1.c.

Instruction fusion can speed up computing memory addresses, by fusing add-immediate instructions with load instructions, overall allowing loads to run faster.

Instruction fusion can also fuse branch instructions with a following arithmetic instruction, effectively making the arithmetic instruction predicated. Thus branch processing (and the overhead of branch predictions) can be eliminated by effectively replacing branches with only one instruction underneath them by a predicated instruction.

We can also fuse load instructions with add displacement instructions to let us create loads with a larger “range” of accessible memory addresses.

1.d.

“instructions may return in any order, including interleaving of sectors for different cache lines”

So this allows critical word first behaviour. Perhaps also early restart.

Shorter cache lines don’t get the spatial locality benefits. And reduces cache capacity.

1.f.

Add one/many of:

Predication. Branch hinting. Delay slots.

2.a.

In single-threaded mode this will lead to a three-cycle stall in the pipeline, since there is no other useful work we could do in the three-cycle gap.

2.b.

Implement CSR idea from TAGE predictor in BOOM article?

Perhaps they append some lower-order bits of the instruction fetch address to make up the missing 3 bits

2.c.i.

The compiler could move loop-invariant code around to try and form as many of these branches which only have 1 instruction underneath them as possible. This would improve utilisation of instruction fusion.

2.c.ii.

When branches are predicted easily, we can convert the branch into a set of predicated instructions. We will not have to modify the predicated registers much, since the original branch would have most likely been either strongly taken or strongly not-taken. This also saves us space in any global history tables used, as less branch information has to be stored.

2.c.iii.

This isn’t as effective in reducing branch prediction penalties. Other ISAs may have support for having many more predicated instructions in a row than just one, meaning “larger” branches can be eliminated in favour of predicated instructions. This reduces the overhead of branch prediction, and the misprediction penalties, much further.

However full support for predicated registers would be much more costly, since many more registers would have to be added into the hardware and ISA support.

2.d.

The architects may not want the Global Count Cache (GCC) to contribute the entire branch target, as its quite an inflexible prediction model since it involves a large global history. Instead the GCC could predict the top 30 bits of the total address, as this is much easier to get right. The Local Count Cache (LCC) predicts a 62-bit target, which is much closer in length to the full 64-bit address. The top 30 bits of the LCC prediction could be compared against the GCC prediction - the GCC therefore acts as a double-check on the LCC prediction.

2.e.

The link stack implements “push” and “pop”, and therefore follows a First In First Out (FIFO) replacement policy.

If we reach the end of the stack, wraparound to the top. Since that will be the oldest address, which we may never return through. So LRU.

3.a.i.

A Store-Hit-Load (SHL) hazard is detected in the Instruction Fetch Unit (IFU).

3.a.ii.

Optimising for low power, the older store’s data should be forwarded straight to the younger load, thus saving the load from having to be replayed. If it is too late and the load has already executed, then the pipeline must be flushed and execution replayed from the load, this time loading the correct data. For maximum power efficiency, the processor should detect which instructions depend on the loaded data - only these instructions would then have to be replayed, instead of the entire pipeline.

3.a.iii.

We want SHL detection to be as fast as possible, to avoid detecting an SHL hazard after its too late and a load has already been committed. Therefore we should use virtual addresses for SHL detection, as we avoid having to perform a TLB lookup each time we make a detection as we would have to do if we used physical addresses.

I think a consequence of this choice might be that two different threads running through SMT on the same core might end up being detected as a SHL, despite them actually accessing different physical addresses.

3.b.i.

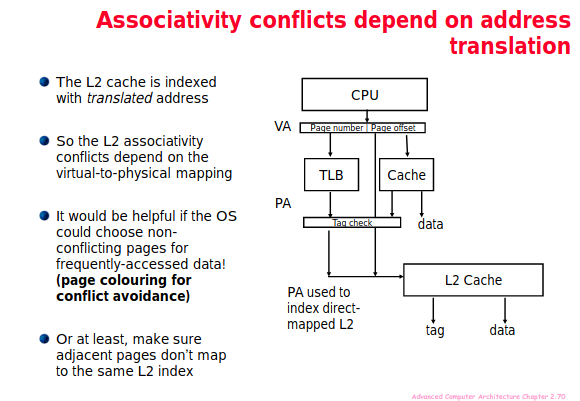
Advantages are that we don’t need to perform a TLB lookup before every cache access, which wastes time and energy.

3.b.ii.

If the data cache were indexed with physical addresses, we would need to perform a TLB lookup before every cache access. Depending on how data is laid out in the TLB, lookup times could vary (e.g. if the data we are looking for is at the beginning of the TLB in one run, and the end of the TLB in another run). This would then result in run-to-run performance variations.

The key variation would be programs would most likely run faster on the second run, since data that is uses would have been added to the TLB after the first run.

