

The Lecture Contains:

- Cache Hierarchy
- States of a Cache Line
- Inclusion Policy
- The First Instruction
- TLB Access
- Memory op Latency
- MLP
- Out-of-order Loads
- Load/store Ordering
- MLP and Memory Wall

◀ Previous Next ▶

Cache Hierarchy

- Ideally want to hold everything in a fast cache
 - Never want to go to the memory
- But, with increasing size the access time increases
- A large cache will slow down every access
- So, put increasingly bigger and slower caches between the processor and the memory
- Keep the most recently used data in the nearest cache: register file (RF)
- Next level of cache: level 1 or L1 (same speed or slightly slower than RF, but much bigger)
- Then L2: way bigger than L1 and much slower
- Example: Intel Pentium 4 (Netburst)
 - 128 registers accessible in 2 cycles
 - L1 data cache: 8 KB, 4-way set associative, 64 bytes line size, accessible in 2 cycles for integer loads
 - L2 cache: 256 KB, 8-way set associative, 128 bytes line size, accessible in 7 cycles
- Example: Intel Itanium 2 (code name Madison)
 - 128 registers accessible in 1 cycle
 - L1 instruction and data caches: each 16 KB, 4-way set associative, 64 bytes line size, accessible in 1 cycle
 - Unified L2 cache: 256 KB, 8-way set associative, 128 bytes line size, accessible in 5 cycles
 - Unified L3 cache: 6 MB, 24-way set associative, 128 bytes line size, accessible in 14 cycles

◀ Previous Next ▶

States of a Cache Line

- The life of a cache line starts off in invalid state (I)
- An access to that line takes a cache miss and fetches the line from main memory
- If it was a read miss the line is filled in shared state (S) [we will discuss it later; for now just assume that this is equivalent to a valid state]
- In case of a store miss the line is filled in modified state (M); instruction cache lines do not normally enter the M state (no store to lcache)
- The eviction of a line in M state must write the line back to the memory (this is called a writeback cache); otherwise the effect of the store would be lost

Inclusion Policy

- A cache hierarchy implements inclusion if the contents of level n cache (exclude the register file) is a subset of the contents of level n+1 cache
 - Eviction of a line from L2 must ask L1 caches (both instruction and data) to invalidate that line if present
 - A store miss fills the L2 cache line in M state, but the store really happens in L1 data cache; so L2 cache does not have the most up-to-date copy of the line
 - Eviction of an L1 line in M state writes back the line to L2
 - Eviction of an L2 line in M state first asks the L1 data cache to send the most up-to-date copy (if any), then it writes the line back to the next higher level (L3 or main memory)
 - Inclusion simplifies the on-chip coherence protocol



The First Instruction

- Accessing the first instruction
 - Take the starting PC
 - Access iTLB with the VPN extracted from PC: iTLB miss
 - Invoke iTLB miss handler
 - Calculate PTE address
 - If PTEs are cached in L1 data and L2 caches, look them up with PTE address: you will miss there also
 - Access page table in main memory: PTE is invalid: page fault
 - Invoke page fault handler
 - Allocate page frame, read page from disk, update PTE, load PTE in iTLB , restart fetch
- Now you have the physical address
 - Access Icache : miss
 - Send refill request to higher levels: you miss everywhere
 - Send request to memory controller (north bridge)
 - Access main memory
 - Read cache line
 - Refill all levels of cache as the cache line returns to the processor
 - Extract the appropriate instruction from the cache line with the block offset
- This is the longest possible latency in an instruction/data access

◀ Previous Next ▶

TLB Access

- For every cache access (instruction or data) you need to access the TLB first
- Puts the TLB in the critical path
- Want to start indexing into cache and read the tags while TLB lookup takes place
 - **Virtually indexed physically tagged cache**
 - Extract index from the VA, start reading tag while looking up TLB
 - Once the PA is available do tag comparison
 - Overlaps TLB reading and tag reading

Memory op Latency

- L1 hit: ~1 ns
- L2 hit: ~5 ns
- L3 hit: ~10-15 ns
- Main memory: ~70 ns DRAM access time + bus transfer etc. = ~110-120 ns
- If a load misses in all caches it will eventually come to the head of the ROB and block instruction retirement (in-order retirement is a must)
- Gradually, the pipeline backs up, processor runs out of resources such as ROB entries and physical registers
- Ultimately, the fetcher stalls: **severely limits ILP**

◀ Previous Next ▶

Module 2: Virtual Memory and Caches

Lecture 4: Cache Hierarchy and Memory-level Parallelism

MLP

- Need memory-level parallelism (MLP)
 - Simply speaking, need to mutually overlap several memory operations
- Step 1: Non-blocking cache
 - Allow multiple outstanding cache misses
 - Mutually overlap multiple cache misses
 - Supported by all microprocessors today (Alpha 21364 supported 16 outstanding cache misses)
- Step 2: Out-of-order load issue
 - Issue loads out of program order (address is not known at the time of issue)
 - How do you know the load didn't issue before a store to the same address? Issuing stores must check for this memory-order violation

Out-of-order Loads

```
sw 0(r7), r6
```

```
... /* other instructions */
```

```
lw r2, 80(r20)
```

- Assume that the load issues before the store because r20 gets ready before r6 or r7
- The load accesses the store buffer (used for holding already executed store values before they are committed to the cache at retirement)
- If it misses in the store buffer it looks up the caches and, say, gets the value somewhere
- After several cycles the store issues and it turns out that 0(r7)==80(r20) or they overlap; now what?

◀ Previous Next ▶

Load/store Ordering

- Out-of-order load issue relies on **speculative memory disambiguation**
 - Assumes that there will be no conflicting store
 - If the speculation is correct, you have issued the load much earlier and you have allowed the dependents to also execute much earlier
 - If there is a conflicting store, you have to squash the load and all the dependents that have consumed the load value and re-execute them systematically
 - Turns out that the speculation is correct most of the time
 - To further minimize the load squash, microprocessors use simple memory dependence predictors (predicts if a load is going to conflict with a pending store based on that load's or load/store pairs' past behavior)

MLP and Memory Wall

- Today microprocessors try to hide cache misses by initiating early prefetches :
 - Hardware prefetchers try to predict next several load addresses and initiate cache line prefetch if they are not already in the cache
 - All processors today also support prefetch instructions; so you can specify in your program when to prefetch what: this gives much better control compared to a hardware prefetcher
- Researchers are working on load value prediction
- Even after doing all these, memory latency remains the biggest bottleneck
- Today microprocessors are trying to overcome one single wall: the **memory wall**

