Of course the baseline idea has to do with storing the required information in the per-Fetch Group lines of the Sequential and Next Fetch Prediction units; the point of the patent is: how do you handle the cases that can go wrong, for example when the cache contents are modified? You want to catch these cases (since the stored way information may now be out of date), but how do you do so easily?

The essential idea is that the Fetch Prediction Unit holds two bitmaps of every line of the I-cache, and an algorithm sets or clears one or both of these bitmaps in response to various events (more importantly when the I-cache is modified).
(The diagram below omits the Sequential Fetch Predictor, but that's hooked in like the Next Fetch Predictor. If the two bitmaps indicate that "tag and TLB reuse are OK", so essentially
- we have used this tag/TLB before and they have been recorded in the Fetch Predictor [one bitmap] and
- this line of the cache has not been changed [other bitmap]
then the enable lines from the Bypass Unit can be suppressed.)

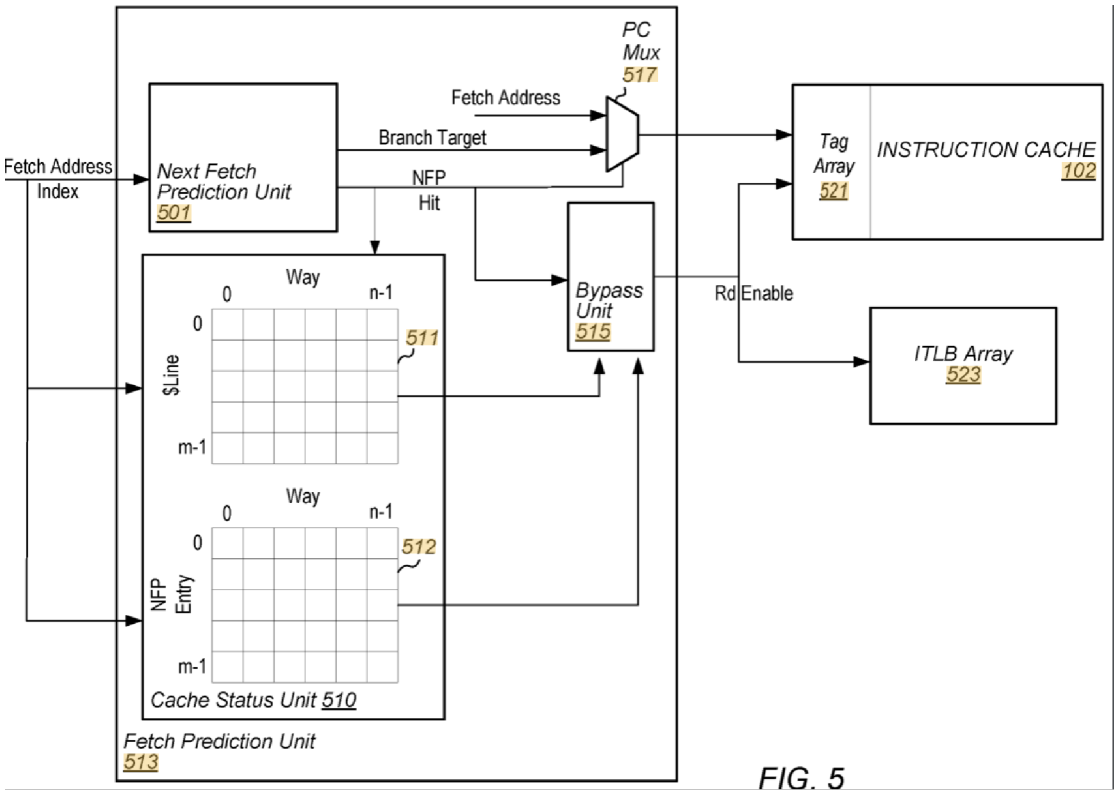


FIG. 5

(2016) optimize Fetch across subroutine calls

(2016) https://patents.google.com/patent/US10203959B1 *Subroutine power optimiztion* [sic] builds on this above idea. Some patents are not about a grand new idea, just about getting a little more value out of pre-existing hardware...

