CS 136: Advanced Architecture

Review of Caches

Why Caches?

- Basic goal:
  - Size of cheapest memory
  - ... At speed of most expensive
- *Locality* makes it work
  - *Temporal* locality: If you reference *x*, you'll probably use it again.
  - *Spatial* locality: If you reference *x*, you'll probably reference *x+1*
- To get the required low cost, must understand what goes into performance

Cache Performance Review

- Memory accesses cause CPU stalls, which affect total run time:

$$\begin{aligned}\text{CPU execution time} = \\ (\text{CPU clock cycles} + \text{Memory stall cycles}) \\ \times \text{Clock cycle time}\end{aligned}$$

- Assumes “CPU clock cycles” includes cache hits
- In this form, difficult to measure

Measuring Stall Cycles

- Stall cycles controlled by misses and *miss penalty*:

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty} \\ &= IC \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \\ &= IC \times \frac{\text{Memory Accesses}}{\text{Instruction}} \times \text{Miss rate} \\ &\quad \times \text{Miss penalty}\end{aligned}$$

- All items above easily measured
- Some people prefer *misses per instruction* rather than misses per (memory) access

Four Questions in Cache Design

Cache design is controlled by four questions:

Q1. Where to place a block? (*block placement*)

Q2. How to find a block? (*block identification*)

Q3. Which block to replace on miss? (*block replacement*)

Q4. What happens on write? (*write strategy*)

Block Placement Strategies

Where do we put a block?

- Most general method: block can go anywhere
  $\Rightarrow$ *Fully associative*
- Most restrictive method: block can go only one place
  $\Rightarrow$ *Direct-mapped*, usually at address modulo cache size (in blocks)
- Mixture: *set associative*, where block can go in a limited number of places inside a (sub)*set* of cache blocks
  - Set chosen as address modulo number of sets
  - Size of a set is "way" of cache; e.g. 4-way set associative has 4 blocks per set
- Direct-mapped is just 1-way set associative; fully associative is *m*-way if cache has *m* blocks

Block Identification

How do we find a block?

- Address divided (bitwise, from right) into *block offset*, *set index*, and *tag*
- Block offset is $\log_2 b$ where b is number of bytes in block
- Set index is $\log_2 s$, where s is number of sets in cache, given by:

$$s = \frac{\text{cache size}}{b \times \text{way}}$$

  (Set index is 0 bits in fully associative cache)
- Tag is remaining address bits
- To find block, index into set, compare tags and check valid bit

Block Replacement: LRU

How do we pick a block to replace?

- If direct mapped, only one place a block can go, so kick out previous occupant
- Otherwise, ideal is to evict what will go unused longest in future

Block Replacement: LRU

How do we pick a block to replace?

- If direct mapped, only one place a block can go, so kick out previous occupant
- Otherwise, ideal is to evict what will go unused longest in future
  - Crystal balls don't fit on modern chips...

Block Replacement: LRU

How do we pick a block to replace?

- If direct mapped, only one place a block can go, so kick out previous occupant
- Otherwise, ideal is to evict what will go unused longest in future
  - Crystal balls don't fit on modern chips...
- Best approximation: LRU (least recently used)
  - Temporal locality makes it good predictor of future
  - Easy to implement in 2-way (why?)
  - Hard to do in >2-way (again, why?)

Block Replacement: Approximations

LRU is hard (in $>$2-way), so need simpler schemes that perform well

- FIFO: replace block that was loaded longest ago
  - Implementable with shift register
  - Behaves surprisingly well with small caches (16K)
    - Implication: temporal locality is small?
- Random: what the heck, just flip a coin
  - Implementable with PRNG or just low bits of CPU clock counter
  - Better than FIFO with large caches

Speeding Up Reads

- Reads dominate writes: writes are 28% of data traffic, and only 7% of overall

$\Rightarrow$ Common case is reads, so optimize that

- But Amdahl's Law still applies!
- When reading, pull out data with tag, discard if tag doesn't match

Write Strategies

- Two policies: *write-through* and *write-back*
- Write-through is simple to implement
  - Increases memory traffic
  - Causes stall on every write (unless buffered)
  - Simplifies coherency (important for I/O as well as SMP)
- Write-back reduces traffic, especially on multiple writes
  - No stalls on normal writes
  - Requires extra “dirty” bit in cache
  - ... But long stall on read if dirty block replaced
  - Memory often out of date $\Rightarrow$ coherency extremely complex

Subtleties of Write Strategies

Suppose you write to something that's not currently in cache?

