CS425
Computer Systems Architecture

Fall 2017

Caches: The Basics
Who Cares about Memory Hierarchy?

Processor-DRAM Memory Gap (latency)

- Processor-Memory Performance Gap: (grows 50% / year)
- "Moore’s Law"
- DRAM 9%/yr. (2X/10 yrs)
- μProc 60%/yr. (2X/1.5yr)

Time

Performance

Latency lags bandwidth

Table 1. Performance milestones in bandwidth and latency for processors, memory modules, local area networks, and disks [3, 5].

Reasons for Bountiful Bandwidth but Lagging Latency

“There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can’t bribe God.”

—Anonymous
Memory abstraction in architecture

**Addressable memory**

- Association between address and values in storage
- Addresses index bytes in storage
- Values aligned in multiples of word size
- Accessed through sequence of reads and writes
- Write binds value to address
- Read returns most recent value stored in address

**Generic memory**

- Command (RD/WR)
- Address (name)
- Data (WR)
- Data (RD)
- Done
Levels of Memory Hierarchy

Capacity
Access Time
Cost

CPU Registers
100s Bytes
<10ns ns

Cache
K Bytes
10-100ns
1-0.1 cents/bit

Main Memory
M Bytes
200ns-500ns
$.0001-.000001 cents/bit

Disk
G Bytes, 10 ms
(10,000,000ns)
10^-5 - 10^-6 cents/bit

Tape
infinite
sec-min
10^-8

Upper Level
faster

Staging Xfer Unit
prog./compiler
1-8 bytes

cache ctrl
8-128 bytes

OS
512-4K bytes

Lower Level
Larger

Registers
Instr. Operands

Cache
Blocks

Memory
Pages

Disk

Files

Tape
Definition of Cache

Definition

- First level of memory hierarchy after registers
- Any form of storage that buffers temporarily data
  - OS buffer cache, name cache, Web cache, …
- Designed based on the principle of locality
  - **Temporal locality**: Accessed item will be accessed again in the near future
  - **Spatial locality**: Consecutive memory accesses follow a sequential pattern, references separated by unit stride
Cache on DLX
Memory Hierarchy: Apple iMac G5 (2005)

Goal: Illusion of large, fast, cheap memory
Let programs address a memory space that scales to the disk size, at a speed that is usually nearly as fast as register access

iMac G5 1.6 GHz clock, 55 ns DRAM vs. Apple II 1 MHz, 400ns DRAM
Perform: CPU 1600 X, DRAM 7.3 X faster in 27 yrs => 2X/2.5y, 9.3y
PowerPC 970 (G5): All caches on-chip
Locality

Spatial locality

- Appears due to iterative execution and linear data access patterns
- Exploited by using larger block sizes – data to be used prefetched with block
- Exploited by data and code transformations by the compiler
- Exploited by unit-stride prefetching mechanisms and policies

Temporal locality

- Appears due to iterative execution and data reuse
- Exploited by caches, through which data is reused
- **Working set**: data that needs to be kept cached in a window of time to maximize locality
- **Reuse distance**: number of blocks of memory accessed between two consecutive accesses to same block
Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper level
  - Hit Rate: the fraction of memory accesses found in the upper level
  - Hit Time: Time to access the upper level which consists of
    - Time to determine hit/miss

- **Miss**: data needs to be retrieved from a block in the lower level
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block to the upper level

- **Hit Time << Miss Penalty** (=500 instructions on 21264!)
Cache Hit

Cache hit Block X

To processor

From processor

upper-level memory

Block X

lower-level memory

Block Y
Cache Miss

Cache miss Block X

To processor

From processor

upper-level memory

Block X

lower-level memory

Block Y
Cache Measures

- **Hit rate:** fraction found in that level
  - So high that usually talk about Miss rate = 1 - Hit rate
  - Miss rate fallacy: as MIPS to CPU performance, miss rate to AMAT in memory

- **AMAT = Hit time + Miss rate x Miss penalty (ns or clocks)**

- **Miss penalty:** time to supply a missed block from lower level, including any CPU-visible delays to save replaced write-back data to make room in upper level cache. {“All active caches are full”}
  - **access time:** time to lower level = $f$ (latency to lower level)
  - **transfer time:** time to transfer block = $f$ (BW between upper & lower levels)
  - **replacement time:** time to make upper-level room for new block, if all active caches are full
Average Memory Access Time (AMAT)

**AMAT components**

\[
\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \\
\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \\
\quad \times \text{Clock cycle time}
\]

\[
\text{CPU time} = IC \times \left( CPI_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}
\]

\[
\text{CPU time} = IC \times \left( CPI_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \\
\quad \times \text{Clock cycle time}
\]

Assuming that cache hits do not stall the machine!
An example

- Assumption on computer A
  - CPI = 1.0 when all memory accesses hit
  - Data accesses are only loads and stores (explain 50% of insts.)
  - Miss penalty: 25 cc
  - Miss rate: 2%

