18-447 Lecture 16: Cache Design in Context (Uniprocessor)

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – understand cache design and operation in context
  – focus on uniprocessor for now

• Notices
  – HW 4, out next week
  – Lab 3, due next week
  – Final Exam, Fri, May 3rd, 1pm
  – Midterm regrade due Monday 4/3 noon

  *Follow Canvas instructions carefully!!*

• Readings
  – P&H Ch 5
Recap: Basic Cache Parameters

- **M = \(2^m\)**: size of address space in bytes
  - example values: \(2^{32}\), \(2^{64}\)

- **G = \(2^g\)**: cache access granularity in bytes
  - example values: 4, 8

- **C**: “capacity” of cache in bytes
  - example values: 16 KByte (L1), 1 MByte (L2)

- **B = \(2^b\)**: “block size” in bytes
  - example values: 16 (L1), >64 (L2)

- **a**: “associativity” of the cache
  - example values: 1, 2, 4, 5(?),... “C/B”

- “map”: addr to idx and b.o.
  - C/a should be a 2-power
Recap: Address Map for Typical Locality

$\log_2 M$-bit address

tag | index | B.O.

tag: $\log_2 M$ - $\log_2 (C/a)$ bits

block index: $\log_2 (C/a)/B$ bits

block offset: $\log_2 (B/G)$ bits

byte offset: $\log_2 G$ bits
\[ M = 2^{32}, \ a = 2, \ C = 1K, \ B = 4, \ G = 2 \]
$M = 2^{32}$, $a = 2$, $C = 1K$, $B = 4$, $G = 2$: “textbook” solution

<table>
<thead>
<tr>
<th>tag</th>
<th>idx</th>
<th>b.o.</th>
</tr>
</thead>
</table>

Diagram of the solution:

- Input tags are compared with the tag0 and tag1 fields.
- If a match is found, the corresponding data is selected.
- The selected data is then passed through a 2-1 mux to DATA.
Same cache parameters but tune for “narrower” data SRAM banks

Can you make the tag SRAMs taller/narrower also?
Same cache parameters but tune for “fatter” data SRAM banks

Can you make the tag SRAMs shorter/wider also?
Same cache parameters but each block frame is interleaved over 2 SRAM banks

- **tag**: PA[31:9]  
- **idx**: PA[8:2]  
- **b.o.**: PA[1] PA[0]

```
<table>
<thead>
<tr>
<th>idx</th>
<th>b.o.</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
```

```
tag0  v0
128 x  23-b
```

```
tag1  v1
128 x  23-b
```

- **row-sel**
- **col-sel**

```
<table>
<thead>
<tr>
<th>row-sel</th>
<th>col-sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx 7</td>
<td>idx 7</td>
</tr>
</tbody>
</table>
```

```
data 0
128-rows x 4-bytes
```

```
data 1
128-rows x 4-bytes
```

```
h0 • bo + h1 • bo
```

```
h1 • bo + h0 • bo
```

```
2-1-mux
```

```
2-1-mux
```

```
h0
HIT
```

```
h1
```

```
16
DATA
```

This part is unchanged.
**3’C worksheet: a=1, B=1, C=2, G=1**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0</td>
<td>compulsory</td>
<td>[ -, - ] → [ 0, - ]</td>
<td>{ } → { 0 }</td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 3’C worksheet: \(a=1, B=1, C=2, G=1\)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0</td>
<td>compulsory</td>
<td>[-,-] (\rightarrow) [0,-]</td>
<td>{ } (\rightarrow) {0}</td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
<td>compulsory</td>
<td>[0,-] (\rightarrow) [2,-]</td>
<td>{0} (\rightarrow) {0,2}</td>
</tr>
<tr>
<td>0x0</td>
<td>0</td>
<td>conflict</td>
<td>[2,-] (\rightarrow) [0,-]</td>
<td>{0,2}_hit</td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
<td>conflict</td>
<td>[0,-] (\rightarrow) [2,-]</td>
<td>{0,2}_hit</td>
</tr>
<tr>
<td>0x1</td>
<td>1</td>
<td>compulsory</td>
<td>[2,-] (\rightarrow) [2,1]</td>
<td>{0,2} (\rightarrow) {0,1}</td>
</tr>
<tr>
<td>0x0</td>
<td>0</td>
<td>conflict</td>
<td>[2,1] (\rightarrow) [0,1]</td>
<td>{0,1}_hit</td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
<td>capacity</td>
<td>[0,1] (\rightarrow) [2,1]</td>
<td>{0,1} (\rightarrow) {0,2}</td>
</tr>
<tr>
<td>0x0</td>
<td>0</td>
<td>conflict</td>
<td>[2,1] (\rightarrow) [0,1]</td>
<td>{0,2}_hit</td>
</tr>
</tbody>
</table>
The Cache and You
(simple, single core from Lab)
The Context

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Programmer-Visible State
(aka Architectural State)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Adding Caches to In-order Pipeline

• On I-fetch and LW assuming 1-cyc SRAM lookup
  – if hit, just like magic memory
  – if miss, stall pipeline until cache ready

• On SW also assuming 1-cycle SRAM lookup
  – if miss, stall pipeline until cache ready *(must we??)*
  – if hit, ??? . . .

