18-447 Lecture 17:
Memory Hierarchy: Cache Design

James C. Hoe
Dept of ECE, CMU
March 24, 2009

Announcements: Project 3 is due
Midterm 2 is coming

Handouts: Practice Midterm 2 solutions

The problem (recap)

- Potentially $M=2^m$ bytes of memory, how to keep the most frequently used ones in $C$ bytes of fast storage where $C \ll M$
- Basic issues (intertwined)
  1. where to “cache” a memory location?
  2. how to find a cached memory location?
  3. granularity of management: large, small, uniform?
  4. when to bring a memory location into cache?
  5. which cached memory location to evict to free-up space?
- Optimizations
Basic Operation

address

(2)
cache lookup

hit?

yes

no

return data

update cache

fetch new from \( L_{i+1} \)

evict old to \( L_{i+1} \)

occupied?

yes

no

(1, 3, 5)

choose location

Ans to (4): memory location brought into cache “on-demand”. What about prefetch?

Direct-Mapped Cache (v1)

\[ \text{let } t = \lg_2 M - \lg_2 (C) \]

\( \lg_2 M \)-bit address

\[
\begin{array}{c|c|c|c|c}
\text{tag} & \text{idx} & g & \text{Tag Bank} & \text{Data Bank} \\
\hline
\text{C/G lines by} & \text{valid} & \text{C/G lines by} & \text{G bytes} \\
\text{t bits} & \text{t bits} & \text{G bytes} & \\
\text{\( \lg_2 (C/G) \) bits} & & & \\
\end{array}
\]

What about writes?
Storage Overhead

- For each cache block of $G$ bytes, must also store additional “$t+1$” bits where $t = \log_2 M - \log_2 C$
  - if $M = 2^{32}$, $G = 4$, $C = 16K = 2^{14}$
    $\Rightarrow t = 18$ bits for each 4-byte block
    60% storage overhead
    16KB cache really needs 25.5KB of SRAM
- Solution: let multiple $G$-byte words share a common tag
  - each $B$-byte block holds $B/G$ words
  - if $M = 2^{32}$, $B = 16$, $G = 4$, $C = 16K$
    $\Rightarrow t = 18$ bits for each 16-byte block
    15% storage overhead
    16KB cache needs 18.4KB of SRAM
  15% of 16KB is small, 15% of 1MB is 152KB
  $\Rightarrow$ larger block size for lower/larger hierarchies

Direct-Mapped Cache (final)

\[
\text{lg}_2 M \text{-bit address} \quad \begin{array}{cc|c|c|c|c} \text{tag} & \text{idx} & \text{bo} & g \\
\end{array}
\]

- $t = \log_2 M - \log_2 C$
- $\log_2 (C/B)$ bits
- $\log_2 (B/G)$ bits

- $\begin{array}{c} \text{Tag Bank} \\ \text{C/B-by-t bits} \end{array}$
- $\begin{array}{c} \text{valid} \\ \text{Data Bank} \\ \text{C/B-by-B bytes} \end{array}$

$G$ bytes

hit?
data

$B$ bytes

let $t = \log_2 M - \log_2 C$
Direct-Mapped Cache

- $C$ bytes of storage divided into $C/B$ blocks
- A block of memory is mapped to one particular cache block according to the address' block index field
- All addresses with the same block index field map to the same cache block
  - $2^t$ such addresses; can cache only one such block at a time
  - even if $C >$ working set size, collision is possible
  - given 2 random addresses, chance for collision is $1/(C/B)$

Notice likelihood for collision decreases with increasing number of cache blocks ($C/B$)

Block Size and $m_i$

- Bytes that share a common tag are all-in or all-out
- Loading a multi-word block at a time has the effect of prefetching for spatial locality
  - pay miss penalty only once per block
  - works especially well in instruction caches
  - effective up to the limit of spatial locality
- But, increasing block size (while holding $C$ constant)
  - reduces the number of blocks
  - increases possibility for collision
Block Size and $T_{i+1}$

- Loading a large block can increase $T_{i+1}$
  - if I want the last word on a block, I have to wait for the entire block to be loaded
- solution 1 critical-word first reload
  - $L_{i+1}$ returns the requested word first then rotate around the complete block
  - supply requested word to pipeline as soon as available
- solution 2: sub-blocking
  - individual valid bits for different sub-blocks
  - reload only requested sub-block on demand
  - note: all sub-blocks stall share common tag

```
tag v s-block0 v s-block1 v s-block2  •••••
```

Test yourself: What is wrong with this?