The hardware we already have
- has the traditional I-cache plus some auxiliary storage for each cache line holding pre-decode bits
- stores information about each Fetch Group (that has already been handled at least once) in the Next Fetch and Sequential Fetch Predictors.

Now consider what happens when we make a call (direct or indirect). We access the I-cache line and load various instructions up to the call instruction, and we also load in the associated pre-decode bits. But it's very little extra energy or effort to instead load the entire line's worth of pre-decode bits...

Think what that gives us. Specifically it tells us what sort of branches appear in the code that will be executed right after we return from the call. We can store this information in the Return Address Stack, and then use it when we access this stack to clock gate the Conditional and/or Indirect Branch Predictors for another cycle.
This is not necessary for a call we have recently encountered because such a call should already be stored in our predictors, but it helps a little if this is the first time recently that we have made this call

and so have no information about what to expect when the call returns. In a similar fashion if there is an unconditional branch in those instructions that will be executed after the return, we can use those to set the Fetch Width for when we access the I-cache after the return.

Summary of Fetch

At this point you've had a thorough overview of everything Fetch related, from Fetch itself to Branch Prediction, to the special characteristics of loops. You might want to read (2018) https://arxiv.org/pdf/1804.00261 *A Survey of Techniques for Dynamic Branch Prediction* as a nice overview/review of many of the ideas we have covered (and some we have not). You'll now appreciate many non-obvious points that appear in the review including
- why less sophisticated predictors may still be interesting even if we use something sophisticated like TAGE
- why loop exit conditions (and nested loops) are special
- how "local" branches (eg branches within a loop [or a function]) are somewhat segregated from branches outside that region, and perhaps that should be exploited?
- why the question of whether/how loops saturate branch history is important, and whether steps should be taken to avoid it

Superficially it seems like branch prediction is solved – just use TAGE! – but not at all, once you start to consider all these additional aspects, in particular how to optimize both for performance and energy, not just performance.

As one example, the paper (2008) http://www-mount.ece.umn.edu/~jjyi/papers/ipdps2008.pdf *Low power/area branch prediction using complementary branch predictors* looks like it might make a nice, small, companion predictor for the primary M1 TAGE system specifically to deal with loops and loop buffer mode. (Recall the goal is not only to be able to predict the branch exit correctly, but also to be able to shut down TAGE soon after we start the loop, so that we don't have to keep it powered up for predicting the exit condition until branch history saturates.)

You may also want to see (and will now appreciate!) some of the new alternative ideas for optimizing wide Fetch (that also acts like a Prefetch engine), like (2022) https://ease-lab.github.io/ease_website/pubs/SSFEP-ACM22.pdf *Shooting Down the Server Front-End Bottleneck*; or ways to reduce the storage costs of a large Fetch engine/BTB (2021) https://arxiv.org/pdf/2106.04205 *Micro BTB: A High Performance and Lightweight Last-Level Branch Target Buffer for Servers*.
One way to think about both of these is that they are both built on the realization that I-prefetch, by itself, is of limited value: it doesn't help much to have the cache lines associated with a new function in the I-cache if the actual Fetch of those lines is going to result in a constant stream of misprediction, flushes, and re-fetches because the Fetch machinery doesn't have any data to cope with these new lines.

One (partial) way of handling this is Apple's Scan On Fill Predictor; but another way to handle it is via having a large L2 BTB which can cover not just the L1I footprint but also a good chunk of the L2 I footprint.
Which then raises the usual various questions for such a structure:
- how should I index into it? (hashing and skewing)
- can/should I compress the contents? (many branch offsets are very short)
- can I prefetch the contents into the primary (L1 BTB) to reduce latency?
The two papers give different answers to these questions, but both provide interesting data, and some good ideas as to an optimal solution.
(Another solution is just to overwhelm the problem with sheer area, which is the IBM z/ solution [144K entries in their BTB!]; but that's clearly not feasible for most designs. However all these issues should make you now understand why IBM thinks this is a sensible way to spend transistors.)

