<<-<<DISCLAIMER: I don’t really know what I’m doing so IDK if answers are correct.

Article link: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf> (note the article is only chapter 2 of that ginormous pdf)

Ofc a work in progress.

#roastme:

1. 1. Virtual. The cache is small and associative enough that it can be indexed only by page offset bits. This means that we can use a VIPT cache design, and check the (physical) tag in parallel with accessing the cache.

* 1. I. <diagram should show the victim cache sitting underneath L1>. Data is allocated into the victim cache when it gets evicted from the L1 cache. It will be accessed if a miss occurs in the L1 cache. The cache is usually small, and is looked through sequentially (?) to find a victim entry with matching tag.

II. The write buffer is allocated into when the L1 cache needs to write to L2 due to evicting an entry. This allows the L1 cache to continue to service requests while its evicted entries are being written to L2. It will be accessed when read misses occur (to make sure they aren’t in the write buffer), and when the L2 isn’t busy (to write and deallocate entries).

III. The write buffer holds entries until they are written. The victim cache holds data until they are evicted by another entry.

The write buffer would most likely be blocking if it’s full. The victim cache isn’t, since it can just evict an entry to make room.

* 1. Stores are delayed until commit time in order to enforce memory ordering, and to prevent loads and stores on misspeculated execution paths from being made. Loading and storing occurring at commit time means that only loads and stores which are certain to not be misspeculations get performed.

Stores will be released when the instruction triggering that store gets committed. They will also be deallocated if it turns out that the instruction that triggered the store was executed on a misspeculated execution path.

* 1. Perhaps yes? The LSU could detect invalidations for its lines, and request for those lines to be refreshed ahead of time, which would save time.
  2. 1. if an instruction being issued is complex and requires multiple independent operations (likely, given that the architecture is CISC)

2. ?

* 1. Because the constituent micro-ops of any given macro-op must retire at the same time, to give the illusion that the instruction decoded into the macro-op performed its operation atomically - i.e all of the changes on the architectural state happened in one go.
  2. - 20 entry limit on ALU scheduler

- 12 entry limit on AGU scheduler

- 18 entry limit on FP scheduler

* 1. I. 1. The macro-micro-op distinction could probably be entirely remove

2. No need for the microcode ROM

* 1. II. 1. Cheaper to produce hardware

2. Smaller chip size

* 1. Most likely each bit represents the takenness of some recent branch, for example the nth bit could represent whether the nth latest branch was taken or not.
  2. Not sure on this one… possibly if a method is being called on an interface, the global history could indicate that it’s most likely to be a certain implementation, so it could be predicted to jump to that implementation’s method.
  3. 1. If the return address on the stack is manually overwritten

2. If the call stack was particularly deep and the current return address ended up getting popped off of the (FIFO) RAS at some point to make space for later addresses.

* 1. The obvious one here is the case where an L1 BTB miss occurs. Not sure about the others
  2. I. If at least one of the threads satisfies the condition?

II. Possibly not. GPUs generally operate on the principle that having many threads running at the same time means that stalls caused by things like control hazards are less important, since other threads can step in and do useful work during that time.

However, it's possible that if a sufficiently reliable predictor was used it could be worth it.