Memory Hierarchies & Cache Memory

CSE 410, Spring 2009
Computer Systems

http://www.cs.washington.edu/410
Reading and References

• Reading
  » Computer Organization and Design, Patterson and Hennessy
    • Section 5.1 Introduction
    • Section 5.2 The Basics of Caches
    • Section 5.3 Measuring and Improving Cache Performance
Large and Fast

• We’d like our computers to have memory systems that are:
  » Large – big enough to hold large data sets, media files (audio/image/video), databases, etc.
  » Fast – in particular, fast enough so the processor doesn’t have to wait around for data
    • (i.e., single-cycle access times)
  » Cheap – of course!!
Small or Slow

- Unfortunately there is a tradeoff between speed, cost, and capacity

<table>
<thead>
<tr>
<th>Storage</th>
<th>Speed</th>
<th>Cost</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static RAM</td>
<td>Fastest</td>
<td>Expensive</td>
<td>Smallest</td>
</tr>
<tr>
<td>Dynamic RAM</td>
<td>Slow</td>
<td>Cheap</td>
<td>Large</td>
</tr>
<tr>
<td>Hard disks</td>
<td>Slowest</td>
<td>Cheapest</td>
<td>Largest</td>
</tr>
</tbody>
</table>

- Fast memory is too expensive to buy a lot (for most people)
- But DRAM is too slow compared to the processor – can’t have every lw or sw access DRAM (long cycle times or lots of stalls)
- And there’s no way we can fetch instructions from disk!
Some rough estimates

- As of about 2008

<table>
<thead>
<tr>
<th>Storage</th>
<th>Delay</th>
<th>Cost/MB</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static RAM</td>
<td>1-10 cycles</td>
<td>~$2-$5</td>
<td>128K-2MB</td>
</tr>
<tr>
<td>Dynamic RAM</td>
<td>100-200 cycles</td>
<td>~$0.02-0.08</td>
<td>128MB-8GB</td>
</tr>
<tr>
<td>Hard Disks</td>
<td>10,000,000+ cycles</td>
<td>$0.0002-$0.002</td>
<td>20GB-1000GB</td>
</tr>
</tbody>
</table>

- Exact numbers have changed over time, but the ratios have stayed surprisingly similar

- Flash memory is appearing in the niche between DRAM and disks, but is not a major player in desktop systems – yet.
A Solution: Memory Hierarchy

- Keep copies of the active data in the small, fast, expensive storage
- Keep all data in the big, slow, cheap storage
- Move frequently used data to fast memory – automatically
What is a Cache?

• A cache allows for fast accesses to a subset of a larger data store

• Very general idea – Example: your web browser’s cache gives you fast access to pages you visited recently
  » faster because it’s stored locally
  » subset because the web won’t fit on your disk

• The memory cache gives the processor fast access to memory that it used recently
  » faster because it’s expensive; usually located on the CPU chip
  » subset because the cache is smaller than main memory
Memory Hierarchy

CPU

Registers

L1 cache

L2 Cache

Main Memory
Locality

• It’s usually difficult or impossible to figure out a program’s memory access patterns without running it

• If programs accessed memory randomly it would be hard to automatically identify what data and instructions are “hot” and move them to faster cache memory

• Fortunately most programs exhibit locality, which the cache can take advantage of
Principle of Locality

- Temporal locality - nearness in time
  - Data being accessed now will probably be accessed again soon
  - Useful data tends to continue to be useful
- Spatial locality - nearness in address
  - Data near the data being accessed now will probably be needed soon
  - Useful data is often accessed sequentially
- Applies to both instructions and data
- Remember – this is observed behavior that we can take advantage of, not something that is necessarily consciously designed (although it can be)
Memory Access Patterns

- Memory accesses **don’t** usually look like this
  - random accesses

- Memory accesses **do** usually look like this
  - hot variables
  - step through arrays
Temporal locality in programs

• The principle of **temporal locality** says that if a program accesses one memory address, there is a good chance that it will access the same address again.

• Loops are excellent examples of temporal locality in programs.
  » The loop body will be executed many times.
  » The computer will need to access those same few locations of the instruction memory repeatedly.

