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

Edit: Neither do we ¯\\_(ツ)\_/¯

We apologise to any future students who read this. - 2019 cohort

It’s all good, in 2022 we still don’t know what we are doing – Rob Stok

Article Link:

<https://www.anandtech.com/show/12785/arm-cortex-a76-cpu-unveiled-7nm-powerhouse>

1. 1. 1. Nano and micro BTBs allow for faster lookup since they are smaller. This means that predictions can be made quickly. Article states that previous gen branch predictor was able to predict almost all take branches, so prediction speed may now be a limiting factor.
      2. If lots of time is available for a prediction a large BTB would allow for a more accurate prediction, and/or is able to make a prediction when the nano/micro BTBs are uncertain.pas
      3. Rollback of any speculatively updated info in the BTB, adding the correct branch target to the BTB

Eg. an entries in the global history from speculatively issues branches should be discarded

* + 1. When the control flow is mispredicted, the branch predictor is informed of what happens, and will switch to sending the fetch queue instructions from the correct path. As this happens at a higher bandwidth than the fetch, the fetch stage wont stall because the new path will be in the fetch queue in time.
  1. 1. Stack was changed by another thread, for example a software scheduler may choose to insert a new subroutine into the stack, that it thinks should have priority.
     2. A C programmer made a mistake and overwrote his stack.  
        Imagine you had gotten into the habit of using unsigned indices in C, and decided to loop over an array backwards, not realizing that you were using an unsigned array index. Well then your array index would underflow, and become very large. This would be unfortunate because the stack is at very large memory addresses. This is a true pintos story.
     3. A very deep stack could lead to the processor having lost track of the top of the stack when the time comes to return to the top of the stack. One could even imagine a scenario in which the top of the stack is in swap, in which case the correct prediction would be a page fault?
     4. An exception unwinding part of the stack and confusing the RAS about which functions have/have not returned
     5. Code which was poorly compiled/handwritten and does not make use of ret and instead does janky stuff like adding and subtracting to/from the stack pointer and branching to a register containing the return address needed.
     6. OS context switch, we will start executing at any point in the other stream of execution (possibly) within a function call
     7. So many branches that we fill up the RAS beyond capacity. Have to start overwriting itself.
  2. 1. Against: Branch Target Prediction avoids having to decode branch instructions to determine their target pc and/or target pc offset. In the event of an offset you have to compute the target.
     2. Against: Branch Target prediction can predict stuff taken/not-taken can’t such as branching to an address stored in a register, whilst taken/not-taken takes up silicon space that could be used for more BTBs and doesn’t even predict every kind of branch.
     3. For: Much simpler/more silicon efficient to store branch history in terms of taken/not-taken, and therefore a taken/not-taken predictor can be both faster, and have more information at its disposal.
     4. For: A direction predictor can store history and therefore be more accurate than just using a BTB
     5. Against: You can assume that a BTB entry -> Branch and just incur any mis-prediction penalty. This way you save energy/silicon as direction predictors can take up quite a bit of hardware.

1. * 1. LOAD-USE delay on fmadd, combined with a single accumulator register which prevents the execution of the next fmadd, until the current one has finished.
     2. Unroll the loop, and use separate accumulator registers, for each unrolled iteration, which are then combined at the end.

Something like:

R1 = A[i] \* B[i]

R2 = A[i + 1] \* B[i+1]

. … etc

R = R + R1 + R2 …

This way we can rename R1...Rn and just have a dependency on R for each cycle of the unrolled loop. (We can assume that the previous load/add will be almost done or done)

* 1. 1. Non-associativity of floating-point which means you could get a different result if the adds are split up between different threads.
     2. Dynamic scheduling as opposed to static scheduling. Also could statically partition hardware (e.g. caches, registers)

Implement software consumer/producer model? That way each core takes as much work as they can.

* + 1. No, since floating-point is non-associative. Additionally, dynamic scheduling could also affect the result

1. 1. Move loads up such that, operand forwarding can occur between loads on s1 and adds on s1. Exactly how much loads need to moved up depends on the arch in question. (Software pipelining)
   2. Issue side - Each load result is assigned to a register in physical register file. Commit side - Loaded value will soon no longer needed as s1 has likely already been renamed to something else for the next load. Until the next load has committed the value remains in the physical register file. Once the next load has been committed the register in question is added to some sort of free list and will likely be reused by some other instruction.
   3. Capacity and contention issues?

Capacity: Cache insufficient to hold values from all arrays at once, leading to lots of weviction.

Contention: If array values happen to have the same low order index bits, they would evict each other.

Without reading the article it’s hard to be precise with sizes but the basic idea is if cache is size N and A and B are N apart A[i] and B[i] map to the same cache line and we get associativity conflicts (unless the cache is 2-way)

* 1. Arrays might map into the same cache bank, leading to stall/delays as less loads can be performed in parallel.
  2. 1. Loop N not divisible by the vector instruction width, so you need to cleanup the left over iterations, which adds complexity and generally sounds un-fun for the compiler writer. And of course adds code size which uses more cache, which makes program slower
     2. Vector instruction set might only have instructions for add 8 floats, so compiler would have to know to add a 3 zero registers. Overhead for add 8 floats vector, may exceed add 5 floats normally.
     3. Load use delay increased by vector instruction.
     4. Perhaps this is a memory bound problem and their is no vector load instruction that can handle 5 relative loads from 5 different address.
     5. Vectorizing could change the order of addition, causing different results than without vectorization.
     6. Compatibility with other cpus that do not have vector instructions, and b/c most people don’t compile with -march=native
     7. The R array could overlap with A, B, C, D or E

1. 1. By adjusting the cache (quick and easy to do in sim) we can isolate the different types of miss
      1. Making the cache sufficiently large we are just left with the compulsory misses
      2. There are no conflict misses in a fully associative cache
      3. For compulsory misses, you can make it both really big and fully associative, then look at the first X misses (Or have the simulator track evicted misses)
   2. Hey how r u doing m8?
      1. Branch predictor may learn the entire problem, and get 100% accuracy after a few iterations, whilst not viable on larger problem sizes (also could be less complex on smaller programs)
      2. Problem data may fit in cache whilst might be memory bound irl
      3. Shorter loops may lead to worse predictor accuracy since predictor has less time to learn
      4. Not all code paths may be tested in smaller problem sizes
      5. Very large data sets may need to be stored on disk, making paging performance relevant, whilst not the case in a simulator
      6. On a small problem, dependent instructions might be nearer together so we can’t issue as many instructions (out-of-order)

(e.g. less instructions to move around)

* + 1. Return Address prediction: A problem with a larger call stack will exhaust the space in the predictor and start making wrong predictions
  1. 1. Train the branch predictor by feeding in valid inputs for x. Then feed an x that is
     2. out of bounds which will likely cause a misprediction
     3. Train the branch predictor so it takes the branch speculatively.

Flush the cache and feed the function the invalid out of bound memory address which is the address of C. This will be executed speculatively and the cache will be changed

Iterate through the outer array A and time how long accesses take with a high precision clock. The quickest access is likely to be the leaked byte.

Repeat until you’ve got all of C