- Compute the speedup of computer B, for which all cache accesses are hit

\[
\text{exec}_{t_B} = (CPU_{cc} + MemStall_{cc}) \times \text{Clock cycle time} \\
= (IC \times CPI + 0) \times cct = IC \times 1.0 \times \text{Clock cycle time}
\]

\[
MemStall_{cc} = IC \times \frac{MemAccess}{Instruction} \times MissRate \times MissPenalty \\
= IC \times (1 + 0.5) \times 0.02 \times 25 = IC \times 0.75
\]

\[
\text{exec}_{t_A} = (CPU_{cc} + MemStall_{cc}) \times \text{Clock cycle time} \\
= (IC \times CPI + IC \times 0.75) \times \text{Clock cycle time} \\
= IC \times 1.75 \times \text{Clock cycle time}
\]
4 Questions for Memory Hierarchy

For a given level of the memory hierarchy

- **Q1**: Where can a block be placed in the upper level? (Block placement)
- **Q2**: How is a block found if it is in the upper level? (Block identification)
- **Q3**: Which block should be replaced on a miss? (Block replacement)
- **Q4**: What happens on a write? (Write strategy)
Q1: Where to Place Blocks?

- Jargon: Each address of a memory location is partitioned into:
  - block address
    - tag
    - index
  - block offset

Fig. C.3
Simplest Cache: Direct Mapped

*Use Index in Address to find Cache Location*

- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address $<1:0>$ => cache index

- Which one should we place in the cache?
- How can we tell which one is in the cache?
1 KB Direct Mapped Cache, 32B blocks

- For a $2^{N}$ byte cache:
  - The uppermost $(32 - N)$ bits are always the Cache Tag
  - The lowest $M$ bits are the Byte Select (Block Size = $2^{M}$)

```
<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Example: 0x50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex: 0x01</td>
<td>Ex: 0x00</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Cache Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x50</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 31</td>
</tr>
<tr>
<td>Byte 63</td>
</tr>
<tr>
<td>Byte 1023</td>
</tr>
<tr>
<td>Byte 0</td>
</tr>
<tr>
<td>Byte 1</td>
</tr>
<tr>
<td>Byte 32</td>
</tr>
<tr>
<td>Byte 33</td>
</tr>
<tr>
<td>Byte 992</td>
</tr>
</tbody>
</table>
```

Stored as part of the cache “state”
Direct Mapped Cache

**Advantages**
- Simple, low complexity, low power consumption
- Fast hit time
- Data available before cache determines hit or miss
  - Hit/miss check done in parallel with data retrieval

**Disadvantages**
- Conflicts between blocks mapped to same block in cache
Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel (N typically 2 to 4)
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result
Two-way Set Associative Cache

Advantages

- Choice of mapping memory block to different cache blocks in a set
  - LRU or other policies for good selection of victim blocks
- Reduction of conflicts

Disadvantages

- Increased complexity – comparators, multiplexor, parallel tag comparison
- Increased power consumption
- Increased hit time, due to comparators and multiplexor
- Data available after cache determines hit or miss
Cache Mapping Example

Mapping block 12 from RAM in 8-block cache

Number of sets = #Blocks / Associativity
Set/Index = (Block Address) MOD (Number of sets in cache)
Q2: How is a block found in the cache

<table>
<thead>
<tr>
<th>Cache tag array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Address</td>
</tr>
<tr>
<td>Tag</td>
</tr>
</tbody>
</table>

- Index points to line in data array – one block or set
- Offset points to byte in block
- Tag compared against tag field in address
- Valid bit ORed with output of tag comparator
Q3: Which block is replaced on a miss

- Easy if direct-mapped (only 1 block “1 way” per set index)
- Three common choices for set-associative cache:
  - Replace an eligible *random* block
  - Replace the least recently used (LRU) block
    - can be hard to keep track of, so often only approximated
  - Replace the oldest eligible block (First In, First Out, or FIFO)
- SPEC2000 benchmark (*misses per 1000 instructions*)

<table>
<thead>
<tr>
<th>Set associativity</th>
<th>Two-way</th>
<th>Four-way</th>
<th>Eight-Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Random</td>
<td>FIFO</td>
</tr>
<tr>
<td>16KB</td>
<td>114.1</td>
<td>117.3</td>
<td>115.5</td>
</tr>
<tr>
<td>64KB</td>
<td>103.4</td>
<td>104.3</td>
<td>103.9</td>
</tr>
<tr>
<td>256KB</td>
<td>92.2</td>
<td>92.1</td>
<td>92.5</td>
</tr>
</tbody>
</table>

 *(From Sussman)*
Q4: What happens on a write?

