Improvements welcome (there are probably many things I have missed):

1.

1. List of things I can think of:
   1. If the ROB is full
   2. If the store data buffer is full
   3. If the “gather-scatter” buffer is full
   4. If the rename buffer is full
   5. If the reservation stations needed for the instruction are full
   6. Nothing in instruction queue
2. If the allocation unit stalls, the processor will not be able to do anything at all. If execution stalls, the processor will still be able to fetch, allocate and rename instructions, to get them ready for execution at a later time (they can just sit in the reservation stations until the stall stops).

The reasons for an allocation stall seem to be harder to fix than those that cause execution stalls. If any of the reasons outlined above have happened, something pathological is going on, and it will take a while for whichever table/buffer is full to escape from being heavily loaded.

Execution stalls would happen relatively commonly (?) due to structural hazards etc, so they are not necessarily such a bad sign, since many of the causes are easily recoverable.

Stalls in instruction execution are turing tax, while stalls in the allocation unit is poor processor design.

Blocking at allocation prevents all instructions passing through the pipeline, whereas blocking at execution, only prevents the instructions which require that unit from continuing. Other types of instruction may still be executed (due to OoO).

Allocation unit is IN-ORDER if the allocation unit cant allocate the head of the queue, the following instructions cannot be allocated even if resources they require are available

1. Within Retire, since the retire stage (aka commit) is the final stage of pipeline, and enforces in order instruction retiring (as a ROB does).
2. FP and Integer register files, so that their values can be updated on retire. FP and Integer rename buffers, so that the renamed registers can be reallocated. The BPred unit should also be connected so that predictions can be updated to reflect the results of previous predictions. I’m not sure why there isn’t a Load-Store Unit on the diagram, but that should also be connected so that stores can be fired off to memory on retire. Allocate/Rename also to create ROB entries.
3. Either they are available at issue time, in which case they will be placed into the RSs at that time, or the RS (or ROB??) will listen on some kind of Common Data Bus for FUs to write back their results. Assuming this uses Tomasulo’s algorithm, the data will be written back with its tag, which will be used by reservation stations to identify whether the data is needed by them.
4. At Commit, from the ROB, which will (presumably) keep track of whether the branch was taken or not, and its direction if needed. It’s also possible that some updates could take place at the Execution stage - for example, it might be beneficial to update global history ASAP. BOOM does this (For future readers, this is related to the 2019 article), not sure whether this architecture does too.
5. Arguments for:

* Easier to implement
* If the threads execute the same program, having shared predictions could lead to better performance
* Faster, as there would be no need to swap out predictions on mispredict

Arguments against:

* Side channels
* Contention if running different programs with conflicting access patterns

2

1. I. Presumably the compiler will want to put 4 (for int)/5 (for FP) useful and independent instructions in between load and use. Another idea would be to try and do as many loads as possible all at once, so that the latencies overlap

II. If the compiler doesn’t do this, then performance will suffer since there will be stalls. In the worst case (if many load-use delays occur), the processor could stall for 4 cycles repeatedly, which would be catastrophic for performance.

1. I. If a written block is repeatedly written to in a short period of time, this block could be written to while it is still sitting in the write buffer. This would avoid having to write to the next-level cache many times. Also the case for reads. You can just get the data from the buffer instead of having to go to the next level of cache.

II. Possibly? It seems like a fairly cheap addition to the cache. It could improve performance in the case where the cache gets filled up with dirty copies of data (then blocks will need to be written back, and the cache would operate at the speed of the next-level cache if there wasn’t a write buffer).

Another reason why it could have one is to smooth out peaks in writing, meaning the L2 cache need not be able to handle as many writes.

III. Yes, coherency in a write-through cache is easier. In write-through, when a core reads some new data from L2, it will know for sure that it has the most up-to-date copy of data. If it wants the most up-to-date version, it can always read from L2. It can keep track of the validity of its cache lines simply by snooping writes to L2. This is not so easy in write-back, where its lines may get invalidated despite no writes occurring to the L2 cache.

1. This answer seems too simple, but here goes:  
   The cache can watch the stream of requests for new data and try and make predictions for which addresses are likely to be accessed next, and then fetch the data at those addresses. This will work better when considering instructions, where the access pattern is fairly simple. Data accesses may not be as predictable.