You may also find interesting the blog post https://xania.org/201602/bpu-part-three and the earlier and later articles in the series, which give you some feel for the Intel side. Likewise this thread https://groups.google.com/g/mechanical-sympathy/c/UFscifOU8AQ/m/iz_1uHmDFAAJ gives some interesting background as to what Intel was doing a few years ago (though the various commenters are sometimes confused about issues like Fetch Prediction vs Branch Direction Prediction and which kicks in when).

Additional Branch-related hardware

We now understand a variety of Branch/Fetch related issues, but there is a whole set of branch related storage on the back-end of the CPU that we need to discuss. This is hardware to track the various speculative state of branching, both to recover from misspeculation, and to hold pending branch data that has not yet been validated (remember, we don't want to train predictors on bad data).

To understand these structures, it's best to start by understanding the problems that need to be solved, and to understand those, always remember our machine is both Speculative and Out of Order. So not just can we learn that particular branch was mispredicted, we can learn of this misprediction in random order relative to other branches (earlier or later) and other instructions (earlier or later).
So supposed we need to recover from a misprediction. What that means, ultimately, is that all the instructions that occurred before the problematic branch need to be allowed to occur; and they will (ultimately) generate what's defined as correct state at the point of misprediction. Meanwhile all later instructions need to be flushed, along with all tentative state changes they may have made.

We discussed in Part 1 much of this machinery:
- the ROB holds instructions in-order, so that when a misprediction occurs we know what counts as instructions "before" the misprediction and instructions "after" the misprediction
- the History File records *tentative* changes to the register file, in such a way that, on misprediction, we can walk through the History file undoing changes (ie mappings of logical to physical registers) until we arrive at the correct logical to physical mapping at the point of misprediction.
- (there are ways to replace or augment the History File with Checkpoints, which can allow for more

rapid recovery after a mispredict. It's probable that Apple uses these, and we have a patent from 2013 that refers to them, but no further details.

(2013) https://patents.google.com/patent/US9311084B2 *RDA checkpoint optimization* discusses how checkpoints handle the RDA (the mechanism used to track when multiple logical registers point to the same physical register as of 2013. Of course this mechanism has changed at least twice since the RDA, so the details are surely irrelevant, but presumably Apple still uses checkpoints.

Checkpoints are essentially a way to store the state of the machine at some point so that, if that branch is mispredicted, the History File only has to work backwards to the checkpoint and no further. But checkpoints take a lot of storage, so you can't be generating them too frequently. The optimal strategy is probably to take checkpoints at branches that are considered
+ especially likely to mispredict, and
+ especially likely to take a long time to resolve.
But we have no further details.)

(2016) Branch Information Table (tentative branch training data)

But there is other branch prediction related machinery. For example
- we need to wind back any branch history vector and/or path history vector to its value at the point of the branch. Easiest to store these in the same location.
- (another way to make the same point) we want to store taken/not taken 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, 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 allows us to save substantial table space and some degree of training effort for the (lets guess about half) of the branches for which it's true.
With compiler help, this can be especially nice and helpful. Many branches are of the form if(exceptional-Condition){ handle exceptional case}, and naive compilation would require skipping over these small exceptional case handlers scattered throughout each function, with the *normal* code flow being a *taken* branch to jump over the handler. But tools can detect (via heuristics and/or profiling) this pattern and move all the exception handling code to a separate cold code segment that lives in its own page. This saves TLB entries (the "active" code footprint is smaller), it saves cache space (no cache lines holding "inactive" code), and most relevant right now, it turns the normal code flow into straightline code with all these if(exceptionalCondition) branches only being taken when the exceptional condition occurs.

We start with the Branch Information Table, discussed in (2016) https://patents.google.com/patent/US10175982B1 *Storing taken branch information.*

Given the job to be solved, 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 retire and training 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.