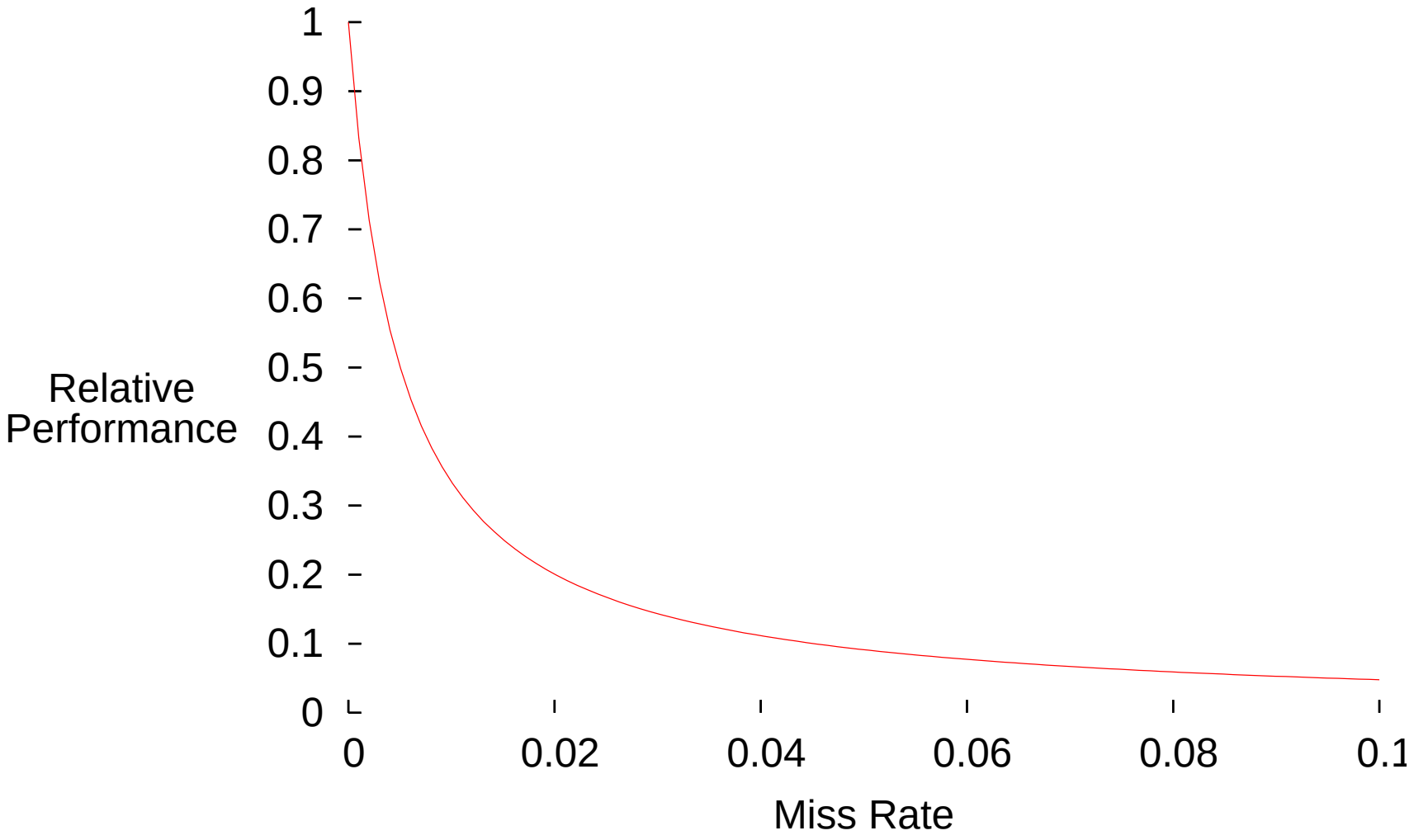
- *Write allocate*: put it in the cache
  - For most cache block sizes, means read miss for parts that weren't written
- *No write allocate*: just go straight to memory
  - Assumes no spatial or temporal locality
  - Causes excess memory traffic on multiple writes
  - Not very sensible with write-back caches

