0.1 Branch Prediction

Simple predictors re-learn faster Sophisticated predictors re-learn slower Selective predictors may choose faster learning predictors until better predictor ***warms up.***

Branch Folding: Instead of just the branch target address in the [[Branch Target Buffer]], store the entire target *instruction*.

This way, you can skip the Instruction Fetch stage for the next instruction, so the effective CPI for the branch is zero.

Could stash target instruction for both taken and not-taken to reduce mispredicition delay.

BTB is:
- ***Indexed*** with low-order PC address bits
- ***Tagged*** with high-order PC address bits

If a slower, direction predictor differs, re-steer:
- Squash first prediction by re-writing PC
- Fetch from improved prediction

Updating the branch predictor:
- Can update BTB as **SOON** as branch outcome is known (before committing)
  - This means that later branches have correct BTB predictions.
- Update BTB after commit - just in case of a branch misprediction.

0.2 Cache

Smaller caches -> Reduced hit time

Way prediction - extra bits are kept in the cache to predict the way of the *next* cache access. Means that you can access the correct set for the desired way first on a correct prediction, getting the performance benefit of direct-mapped caches (better hit time) and good hit rate with associativity.

Multiple banks in L2 allows for [[Non-Blocking Caches]] to use hit-under miss on cache miss (can service cache hit at the same time)

Early restart:
- Processor can continue as soon as requested word arrives
- restart CPU as soon as the requested word of the block arrives.
- Request the missing word first (critical word first) - Useful in large blocks
- Divide cache line into sectors - each with its own validity bit.

Write buffers:
- [[Write Through]] and [[Write Back]] caches rely on write buffers, to contain all stores that must be sent to the next level of the memory hierarchy.
- -Coalescing write buffers - Merge adjacent writes into single entry
- Write merging - Merges consecutive word writes to memory addresses to take up less space in the write buffer
  - Reduces stalls, less space taken up in the write buffer
  - Takes advantage of spatial locality (writing)

Hardware Prefetcing: Extra block fetched placed in a ***stream buffer***. After a cache miss, the stream buffer prefetches the **next** successive cache line.

On future misses, check the head of the stream buffer:
- If a hit, allocate into cache and prefetch the next line for the stream buffer
- If a miss, clear the stream buffer and start over.

Relies on having **extra memory bandwidth**.
Have multiple stream buffers - because programs often make interleaved, sequential streams of accesses.

Skewed-Associative Cache: Reduce conflict misses by using *different* indices in each cache way. Hash the cache index and some tag bits (i.e. XOR) As the indexes are pseudo random, conflict misses are reduced as they don't map to the same index anymore -> useful for loops.

Costs:
- Must have an address decoder per way
- Latency of hash function
- Harder to implement LRU

0.3 Sidechannels

Evict and Time:
- Let victim run
- Attacker evicts line of interest
- Let victim run again, measure difference in time to find if attacker affected it

Flush and Reload:
- Attacker evicts line of interest
- Victim executes
- Attacker reloads evicted line
- Fast reload
- victim touched the line, cache tag was used
- Slow reload
- cache tag was not used by victim

Prime and Probe:
- Attacker primes cache
- Victim executes, attacker times access to its own cache lines

0.4 Loads & Stores

Need to **STALL** loads until all possibly aliasing store addresses are known.

As loads and stores use **COMPUTED ADDRESSES**, they may not be known at issue time, so any store / load instruction could be a data dependence (RAW). Could wait as above, or:

- Speculate whether it is or isn't
- then check for misprediction
- Add a fowarding predictor to improve this speculation

Allow a load to proceed **BEFORE** we know for sure if / which any prior uncommited store instruction writes to its address.

Can use the idea of **store-wait** (Alpha 21264):
- Guess that ,there is no memory dependence
- Squash if there is, and mark the load instruction as *store-wait* in the load unit
- Next time that load is needed, it will wait for all previous stores to complete

Can also use the idea of a store set to further improve this, so that you're not waiting on EVERY store.

Store unit can mark entries as valid / speculative, so speculative instructions aren't stored in memory.

0.5 Return Address Stack

After decode, the instruction is checked:
- If JSR, add PC to RAS
- If ret, pop RAS, prediction was correct

If the call stack is deeper than the RAS:
- RAS will be empty at some point
- stack overflow
  - RAS prediction may be wrong: - Switching threads would change the stack pointer

Updating the RAS:
- On commit - might not have a prediction in time for addresses deeper in the stack.

If a conditional branch is mispredicted, and inside the taken version of that branch is a function call, then a return address is going to be pushed onto the stack.

To mitigate this and optimise for loops - we can have a shift register that records *prediction state*.

On every prediction, every time we hit a JSR instruction, we increment this register and let it push a return address onto the stack.

If we suffer a misprediction - we pop the RAS n times, where n is the value in this shift register, and we're back to the original state.

BTB entries would be affected though -> we'd have entries for return addresses when they shouldn't be there. That's fine though - as we shouldn't get to those return addresses unless we were supposed to be there, assuming there are no erroneous jumps.