A second aspect of these relatives sizes is that the BIT is split into two "slices", the TBIT is split into three "slices", and the entries in the BIT second slice are the same size as the entries in the TBIT first slice, so that there's an element of dynamic sharing available here.
Specifically
In the the BIT
+ the first slice holds data for all branches
+ the second slice holds data for conditional branches
In the TBIT
+ the first slice holds data for indirect branches (eg virtual function calls)
+ the second slice holds data for all (taken) branches
+ the third slice holds data for calls and returns.

So I think the way this plays out is that for indirect calls (which tend to be rare) the TBIT indirect call storage is "overlaid" with the BIT conditional branch storage so that
+ traditional apps that rarely use indirect branches aren't wasting the indirect branch space, they simply have more conditional branch space available

+ apps that use a lot of indirect branches in the form of virtual function calls can use and simultaneously read/write both the first TBIT slice (holding data relevant to training) and the third TBIT slice (holding data relevant to the return address stack).

“Slices” is a word Apple uses in a few places and seems to refer to a structure that holds two or more “tables” with different sized “rows”. It’s unclear to me exactly how this is implemented!
This makes (only slightly!) more sense if we see a picture:

Slice	Field	Comments	
0	TBITTIDX	tBIT index for branch	For all branches
0	BDPADDRLO	Branch Address [5:2]	
0	BDPTAKEN	Branch predicted taken	
0	WRAP	BIT index wrap bit	
1	BSPADDRHI	Branch address [26:6]	BDP branches only
1	BDPUPDU	Branch will update BPD U bit	

The BIT above↑,
and the TBIT below ↓

Slice	Field	Comments	
0	BTPHITIDX	BTP hit table index	For BTP branches only (shared with BIT slice 1)
0	BTPUBITS	U-bits for BTP tables 1-6	
0	BTPCTRBITS	Ctr bit for BTP hit table	
0	BTPBRNFGPC	Fetch group PC for BTP branch	
1	BDPGHIST	BDP GHIST vector	For all branches
1	BDPPHIST	BDP PHIST vector	
2	RASPUSH	RAS push branch	For calls and returns only
2	RASPOP	RAS pop branch	
2	RASAGE	Age of RAS branch	
2	RASOPPTR	RAS pop pointer	

BDP means “Branch Direction Predictor”,
BTP means “Branch Target Predictor” which I take to mean the ITTAGE predictor.
The GHIST vector is the “target path history” vector, created by, on *taken* branches
- shifting the prior GHIST vector left by one bit then xor’ing in the new branch target
The PHIST vector is the “PC path history” vector created by
- shifting the prior PHIST vector left by one bit then xor’ing in 4 bits of new branch PC.
I’ve given you the rough theory of why a path address is useful, but I’ve seen no explanation as to why one might want to track both of these, or where one might be more useful than the other.
Also note that (at least according to what we see in this patent) a *branch direction* history is not tracked.

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 PHIST 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 path 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.
We've seen how the PHIST vector can be updated as necessary. How about the GHIST vector? I've no idea! Maybe the GHIST value that's stored in the TBIT incorporates the new target, and is by definition always correct if it needs to overwrite the old value? This is possible because GHIST only updates on taken branches.
It's unclear when PHIST updates, but if it also updates only on *taken* branches, then we need the lower 4 bits to undo the 4 bits that have been xor'd into PHIST in the event of misprediction so this (predicted taken) branch was not taken?
These seem like details (eg use 4 bits for PHIST data, update PHIST only on taken branches) that are something of a compromise between "what simulations show works well" and "what can we fit in our table given the bit layout", and have probably changed between then and now.

+ 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), as expected, but also add a "BTP hit table", which is probably tracking how frequently the predictor is used for power saving purposes (shut it off if it's rarely being used).
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. (And once again, it's quite likely the details have changed by M1.)

