Branch Prediction

Simple predictors learn faster, sophisticated slower

Selective predictors choose faster until better one *warms up*

Branch Folding - Instead of just the branch target address in the BTB, 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 misprediction delay.

BTB - *indexed* with low-order, *tagged* with high-order PC bits

If a slower, direction predictor differs, **resteer** by squashing first prediction by rewriting PC and fetch from improved prediction

Updating the branch predictor: - Can update BTB as soon as branch outcome is known (*before committing*), meaning *later branches* have *correct BTB predictions*.

Update BTB after commit - just in case of a branch misprediction.

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 - CPU continues as soon as requested word arrives. Request the **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 sent to the next level of memory. **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, as it *reduces stalls* and exploits *spatial locality*.

Hardware Prefetcing - Extra block fetched placed in a **stream buffer**. After a cache miss, the stream buffer prefetches the next successive cache line. Relies on having *extra memory bandwidth*.

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

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 and harder to implement LRU

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.

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** and check for misprediction. Add a **forwarding predictor to improve** this speculation.

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

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.

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: will be empty at some point - **stack overflow**.

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

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.

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.

Vector Instructions

Less fetching from instruction cache, less power, and *exploits parallelism*

Lanes execute in parallel.

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.

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**.

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*.

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.

Execute 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 executes instruction and **places into target register** (*true*) or **copies original value** of (stale physical) destination register into physical destination register (*false*).

Load-Store Stage

Dual Ported L1 Data $:

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 cache**.

Line-fill buffers:

Cache requests to populate the line-fill buffer.

Allows for **cache eviction and requests to happen in parallel**.

Flushed into L1 D$ 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** after a cache miss.

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)

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 - Rouge In-Flight Data Load) -

Spectre 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.

Formulae

Cache formula:

log2(cache line size) = offset bits

log2(cache size / no of ways * cache line size) = index bits

tag bits = address bits - offset bits - index bits