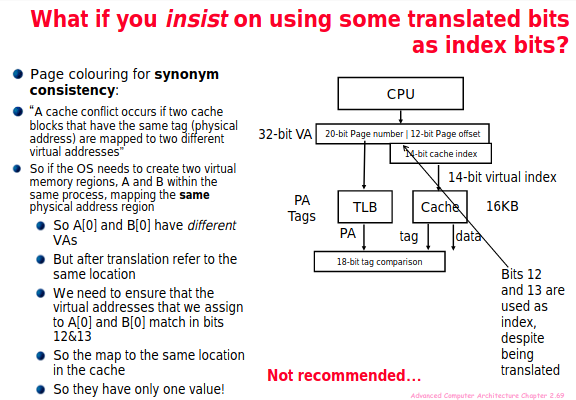
The Operating System will have a page allocation scheme which can be quite nondeterministic. So whilst we may allocate contiguous pages in virtual memory, their respective frames may be all over. Since the $ is physically indexed, these can potentially cause cache conflicts. Therefore muddling performance.



3.c. Assuming its VIPT.

If we have a page size of this minimum, we can fit two pages into the same cache.

So synonyms (many virtual addresses for the same physical) can all be in the cache in many places, polluting it. If the page size were bigger, these synonyms need only match in page offset to be indexed at the same location in $, and since it is physically indexed, their tags will also match, and so only one $ entry is possible.



3.d.i. The article states that only the most urgently needed lines are prefetched into the nearest cache levels. The article also states that the L1 cache is smaller, so can’t tolerate as many speculative requests as cache levels further down, such as L3, hence why prefetching is very conservative for L1.

Also the L1 cache is smaller, so it’s more easily polluted if data is prefetched that isn’t actually useful.

4.a.

RUU size and LSQ size are coupled. RUU size is effectively the number of reservation stations available for non-load/store instructions, and LSQ size is the same idea but for load/store instructions. Both individually improve performance and decrease energy usage, since increasing either of the parameters leads to more out-of-order execution opportunities being taken and more stalls being filled in the pipeline, but increasing both parameters together really magnifies the effect.

Fetch width and decode width are also coupled. Fetch width is the number of instructions fetched in one cycle, and decode width is the number of instructions decoded per cycle. When either of the parameters is increased, there is a performance boost to the system due to more instructions in the pipeline, however this also comes with an increase in energy usage. Increasing fetch width without increasing decode width will lead to decode width being the bottleneck in the system - increasing both together will increase throughput through the pipeline, leading to magnified performance boosts but also much worse energy efficiency.

4.b.

This allows us to compare real-world branch predictors (e.g. bimodal tables, global history tables and combination predictors) to the optimal output, to see how close we can get our predictions to true accuracy. In real life, even the best predictors come nowhere near “perfect” prediction behaviour - there’s often a tradeoff between performing extremely well in one specific branching situation, and performing only relatively well across a more general set of branching patterns.

Also can be used as a control parameter, so branch misprediction effect is removed when experimenting with other params.

4.c.i.

Some parameters would include how many cores there are, and the cache configuration for the caches in each core (e.g. n-way associativity, how many sets, overall size of cache).

Also cache coherency protocol, and specific cache policies like whether to use write-back/write-through.

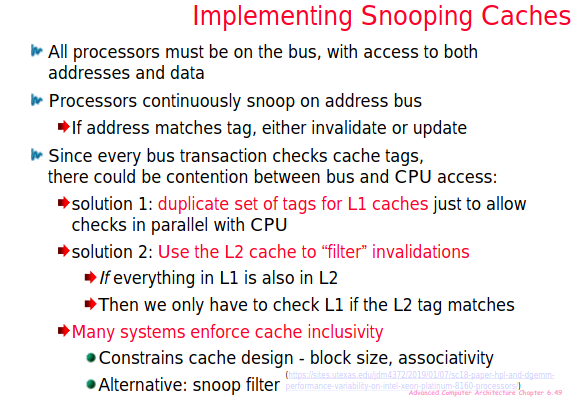
Shared memory specifics (size, location in memory hierarchy).

4.c.ii.

The advantage of having a highly-detailed model of each core is that very minute and precise tweaks can be made to find, for example, the most energy-efficient configuration of the processor, as well as the most performant (perhaps in terms of IPC). This would also mimic real-life processors more accurately which will often have thousands of parameters that could be modified.

However, the benefit of having a less-detailed but faster simulator would be that it takes much less time to run simulations and collect data, which can often take a very long time on highly-detailed simulators.

4.d.



4.e.

Assuming re-use distance means distance between instructions where this piece of data is used again. This would then improve temporal locality, as the compiler can spot where the same piece of data is called twice or more, and if possible move the instructions to be closer together in the code-generation stage.