Optimizing Cache Performance

- Cache performance (especially miss rate) can have dramatic overall effects
- Near-100% hit rate means 1 clock per memory access
- Near-0% can mean 150-200 clocks
  - $\Rightarrow$ 200X performance drop!
- Amdahl's Law says you don't have to stray far from 100% to get big changes

Effect of Miss Rate

Performance effect of per-instruction miss rate, assuming 200-cycle miss penalty



Complications of Miss Rates

- Common to separate L1 instruction and data caches
  - Allows I-fetch and D-fetch on same clock
  - Cheaper than dual-porting L1 cache
- Separate caches have slightly higher miss rates
  - But net penalty can be less
  - Why?

The Math of Separate Caches

- Assume 200-clock miss penalty, 1-clock hit
- Separate 16K I- and D-caches can service both in one cycle
- Unified 32K cache takes extra clock if data & instruction on same cycle (i.e., any data access)

The Math of Separate Caches (cont'd)

- $MR_{16KI} = 0.004$ (from Fig. C.6)
- 36% of instructions transfer data, so $MR_{16KD} = .0409/0.36 = 0.114$
- 74% of accesses are instruction fetches, so $MR_{split} = .74 \times 0.004 + .26 \times 0.114 = 0.0326$
- Unified cache gets an access on every instruction fetch, plus another access when the 36% of data transfers happen, so $MR_U = .0433/1.36 = 0.0318$

$\Rightarrow$ Unified cache misses less

The Math of Separate Caches (cont'd)

- $MR_{16KI} = 0.004$ (from Fig. C.6)
- 36% of instructions transfer data, so $MR_{16KD} = .0409/0.36 = 0.114$
- 74% of accesses are instruction fetches, so $MR_{split} = .74 \times 0.004 + .26 \times 0.114 = 0.0326$
- Unified cache gets an access on every instruction fetch, plus another access when the 36% of data transfers happen, so $MR_U = .0433/1.36 = 0.0318$

$\Rightarrow$ Unified cache misses less

- BUT... Miss rate isn't what counts: only total time matters

The Truth about Separate Caches

- Average access time is given by:

$$\begin{aligned}\bar{t} &= \%\text{instrs} \times (t_{\text{hit}} + \text{MR}_\text{I} \times \text{Miss penalty}) \\ &\quad + \%\text{data} \times (t_{\text{hit}} + \text{MR}_\text{D} \times \text{Miss penalty})\end{aligned}$$

- For split caches, this is:

$$\begin{aligned}\bar{t} &= 0.74 \times (1 + 0.004 \times 200) \\ &\quad + 0.26 \times (1 + 0.114 \times 200) \\ &= .74 \times 1.80 + .26 \times 23.80 = 7.52\end{aligned}$$

- For a unified cache, this is:

$$\begin{aligned}\bar{t} &= 0.74 \times (1 + 0.0318 \times 200) \\ &\quad + 0.26 \times (1 + 1 + 0.0318 \times 200) \\ &= .74 \times 7.36 + .26 \times 8.36 = 7.62\end{aligned}$$

Out-of-Order Execution

- How to define miss penalty?
  - Full latency of fetch?
  - “Exposed” (nonoverlapped) latency when CPU stalls?
  - We’ll prefer the latter
  - ... But how to decide when stall happens?
- How to measure miss penalty?
  - Very difficult to characterize as percentages
  - Slight change could expose latency elsewhere