The article states that the prefetcher monitors memory address patterns and generates prefetches based on that. If I was an intel engineer the kind of memory access patterns I would look for fall into a few categories: i. Linear accessing of addresses in order, ii. Accessing every 10th word or similar(could be caused by matrix multiplication w/o transposing first , or a linked list with nodes allocated in the same place and largeish data payload), iii. Accesses made to the last word which was loaded(also could be caused by linked list or more generally any kind of pointer based data structure). All of these would allow me to determine what/where I want to prefetch.

1. I. Having many threads, so that during the memory access latencies other threads can run (latency hiding).

GPU’s is a weak point for me, but iirc b/c GPUs are so parallel, they have larger busses, lower clock speeds, and trade-off more bandwidth for higher latency. As long as the programmer places his data strategically all data needed for loads in every core will fit in relatively few memory cycles, since the bus is relatively larger.

II. Number of threads is limited by lots of things, the most important (?) being the amount of registers a core has. The threads will also contend with each other for other shared resources such as the branch predictor etc.

3

1. I. Contention? Having two threads means that the cache will need to respond to two separate threads, which will be accessing disparate memory locations (meaning it won’t be able to prefetch as effectively).

Also if the two threads are doing completely different things, you have only half of the cache capacity which could make a big difference in performance

Some other thread could have associativity conflicts with the thread you care about.

II. Side channels - threads can measure metrics on the shared cache to try and infer access patterns of other threads

III. If the two threads are accessing the same data. E.g. parallel processing.

IV. No idea, this is article-specific anyway . Probably something to do with less caches to sync in the mesh.

1. No idea, presumably a hash function leads to less collisions in the distributed tag directory?

I also have no idea. Maybe this deserves piazza.

I guess this is similar to a skewed-associated cache

1. List of ideas:

* Speed of next-level cache
  + RAM latency if ram is next level cache.
    - IIRC xeon phi products like this one where sold as graphics cards, so the speed of the pci-e connection to main system ram may also be relevant
* Write-allocate vs non-write-allocate, if writing
* Whether the cache gives priority to reads over writes, on misses
* Whether the cache supports critical-word-first
* Whether the cache supports early restart
* Whether the load instruction causing the miss was a speculated load that doesn’t block anything.
* Whether “hit-under-miss” is supported?
* If we have victim cache, or a write buffer we can try to find it in these units instead of request form main memory
* We might want to broadcast the request and if a nearby core has this, we dont have to go to main memory

Not sure what’s going on in question 4, someone else can have a go at that one.

4.

* 1. Predication without speculation? No need to flush pipeline because only the instructions with the taken bit will be able to modify the architectural state
  2. Remove branch prediction altogether? But increased length of execution might actually make energy usage worse
  3. Improve the branch predictor and use something state of the art like TAGE. There is a tradeoff but it might end up being better energy wise overall
  4. If branch predictor is not confident about it's result, don’t speculate based on that prediction and instead stall.
  5. Smaller problem size
  6. Only simulate a small section of the overall architecture
  7. FPGA
  8. Maybe it's possible to JIT/optimize out parts of the simulation, that consistently don’t really effect the final result, and/or if you have a loop, it's fair to assume the 334th iteration is roughly the same as the 335th iteration, so just reuse simulation data.
  9. Benchmark suites could be significantly smaller than real programs. This could mean the cache would do better and could maybe hold most or all of the required instructions/ data. This is unrealistic when it comes to real programs
  10. Benchmark suites can have extreme design choices. For example, one suite could be extremely memory-bound or compute-bound. Or they could have significantly more branches than a typical program which again would throw off results and not be representative
  11. Some compilers can do much better than others. For example one compiler could vectorise instructions which would affect performance
  12. The ordering of instructions can make a big difference especially on an in-order pipelined architecture. If one compiler can move loads up, the performance will be better compared to if the loads were interleaved with uses
  13. Most compilers are optimized for real world hardware, but if you are simulating an architecture, the cpu in question does not exist, and therefore there may not be an compiler\ optimized for it yet. You may find that a particular architecture that appears bad at first is better once compiler writers have made a backend optimized for that arch.
  14. One might end up designing an architecture for code generated by a specific compiler. For example if a compiler has bad software prefetching, you may end up building a much larger hardware prefetcher than needed.