So do we know the current size of the BIT and TBIT? Well, to be honest we don't even know if these structures still exist in this form, though something like them surely does.
You might consider a probe consisting of a long stream of conditional (not taken) branches, after a long dependency conditional branch, which might block when the BIT becomes full. That sounds good in theory but night now work! The patent suggests that, insofar as all this data exists purely for the point of *training the predictors*, not for correctness, if one of the tables ever becomes full, the machinery should just keep going overwriting older entries. Presumably the tables are sized so that this should rarely happen, and under conditions where the tables do overflow the situation is so artificial (eg benchmark!) that you're probably not learning anything useful anyway from super-accurately training the predictors?

If you want to take anything away from this complicated (and somewhat guess-filled) section, it's that Apple takes very seriously the point that branch predictors should be trained *only* on fully accurate (non-speculative) data, and does some non-trivial work to ensure that.

(2016) moving instructions from Fetch to I-Queue/Decode

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):

(2016) https://patents.google.com/patent/US10445091B1 *Ordering instructions in a processing core instruction buffer*.

The patent shows us both the naive solution, and a somewhat neater solution.

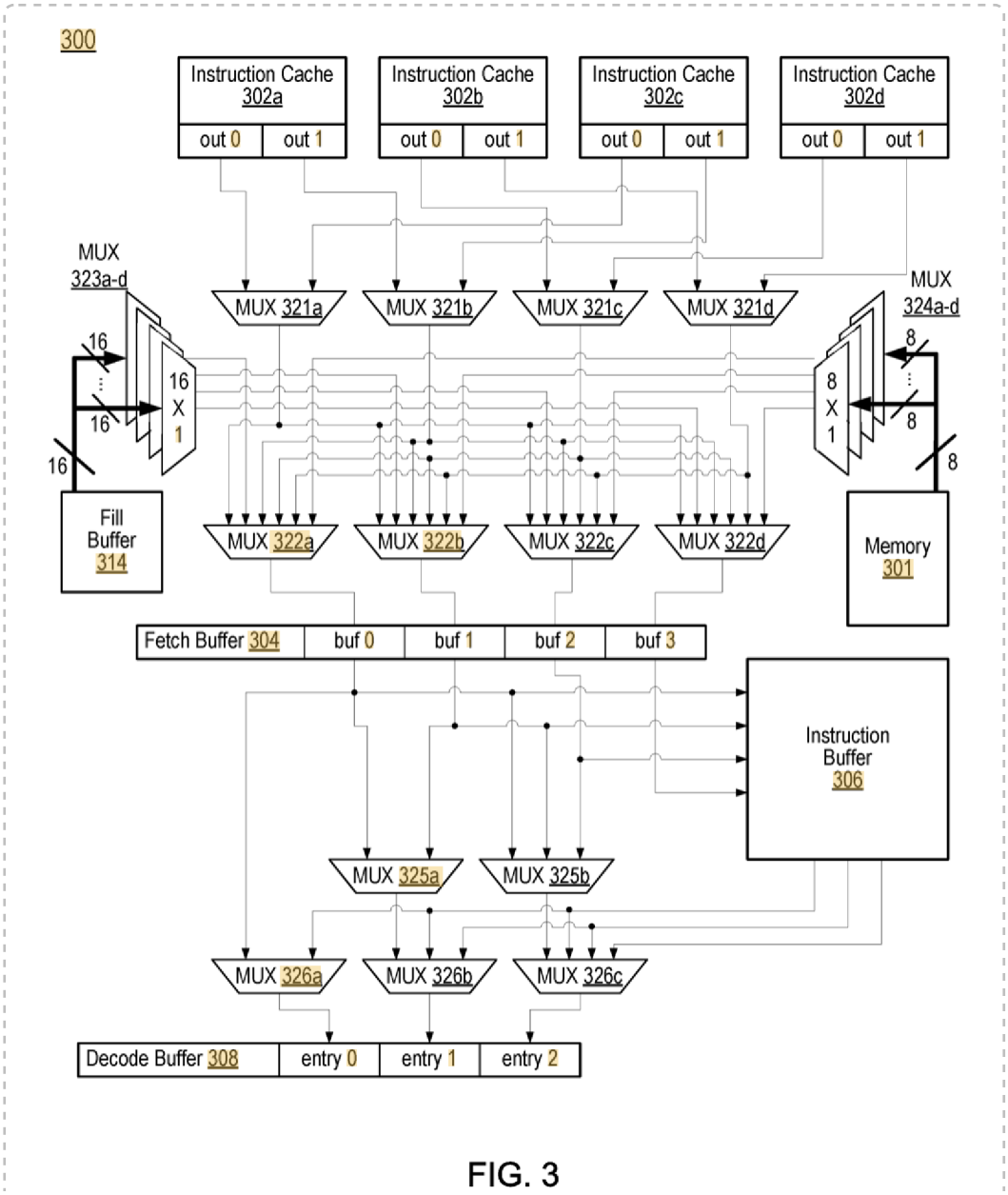


FIG. 3

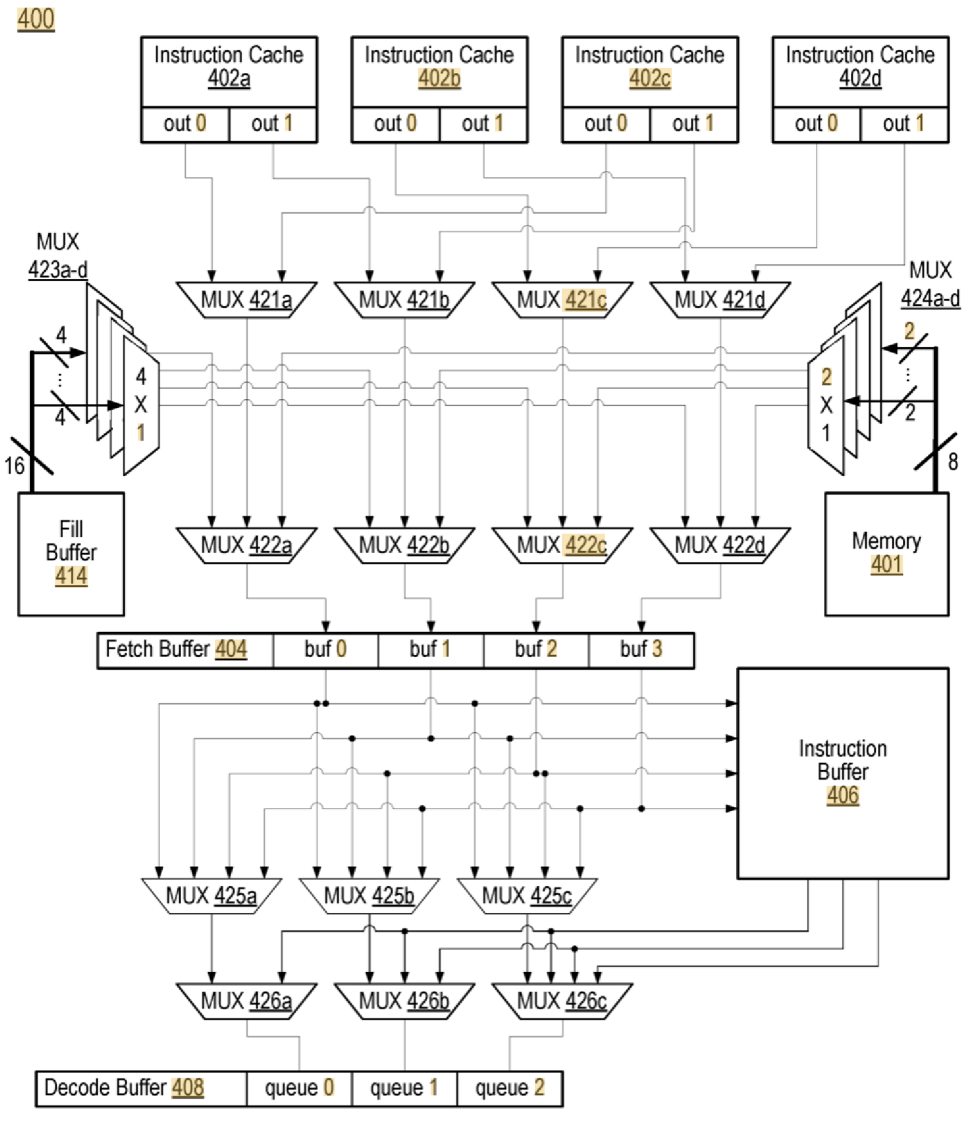


FIG. 4

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).