• For example:

```
Loop:   lw   $t0, 0($s1)
       add  $t0, $t0, $s2
       sw   $t0, 0($s1)
       addi  $s1, $s1, -4
       bne  $s1, $0, Loop
```

» Each instruction will be fetched over and over again, once on every loop iteration.
Temporal locality in data

• Programs often access the same variables over and over, especially within loops. Below, \texttt{sum} and \texttt{i} are repeatedly read and written.

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + f(i);
```

• Commonly-accessed variables can sometimes be kept in registers, but this is not always possible.
  » There are a limited number of registers.
  » There are situations where the data must be kept in memory, as is the case with shared or dynamically-allocated memory.
Spatial locality in programs

• The principle of \textit{spatial locality} says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

\begin{verbatim}
sub $sp, $sp, 16
sw $ra, 0($sp)
sw $s0, 4($sp)
sw $a0, 8($sp)
sw $a1, 12($sp)
\end{verbatim}

• Nearly every program exhibits spatial locality, because instructions are usually executed in sequence—if we execute an instruction at memory location $i$, then we will probably also execute the next instruction, at memory location $i+1$.

• Code fragments such as loops exhibit \textit{both} temporal and spatial locality.
Spatial locality in data

• Programs often access data that is stored contiguously.
  » Arrays, like `a` in the code on the top, are stored in memory contiguously.
  » The individual fields of a record or object like `employee` are also kept contiguously in memory.

• Can data have both spatial and temporal locality?

```python
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + a[i];