Six Basic Cache Optimizations

$$\text{Average access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

- Six “easy” ways to improve above equation
- Reducing miss rate:
  1. Larger block size (compulsory misses)
  2. Larger cache size (capacity misses)
  3. Higher associativity (conflict misses)
- Reducing miss penalty:
  4. Multilevel caches
  5. Prioritize reads over writes
- Reducing hit time:
  6. Avoiding address translation

Types of Misses

Uniprocessor misses fall into three categories:

- *Compulsory*: First access (ever to location)
- *Capacity*: Program accesses more than will fit in cache
- *Conflict*: Too many blocks map to given set
  - Sometimes called *collision* miss
  - Can't happen in fully associative caches

Reducing Compulsory Misses

1. Use larger block size
   - Brings in more data per miss
   - Unfortunately can increase conflict and capacity misses
2. Reduce invalidations
   - Requires OS changes
   - PID in cache tag can help

Reducing Capacity Misses

- Increase size of cache
  - Higher cost
  - More power draw
  - Possibly longer hit time

Reducing Conflict Misses

- Increase Associativity
  - Gives CPU more places to put things
  - Increases cost
  - Slows CPU clock
    - May outweigh gain from reduced miss rate
    - Sometimes direct-mapped may be better!

Multilevel Caches

$$\text{Average access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

- Halving miss penalty gives same benefit as halving miss rate
  - Former may be easier to do
- Insert L2 cache between L1 and memory
  - To first approximation, everything hits in L2
  - Also lets L1 be smaller and simpler (i.e., faster)
- Problem: L2 only sees "bad" memory accesses
  - Poor locality
  - Poor predictability

  $\Rightarrow$ L2 must be significantly bigger than L1

Analyzing Multilevel Caches

$$
\begin{aligned}
\text{Average access time} &= \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1} \\
&= \text{Hit time}_{L1} \\
&\quad + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} \\
&\quad\quad + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})
\end{aligned}
$$

We're back where we started, except now all L2 badness is reduced by a factor of Miss rate_{L1}. But now Miss rate_{L2} is much higher because of "bad" accesses.

Multilevel Inclusion

- Should L2 contain everything that's in L1?
- Pro: SMP can talk just to L2
- Con: Block sizes must be identical or complexity rises
  1. L1 caches A0, L2 caches A0 and A1 together
  2. CPU miss on B1 replaces A1 in L1, both in L2
  3. Inclusion now violated! L1 has A0, B1; L2 has B0, B1
- Con: If inclusion violated, SMP coherency more complicated
- Alternative: explicit *exclusion*
  - If block moved into L1, evicted from L2
- Upshot: L1 and L2 must be designed together

Read Priority

- Memory system can buffer and delay writes
  - CPU can continue working

. . . but CPU *must* wait for reads to complete

- Obvious fix: don't make reads wait for writes
- Problem: Introduces RAW hazards
  - Must now check write buffer for dirty blocks

Virtually Addressed Caches

- Address translation takes time
- Can't look up in cache until translation is done
- Solutions:
  1. Make sure set index comes from "page offset" portion of virtual address
     - Can then look up set and fetch tag during translation
     - Only wait for comparison (must do in any case)
     - Can fetch data on assumption comparison will succeed
     - Limits cache size (for given way)
  2. Index cache directly from virtual address
     - Aliasing becomes problem
     - If aliasing is limited, can check all possible aliases (e.g., Opteron)
     - *Page coloring* requires aliases to differ only in upper bits
       $\Rightarrow$ All aliases map to same cache block

Summary of Cache Optimizations

Technique	Hit time	Miss penalty	Miss rate	HW cost	Comment
Large block size		−	+	0	Trivial: P4 L2 is 128B
Large cache size	−		+	1	Widely used, esp. L2
High associativity	−		+	1	Widely used
Multilevel caches		+		2	Costly but widely used
Read priority		+		1	Often used
Addr. translation	+			1	Often used