(2016) rapid recovery after cold start

Two points that should be clear from all the above discussion of instruction fetch and branching:

- instruction prefetch, by itself, is only half the battle! If instructions are pre-fetched from a region of memory that has not been seen before, or not been seen recently, then it's nice that we will not waste time waiting for the instructions, but we will still waste a substantial amount of time guessing about half the branches incorrectly before we train the various predictors. This is, as far as I can tell, a problem that has not yet been seriously tackled.
(The closest is that IBM z/ series implements truly massive L2 predictor storage, and tries to shift large amounts of data from it into the L1 predictors when it looks like something "significant" has changed; for example the program being executed moving to a different phase of operation, or a context switch.)

It's conceptually obvious that one wants ever better instruction prefetchers, but how to handle the branch misprediction problem?
A perfect solution (in other words capturing the full TAGE state of prediction) seems impossible, but one could imagine a solution that at least captures the single best context-free guess for each branch (ie the more common direction, a single bit) and somehow storing that in some sort of persistent storage associated with the source instructions in RAM (and even, if one were truly ambitious, pushing this out to the code storage in flash).
I'd love to see someone model the value of such a scheme, to at least see if it's worth doing before we think about the difficulties of an actual practical implementation.

- every time a core loses power (ie really powers down, rather than just sleeping) it loses all this state, including not just the branch predictor state but also the instruction code states.
This seems like just a fact of life, nothing you can do about it, but amazingly (though it's obvious once you see the trick) that's not quite true!

the conceptual insight

What determines how aggressive you make an instruction prefetcher? Ultimately you are balancing energy and accuracy. You don't want to waste energy on useless instruction loads, but you also don't want useless instruction loads to remove valuable instructions from the cache.
But suppose that there is nothing of value in the cache! Then the optimal tuning of your I-prefetcher would be more aggressive, since your only constraint now is not to waste energy. You could even keep a count of invalid lines in your cache, and gradually dial down prefetcher aggression based on this count of invalid lines which, in a way, is a generalization that works under all conditions, not just after cold start.
Even better (it's not obvious how to implement this, though one may be able to fake it well enough) would be applying this same degree of variable prefetch aggressiveness after every context switch, where one doesn't care about overwriting non-shared lines from the previous process.
There is a second, less obvious, place where one can utilize this idea, in the branch predictor.
Recall how TAGE works, and the essential idea of having many tables each storing histories of a particular length, from short to long histories.
How is that table populated; that is, when we encounter a new branch, do we fill in an entry in the shortest-history table? The longest-history table? All the tables?
There is an algorithm to do this, based again on the idea of not wanting to knock out existing useful entries, and the algorithm replaces entries in one to two tables. But again, if there is no pre-existing useful information in the tables, placing an entry in every length table is better than nothing, up until the point where the tables are mostly full of useful entries.

the patent details

These two ideas are essentially the content of (2016) https://patents.google.com/patent/US10007616B1 *Methods for core recovery after a cold start*. The patent suggests (again, details have surely changed) during the aggressive phase of a cold start, replacing each fetch of a single line from the L2 with three lines (perhaps the previous and next line? or perhaps the next two lines?), and replacing four (rather than two or one) TAGE entries.