<table>
<thead>
<tr>
<th>Policy</th>
<th>Write-Through</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data word written to cache block is also written to next lower-level memory</td>
<td>Write new data word only to 1 cache block</td>
</tr>
<tr>
<td></td>
<td>Example, instr. sw to L1$ also goes to L2$</td>
<td>Update lower level just before a written block leaves cache, so not lose true value</td>
</tr>
<tr>
<td>Debugging</td>
<td>Easier</td>
<td>Harder</td>
</tr>
<tr>
<td>Can read misses force writes?</td>
<td>No</td>
<td>Yes (used to slow some reads; now write-buffer)</td>
</tr>
<tr>
<td>Do repeated writes touch lower level?</td>
<td>Yes, memory busier</td>
<td>No</td>
</tr>
</tbody>
</table>

Two options on a write miss:
- *Fetch line from lower-level and perform write hit (“write allocate”)*
- *Perform write only to the lower-level cache (“no-write allocate”)*
Write Buffers for Write-Through Caches

Q. Why a write buffer?
   A. So CPU doesn’t stall

Q. Why a buffer, why not just one register?
   A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?
   A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Q. Can Write Buffer work with Write-Back Cache?
   A. Yes. Send a block in the write-buffer on each write-back.
Write Buffer Optimization: Write Combine Buffer

- Write buffer mechanics, **with merging**
  - An entry may contain multiple words (maybe even a whole cache block)
  - If there’s an empty entry, the data and address are written to the buffer, and the CPU is done with the write
  - If buffer contains other modified blocks, check to see if new address matches one already in the buffer – if so, combine the new data with that entry
  - If buffer full and no address match, cache and CPU wait for an empty entry to appear (meaning some entry has been written to main memory)
  - Merging improves memory efficiency, since multi-word writes usually faster than one word at a time
Recap: Average Memory Access Time (AMAT)

**AMAT components**

Average memory access time = Hit time + Miss rate × Miss penalty

CPU time = (CPU execution clock cycles + Memory stall clock cycles) × Clock cycle time

\[ \text{CPU time} = IC \times \left( CPI_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time} \]

\[ \text{CPU time} = IC \times \left( CPI_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time} \]

Assuming that cache hits do not stall the machine!
Example

UltraSPARC III

- in-order processor
- $CPI_{execution} = 1.0$
- miss penalty = 100 cycles
- miss rate = 2%
- 1.5 memory references per instruction
- 30 cache misses per 1000 instructions

CPU time = $IC \times \left( 1.0 + 0.02 \times \frac{1.5}{1} \times 100 \right) \times \text{Clock cycle time} = IC \times 4 \times \text{cycle time}$

CPU time = $IC \times \left( 1.0 + \frac{30}{1000} \times 100 \right) \times \text{Clock cycle time} = IC \times 4 \times \text{cycle time}$
Example

UltraSPARC III

- Cache miss latency increases execution time by 4x
- Higher clock rates imply more clock cycles wasted due to miss penalty
  - Higher relative impact of cache on performance
- HW/SW cache-conscious optimizations attempt reduce AMAT
- Performance depends on both clock cycle and AMAT – trade-off
Example

Direct-mapped vs. set-associative cache

- 1 GHz processor
- $CPI_{\text{execution}} = 2.0$
- 64 KB caches with 64-byte blocks
- 1.5 memory references per instruction
- Direct mapped cache miss rate = 1.4%
- Set associative cache stretches clock cycle by 1.25, miss rate = 1.0%
- 75 ns miss penalty (i.e. 75 cc or 60 cc)
- 1 cycle hit time

\[ AMAT_{\text{direct-mapped}} = 1.0 + (.014 \times 75) = 2.05\, ns \]

\[ AMAT_{2-\text{way}} = 1.0 \times 1.25 + (.01 \times 75) = 2.00\, ns \]
Example

Direct-mapped vs. set-associative cache

\[
\text{CPU time} = IC \times \left( CPI_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{miss penalty} \right) \times \text{clock cycle time}
\]

\[
\text{CPU time}_{\text{direct-mapped}} = IC \times (2.0 \times 1.0 + 0.014 \times 1.5 \times 75) = 3.58 \times IC
\]

\[
\text{CPU time}_{\text{two-way}} = IC \times (2.0 \times 1.25 + 0.01 \times 1.5 \times 75) = 3.63 \times IC
\]

- Associative cache achieves lower AMAT than direct-mapped cache
- Direct-mapped cache achieves higher performance than associative cache

Why? In this example common case (hits) are faster for Direct-mapped cache.
Overlapping memory latency in OOO processors

**Miss penalty in OOO**

- Processor can execute instructions while cache miss is pending
- Processors can execute instructions also while cache hit is pending
- Hard to attribute stall cycles to instructions
  - Stall cycle is any cycle where at least one instruction does not commit
  - First

\[
\text{Memory stall cycles per instruction} = \frac{\text{Misses per instruction}}{\text{Misses per instruction}} \times (\text{Total miss latency} - \text{overlapped miss latency})
\]

CS425 - Vassilis Papaefstathiou