let $t = \lg_2 M - \lg_2 (C)$
Direct-Mapped Cache (double check)

Data Bank

Tag Bank

Let $t = \log_2 M - \log_2 (C)$

"a"-way Set Associative Cache

Let $t = \log_2 M - \log_2 (C/a)$
"a"-way Set-Associative Cache

- C bytes of storage divided into a banks each with C/a/B blocks
- Requires a comparators and a-to-1 multiplexer
- An address is mapped to a particular block in a bank according its block index field, but there are a such banks (together known as a "set")
- All addresses with the same block index field map to a common "set" of cache blocks
  - 2^t such addresses: can cache a such blocks at a time
  - if C > working set size
    higher-degree of associatively ⇒ fewer collisions

Replacement Policy for Set Associative Caches

- A new cache block can evict any of the cached memory block in the same set, which one?
  - pick the one that is least recently used (LRU)
    simple function for a=2, complicated for a>2
  - pick any one except the most recently used
  - pick the most recently used one
  - pick one based on some part of the address bits
  - pick the one that you will need again furthest in the future
  - pick a (pseudo) random one

- Replacement policy only has second-order effect
  - if you actively use less than a blocks in a set, any sensible replacement policy will quickly converge
  - if you actively use more than a blocks in a set, no replacement policy can help you
Pseudo-Associativity:
e.g MIPS R10K 2-way L2

- a-way associativity is a placement policy
  - it says an address could be mapped to different locations in the cache
  - *** it doesn’t say lookups must be done in parallel banks

Pseudo a-way associativity:
- given a direct-mapped array with C/B cache blocks
- implement C/B/a sets
- given an address A, sequentially search:
  - \{0, A[\lg(C/B/a)-1: \lg(B)]\}, \{1, A[\lg(C/B/a)-1: \lg(B)]\}, ...
    - \{a-1, A[\lg(C/B/a)-1: \lg(B)]\}
- Optimization: record the most recently used way (MRU) to look up first

How does this compare with true associative caches?

Fully Associative Cache: \(a=C/B\)

let \(t=\lg_2 M - \lg_2(B)\)
Fully Associative Cache: $a=C/B$

- A “content-addressable” memory
  - not your regular SRAM
  - present a tag, return the block with matching tag, or else miss
  - no index bits used in lookup
- Any address can go into any of the $C/B$ blocks
  - if $C >$ working set size, no collisions
- Requires 1 comparator per cache block, a huge multiplexer, and many long wires
  - considered exorbitantly expensive/difficult for more than 32~64 blocks at L1 latencies.

Fortunately, there is little reason for very large fully associative caches. For any reasonably large values of $C/B$, $a=4$--$5$ is as good as $a=C/B$ for typical programs.

Recap: Basic Cache Parameters

- Let $M=2^m$ be the size of the address space in bytes
  - sample values: $2^{32}, 2^{64}$
- Let $G=2^g$ be the cache access granularity in bytes
  - sample values: 4, 8

- Let $C$ be the “capacity” of the cache in bytes
  - sample values: 16 KBytes (L1), 1 MByte (L2)
- Let $B=2^b$ be the “block size” of the cache in bytes
  - sample values: 16 (L1), >64 (L2)
- Let $a$ be the “associativity” of the cache
  - sample values: 1, 2, 4, 5(?),... “$C/B$”
Recap: Address Fields

$\lg_2 M$ -bit address

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>B.O.</th>
</tr>
</thead>
</table>

Recap: $M=32$, $G=$_______, $C=$_______, $B=$_______, $a=$_______
Classification of Cache Misses

- **Compulsory miss** (design factor: B and prefetch)
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below
  - dominates when locality is poor

- **Capacity miss** (design factor: C)
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache of the same capacity
  - dominates when C < W

- **Conflict miss** (design factor: a)
  - data displaced by collision under direct-mapped or set-associative allocation
  - defined as any miss that is neither a compulsory nor a capacity miss
  - dominates when C=W or when C/B is small

![Hit Rate vs. B](image)
Classification of Cache Misses

- **Capacity miss** (design factor: \(C\))
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache of the same capacity
  - dominates when \(C < W\)
    - for example, the L1 cache can never be made big enough due to cycle-time tradeoff

![Graph showing hit rate vs. working set size](image)

- **Conflict miss** (design factor: \(a\))
  - data displaced by collision under direct-mapped or set-associative allocation
  - defined as any miss that is neither a compulsory nor a capacity miss
  - dominates when \(C = W\) or when \(C/B\) is small

![Graph showing hit rate vs. design factor](image)