• For SW, need to check tag array to ascertain hit before committing to write data array
  – data array write happens in the next cycle
  – if SW is followed immediately by LW

⇒ structural hazard on data array ⇒ stall, *whom?*
Store Buffer

• Why stall when memory port is usually free?
• After tag array hit, buffer SW address and data until next free data array cycle (not used by LW)
  – younger LW keep going (reorder w. buffered SW)
  – Must not evict buffered SW’s target cache block
• Memory dependence and forwarding
  – younger LW must check against pending SW-addresses in store buffer (CAM) for RAW dependence

![Diagram of Store Buffer](image)
Must wait for a miss? (uniprocessor)

- In-order pipeline must stall for LW-miss
- Younger instructions can move ahead of SW-miss
  - except LW to same address; if so, stall or forward
  - additional SW-misses to same and different addr’s can be “completed” from pipeline’s view
- Modern out-of-order execution supports non-blocking miss handling for both LW and SW
  - too expensive to stall *(CPU/memory speed gap)*
  - significant complexity in
    - detecting and resolving memory dependencies
    - constructing precise exception state
Details and more details when building a cache for real
Basic Operation
Ans (1): demand-driven

Can cache decide to prefetch an addr without a miss first? when and which?

M address

hit?

(1)

hit?

(1')

return data

update cache

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

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

occupied?

occupied?

no

no

yes

yes

(2) how?

(3) how?

choose location

choose location

select location

select location

Can cache decide to prefetch an addr without a miss first? when and which?
Write-Through Cache

- On write-hit in \( L_i \), should \( L_{i+1} \) be updated?
- If yes, \( L_i \) is write-through
  - simple management (discard on replacement)
  - external agents (DMA and other proc’s) see up-to-date values in \( L_{i+1} \) (e.g., DRAM)
- With write-through, on a write-miss, should a cache block be allocated in \( L_i \) (aka write-allocate)?

------------------------

- Write-through to DRAM not viable today
  3.0GHz, IPC=2, 10% SW, ~8byte/SW \( \Rightarrow \) ~5GB/s/core
  L1 (w. parity) write-through to L2 (w. ECC) is in use
Write-Back Cache

• Hold changes in $L_i$ until block is displaced to $L_{i+1}$
  – on read or write miss, entire block is brought into $L_i$
  – LWs and SWs hit in $L_i$ until replacement
  – on replacement, $L_i$ copy written back out to $L_{i+1}$
    adds latency to load miss stall

• “Dirty” bit optimization
  – keep per-block status bit to track if a block has been modified since brought into $L_i$
  – if not dirty, no write-back on replacement

• What if a DMA device wants to read a DRAM location with a dirty cached copy?
  How to find out? How to access?
Write-Back Cache and DMA

- DRAM not always up-to-date if write-back
- DMA should see up-to-date value (aka, cache coherent)
- Option 1: SW flushes whole cache or specific blocks before programming DMA
- Option 2: cache monitors bus for external requests
  - ask request to a dirty location to "retry"
  - write out dirty copy before request is repeated
Idempotency and Side-effects

• Loading from real memory location $M[A]$ should return most recent value stored to $M[A]$
  ⇒ writing $M[A]$ once is the same as writing $M[A]$ with same value multiple times in a row
  ⇒ reading $M[A]$ multiple times returns same value

  This is why memory caching works!!

• LW/SW to mmap locations can have side-effects
  – reading/writing mmap location can imply commands and other state changes
  – e.g., a mmap device that is a FIFO
    • SW to 0xffff0000 pushes value
    • LW from 0xffff0000 returns popped value

What happens if 0xffff0000 is cached?
Programmer-Visible State
(aka Architectural State)

- PC
- Instruction
- Memory

- Read register 1
- Read register 2
- Write register
- Read data 1
- Read data 2

- RFWrite
- MemRead
- MemWrite

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

think interfaces not modules
Harvard vs Princeton Architecture

- Historically
  - “Harvard” referred to Aiken’s Mark series with separate instruction and data memory
  - “Princeton” referred to von Neumann’s unified instruction and data memory

- Contemporary usage: split vs unified “caches”
- L1 I/D caches commonly split and asymmetrical
  - double bandwidth and no-cross pollution on disjoint I and D footprints
  - I-fetch smaller footprint, high-spatial locality and read-only ⇒ I-cache smaller, simpler

  what about self-modifying code?