0.6 Vector Instructions:

- Less fetching from instruction cache, but exploits parallelism
- Lanes execute in parallel
- Have vector predicate registers (512 bit wide - 1 bit per lane)
- Zero masking - when predicate is false, register for lane set to 0
- Masking - when predicate is false, register for lane retains old value
- If trip count is not divisible by vector instruction, need additional non-vector instruction to mop up
- If arrays are potentially aliased or are data dependent, cannot use vector instructions
- Use 'gather' instruction for indirect array indexing
- If the alignment of the operand pointers is not known, need instructions to align onto a 32 byte boundary

0.7 Cache Coherency

- If reading:
  - If another cache has dirty or shared-dirty, get from cache using bus-read
  - They set to shared-dirty, you set to valid
  - Else valid from memory
- If writing:
  - No action if dirty
  - if valid or shared-dirty, send invalidation on bus, set ours to dirty
  - On miss - line comes from owner, everyone else set to invalid
- Cache controllers send out signals
- duplicate tags in L1 to allow for parallel checks - or check in L2 with multilevel inclusion

0.8 GPUs

- SIMD but on multiple threads
- Each SM uses FGMT on each warp (thread on CPU)
- Each warp using 32-bit wide SIMD vector instruction for lanes (in lockstep)
- If threads in a warp diverge, bad for spatial locality

1 Sonic Boom

1.1 Instruction Fetch Stage

Additional pipeline stage is added after the frontend-decoders shift the decoded instructions into a dense fetch packet:
- Needed to pass into the backend
- Provides an interface - decoder always gets 4 instructions, even if 4-8 instructions are issued
- Overflow of instructions (fetched instructions - 4) are stored in the fetch buffer

Next Line Predictor (micro-BTB):
- Small, fully-associative cache that stores branch address and target address
- Can be improved with [[Branch Folding]]
- Improves fetch throughput on small-bodied loops (JMP to top predicted)

Repair mechanisms - restores predictor state after misspeculation:

- Loop-predictor and RAS are snapshotted and repaired on mispredict
- Maybe use a RAS pop shift register?

Superscalar branch resolution, with multiple branch resolution units:

- Handles when multiple branches are in the same packet
- Additional pipeline stage after WB to read the fetch-PC queue - a queue for PCs to fetch soon - and determine the target address for the oldest mispredicted branch from a vector of branch mispredictions
- Allows for aggressive scheduling of branches - don't need to wait for a branch resolution unit to finish

TAGE:

- High accuracy for dense areas with many conditional branches
- Provides bit vector of taken / not taken, each bit for each instruction in the fetch packet
- Updated during commit stage

1.2 Instruction Fetch Stage

SFB recorder:

- Records difficult to predict branches into internal predicated microOps
- Detects short-forwards and translates into set-flag and conditionally execute instructions
- Set-flag writes outcome of branch to predicate register
- Conditionally execute either
-
  - True - Executes instruction and places into target register
  - False - Copies original value of (stale physical) destination register into physical destination register

1.3 Instruction Fetch Stage

Dual Ported L1 Data cache:

- Two banks - for odd and even addresses (implemented as 1R1W SRAMs)
- Good with load-heavy bursts
- Can use [[Loads & Stores]] - memory dependence prediction
- Is a [[Non-Blocking Caches]]

Line-fill buffers:

- Cache requests to populate the line-fill buffer
- Allows for cache eviction and requests to happen in parallel
- Flushed into L1 Dcache when evictions are complete
- Misspeculated cache refills stay in line-fill buffers, which means they must be searched in parallel
- Flushed on context switch?
- Uses a next-line prefetcher - [[Hardware Prefetching]] - after a cache miss
- Updates L1 Dcache during speculation window - if there are no cache evictions in queue

1.4 Side Channel Vulnerabilities

SonicBOOM vulnerabilities - speculative execution changes the microarchitectural state temporarily:

- Spectre-v1: bypassing bounds checks
- Spectre-v2: branch target injection
- Spectre-v5: return stack buffer attack

Issues with Load / Store Unit:

- 3 cycle delay to allocate misses in the MSHRs and request data from L2
- Loads are optimistically fired to use OoO execution

Attack methodology:

- Attacker establish desired conditions
- Trigger victim execution with invalid instruction
- Victim loads secret into covert channel (data cache)
- Processor resets pipeline by squashing and rolling back
- Attacker probes data cache for information (flush + reload)

Bypassing Bounds Checks:

- Mis-train conditional branch direction prediction
- Trigger using a conditional branch

Branch Target Injection:

- Mis-train BTB for branch target prediction
- Trigger using a call to a mis-speculated target (malicious function)

- Mis-train BTB for branch target prediction
- Trigger using a call to a mis-speculated target (malicious function)

Return Stack Buffer Attack:

- Craft malicious gadget after a call site
- Trigger using a return, as exploiting RAS miss-match with software stack

RIDL (Line Buffers - MDS Attacks)

Mitigations:
Speculative Taint Tracking:

- **Allow OoO execution**
- **Destination of speculative load is marked as tainted in a hardware taint file**
- **Subsequent loads that use tainted addresses activate data memory fencing for all loads in the LSQ (fence_dmem).**
  - **This is detected at AGU (memaddrcalc)**
  - **All loads in LSQ - as we are in danger zone**
- **Also notifies frontend to send out dispatch hazards (dis), forcing the pipeline to stall and the processor to run in order temporarily.**
- **When fencing used, data cache refill is temporarily disabled and MSHRs are cleared until load is no longer speculative (and data is untainted)**
- **Untaint when instruction is commitable**

- **On misprediction, data in tainted registers is invalid**

Retpolines:

- **Use a direct branch to a function that evaluates the indirect address and places it in a register, using a call instruction**
- **Ret back once evaluated**
- **Removes need for target address speculation**

Cache formula: log2(cache line size) = offset bits log2(cache size / no of ways * address bits) = index bits tag bits = address bits - offset bits - index bits