Somewhat similar, or, if you prefer, a generalization, is (2017) https://patents.google.com/patent/US10346309B1 *Sequential prefetch boost*. The idea is (for both I-prefetch and D-prefetch) the L1 prefetcher monitors whether demand or prefetch misses take longer than expected to return, and if so (ie they are missing at L2, or at SLC) then prefetching is made more aggressive on the assumption that something has changed (perhaps a cold start, perhaps change in the code behavior) and need to pull more state in the L1 caches more rapidly.

That seems kinda basic and obvious if you just assume that each prefetch request for one line is turned into something like a prefetch request for that line and the next three lines.

However a more sophisticated and interesting way to interpret and implement the patent is to have a

Prefetcher Address Engine (again I or D) decoupled from the actual prefetching, and generating a stream of prefetch requests that go into a queue.
Now what varies over time is how many entries from that queue we allow to be currently active and submitted to the L2 and beyond. So (to just make up some numbers, the patent does not give any) perhaps the queue might hold 64 entries. Under normal conditions we allow 12 of the entries to be active (submitted to the L2), but if we have seen a pattern of misses from the L2, maybe we allow up to 32 entries to be active, and if we have seen a pattern of misses from the SLC, then maybe we allow all 64 entries to be active.

This design preserves the goal of "boosting" the amount of prefetch when appropriate, while preserving the goal of having the prefetches be "calculated" on some principle, rather than just blindly sequential. And just like everywhere else in the CPU, we want to connect two somewhat independent tasks by a queue so that if one of them is forced to stall (either Prefetch Address Generation runs out of ideas for new addresses, or actual Prefetch runs out MSHR's) the other can keep going for a few more cycles.
Obviously a scheme like this fits well into FDIP as we've already described it, as well as matching what we know of the Data Prefetch side, where slightly earlier patents like (2017) https://patents.google.com/patent/US10331567B1
Prefetch circuit with global quality factor to reduce aggressiveness in low power modes (which we have already discussed, as adding Large Stride and Spatial Memory Streaming Predictors to the base AMPM predictor) show a Prefetch Address Queue holding addresses to be sent out to L2.

I think it's pretty clear that Apple is not there yet (many pieces are missing). But one day...

Decode

The previous long section had to do with ensuring a constant (usually correct) stream of instructions flowing through Decode. Decode converts a short instruction into a longer bit pattern that specifies exactly what the machine has to do to perform an instruction.
Along with the sexier functionality like dropping NOPs and Fusion, is there anything else interesting in Decode?

Well, of course power is always an issue, and so we have (2010) https://patents.google.com/patent/US20120079249A1 *Training Decode Unit for Previously-Detected Instruction Type*.
The patent is obviously for a Swift type of CPU, but the pattern makes enough sense that it probably still holds. Each Decoder (so 8 as of M1) is a block that consists of a few sub-decoders (the patent suggests Integer, Load/Store, Vector Integer, and Vector FP).
Integer and Load/Store need to be active all the time, but a lot of code uses neither, or only one, of the two vector decoders. And so the instruction to be decoded is fed directly to the Integer and Load/Store decoders, but is usually blocked from going to the Vector Decoders and so, because these do not engage in any logic transitions, they only burn a little leakage power, no more.

But what about code that wants to use these instructions?
Two things take care of this.
The first is that after such an instruction is encountered both Integer and Load/Store will indicate they did not “accept” the instruction, so it will loop around to be passed through the Decoder again. This costs at least a cycle (possibly more depending on exactly how it’s implemented) and we don’t want to keep doing it! So the PC that resulted in this loop around is recorded in a small table and next time that PC is encountered the instruction will not be blocked from the Vector Decoders.
Secondly after each Vector instruction a timer is started/reset, such that for some number of subsequent cycles the appropriate Decoder is kept active so that, in the common case what happens is a first instruction (perhaps beginning a loop) wakes up the FP Decoder which then stays awake as long as the loop is running, then goes to sleep. This allows the table of “Vector PCs” to remain small while still catching most cases that matter.

In terms of precise details, once can see a number of ways this scheme is not optimal, but it’s a nice start and I expect, like every good idea from those early PASemi days, it’s been substantially refined.