- L2 and L3 are unified for simplicity
Multi-Level Caches

L1-I

L1-D

L2-Unified

DRAM

- a few pclk latency
- many GB/sec on random word accesses

Intermediate cache levels bridge latency and bandwidth gap between L1 and DRAM

- hundreds of pclk latency
- ~GB/sec on sequential block accesses

On-chip or off-chip?
aBC of Multi-Level Cache Design

• Upper-level caches (L1)
  – small $C$: upper-bound by SRAM access time
  – smallish $B$: upper-bound by $C/B$ effects
  – $a$: required to counter $C/B$ effects

• Lower-level caches (L2, L3, etc.)
  – large $C$: upper-bound by chip area
  – large $B$: to reduce tag storage overhead
  – $a$: upper bound by complexity and speed

• New very large (10s MB) on-chip caches on are distributed structures
  – same basic notions of ways and sets
  – but they don’t look or operate anything like “textbook”
Modern Last-Level Cache (LLC)

- Disaggregated, asynchronous; partitioned by address; shared by all cores within a socket
- Hold, fast “coherent” copies of local and remote DRAM locations

Departure from classic uniproc. hierarchy
Inclusion Principle

- Classically, $L_i$ contents is always a subset of $L_{i+1}$
  - if an address is important enough to be in $L_i$, it must be important enough to be in $L_{i+1}$
  - external agents (DMA and other proc’s) only have to check the lowest level to know if an address is cached—do not need to consume L1 bandwidth

- Inclusion no longer taken as a given
  - nontrivial to maintain if $L_{i+1}$ has lower associativity
  - too much redundant capacity in multicore with many per-core $L_i$ and shared $L_{i+1}$
  - Last-level cache “directories” track cached addr
Inclusion Violation Example

x, y, z have same L1 idx bits
y, z have the same L2 idx bits
x, {y, z} have different L2 idx bits
Aside: Victim “Cache”

- High-associativity is an expensive solution to avoid conflicts in a few sets only
- Augment a low-associative main cache with a very small but fully associative victim cache
  - blocks evicted from main cache is first held in victim cache
  - if an evicted block is referenced again soon, it is returned to main cache
  - if an evicted block doesn’t get referenced again, it will eventually be displaced from victim cache to next level

Plays a different role outside of standard memory hierarchy stacking
Aside: Software-Assists

- Separate “temporal” vs “non-temporal” hierarchy
  - exposed in the ISA (e.g., Intel IA64 below)
  - load and store instructions include *hints* about where to cache on a cache miss
  - “*hint*” only so implementation could support a subset or none of the levels and actions

```
   temporal
     L1 -> L2 -> L3

  non-temporal-L1
     NT L1
     L1 -> NT L2
     L2 -> NT L3
     L3

  non-temporal-L2
     NT L2
     L2 -> NT L3
     L3

  non-temporal-All
     NT L1
     L1 -> NT L2
     L2 -> NT L3
     L3
```

18-447-S24-L16-S32, James C. Hoe, CMU/ECE/CALCM, ©2024
Test yourself

What cache is in your computer?

• How to figure out what cache configuration is in your computer
  – capacity (C), associativity (a), and block-size (B)
  – number of levels
• The presence or lack of a cache should not be detectable by functional behavior of software
• But you could tell if you measured execution time to infer the number of cache misses
Capacity Experiment: assume 2-power $C$

- For increasing Range = 1, 2, 4, 8, 16, ...
  - allocate a buffer of size $R$
  - repeatedly {read every byte in buffer in sequence}
  - measure average read time in steadystate

- Analysis
  - for small $R \leq C$, expect all reads to hit
  - for large $R > C$, expect reads to miss and detect corresponding jump in average memory access time

- If continuing to increase $R$, read time jumps again when buffer size spills out to next cache level

Warning: timing won’t be perfect when you try this
Block Size Experiment: knowing $C$

- Allocate a buffer of size $R \gg C$
- For increasing $S=1,2,4,8,\ldots$,
  - repeatedly \{read every $S$’th byte in buffer in sequence\}
  - measure average read time in steadystate
- Analysis
  - since $R \gg C$, expect first read to a block to miss when revisiting a block
  - reads to same block in same round should hit
  - expect increasing average read time for increasing $S$ until $S \geq B$ (no reuse in block)
Associativity Experiment: knowing C

- For increasing \( R \), where \( R \) is a multiple of \( C \)
  - allocate a buffer of size \( R \)
  - repeatedly \{read every \( C \)’th byte in buffer in sequence\}

- Analysis
  - all \( R/C \) references map to the same set
  - for small \( R \) s.t. \( (R/C) \leq a \), expect all reads to hit
  - for large \( R \) s.t. \( (R/C) > a \), expect some reads to miss since touching more addresses than ways

note: 100% cache miss if LRU is used

How to detect associativity for lower-level caches?
Know your cache

• What else can you tell?
  – write-back vs write-through/write-allocate
  – unified vs. split design
  – I-cache C, B, a
  – $t_i$
  – replacement policy of associative caches

• Same mental exercise is required to control cache use in performance tuning

Caveat: experiments may not predict behaviors exactly for modern CPUs with virtual memory, complex hierarchies, and prefetchers