employee.name = "Homer Simpson";
employee.boss = "Mr. Burns";
employee.age = 45;
```
How caches take advantage of temporal locality

• The first time the processor reads from an address in main memory, a copy of that data is also stored in the cache.
  » The next time that same address is read, we can use the copy of the data in the cache instead of accessing the slower dynamic memory.
  » So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster.

• This takes advantage of temporal locality—commonly accessed data is stored in the faster cache memory.
How caches take advantage of spatial locality

- When the CPU reads location $i$ from main memory, a copy of that data is placed in the cache.
- But instead of just copying the contents of location $i$, we can copy several values into the cache at once, such as the four bytes from locations $i$ through $i + 3$.
  » If the CPU later does need to read from locations $i + 1$, $i + 2$ or $i + 3$, it can access that data from the cache and not the slower main memory.
  » For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.
- Again, the initial load incurs a performance penalty, but we’re gambling on spatial locality and the chance that the CPU will need the extra data.
Definitions: Hits and misses

• A cache hit occurs if the cache contains the data that we’re looking for. Hits are good, because the cache can return the data much faster than main memory.
• A cache miss occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.
• There are two basic measurements of cache performance.
  » The hit rate is the percentage of memory accesses that are handled by the cache.
  » The miss rate (1 - hit rate) is the percentage of accesses that must be handled by the slower main RAM.
• Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster.
Effective Access Time

\[ t_{\text{effective}} = (h) t_{\text{cache}} + (1-h) t_{\text{memory}} \]

aka, Average Memory Access Time (AMAT)
Cache Contents

• When do we put something in the cache?
  » when it is used for the first time
    • or when we access something nearby

• When do we overwrite something in the cache?
  » when we need the space in the cache for some other entry
  » all of memory won’t fit on the CPU chip so not every location in memory can be cached
A small two-level hierarchy

8-word cache

32-word memory  (128 bytes)
Fully Associative Cache

- In a fully associative cache,
  - any memory word can be placed in any cache line
  - each cache line stores an address and a data value
  - accesses are slow (but not as slow as you might think)

<table>
<thead>
<tr>
<th>Address</th>
<th>Valid</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010100</td>
<td>Y</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0000100</td>
<td>N</td>
<td>0x09D91D11</td>
</tr>
<tr>
<td>0100100</td>
<td>Y</td>
<td>0x00000410</td>
</tr>
<tr>
<td>0101100</td>
<td>Y</td>
<td>0x00012D10</td>
</tr>
<tr>
<td>0001100</td>
<td>N</td>
<td>0x00000005</td>
</tr>
<tr>
<td>1101100</td>
<td>Y</td>
<td>0x0349A291</td>
</tr>
<tr>
<td>0100000</td>
<td>Y</td>
<td>0x000123A8</td>
</tr>
<tr>
<td>1111100</td>
<td>N</td>
<td>0x00000200</td>
</tr>
</tbody>
</table>
Direct Mapped Caches

• Fully associative caches are often too slow
• With direct mapped caches the address of the item determines where in the cache to store it
  » In our example, the lowest order two bits are the byte offset within the word stored in the cache
  » The next three bits of the address are an index that dictates the location of the entry within the cache
  » The remaining higher order bits record the rest of the original address as a tag for this entry
    • The tag bits indicate the actual memory location of the word in a particular cache line
Address Tags

- A *tag* is a label for a cache entry indicating where it came from
  - The upper bits of the data item’s address
- The *index* indicates where the entry goes in the cache

<table>
<thead>
<tr>
<th>7 bit Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag (2)</th>
<th>Index (3)</th>
<th>Byte Offset (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>111</td>
<td>01</td>
</tr>
</tbody>
</table>
## Direct Mapped Cache

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache Contents</th>
<th>Cache Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000000</td>
<td>11 Y 0x00000001 000₂ = 0</td>
<td></td>
</tr>
<tr>
<td>10001000</td>
<td>10 N 0x09D91D11 001₂ = 1</td>
<td></td>
</tr>
<tr>
<td>01010000</td>
<td>01 Y 0x00000410 010₂ = 2</td>
<td></td>
</tr>
<tr>
<td>00011000</td>
<td>00 Y 0x00012D10 011₂ = 3</td>
<td></td>
</tr>
<tr>
<td>10100000</td>
<td>10 N 0x00000005 100₂ = 4</td>
<td></td>
</tr>
<tr>
<td>11101000</td>
<td>11 Y 0x0349A291 101₂ = 5</td>
<td></td>
</tr>
<tr>
<td>00110000</td>
<td>00 Y 0x000123A8 110₂ = 6</td>
<td></td>
</tr>
<tr>
<td>10111000</td>
<td>10 N 0x00000200 111₂ = 7</td>
<td></td>
</tr>
</tbody>
</table>
N-way Set Associative Caches

• Direct mapped caches cannot store more than one address with the same index
• If two addresses collide, then you overwrite the older entry
• 2-way associative caches can store two different addresses with the same index
  » 3-way, 4-way and 8-way set associative designs too
• Reduces misses due to conflicts
• Larger sets imply slower accesses
# 2-way Set Associative Cache

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>Value</th>
<th>Tag</th>
<th>Valid</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>11</td>
<td>Y</td>
<td>0x000000001</td>
<td>00</td>
<td>Y</td>
<td>0x00000002</td>
</tr>
<tr>
<td>001</td>
<td>10</td>
<td>N</td>
<td>0x09D91D11</td>
<td>10</td>
<td>N</td>
<td>0x0000003B</td>
</tr>
<tr>
<td>010</td>
<td>01</td>
<td>Y</td>
<td>0x00000410</td>
<td>11</td>
<td>Y</td>
<td>0x000000CF</td>
</tr>
<tr>
<td>011</td>
<td>00</td>
<td>Y</td>
<td>0x00012D10</td>
<td>10</td>
<td>N</td>
<td>0x000000A2</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>N</td>
<td>0x00000005</td>
<td>11</td>
<td>N</td>
<td>0x00000333</td>
</tr>
<tr>
<td>101</td>
<td>11</td>
<td>Y</td>
<td>0x0349A21</td>
<td>10</td>
<td>Y</td>
<td>0x00003333</td>
</tr>
<tr>
<td>110</td>
<td>00</td>
<td>Y</td>
<td>0x000123A8</td>
<td>01</td>
<td>Y</td>
<td>0x0000C002</td>
</tr>
<tr>
<td>111</td>
<td>10</td>
<td>N</td>
<td>0x00000200</td>
<td>10</td>
<td>N</td>
<td>0x00000005</td>
</tr>
</tbody>
</table>

The highlighted cache entry contains values for addresses 10101xx₂ and 11101xx₂.
Associativity Spectrum

Direct Mapped
Fast to access
Conflict Misses

N-way Associative
Slower to access
Fewer Conflict Misses

Fully Associative
Slow to access
No Conflict Misses
Spatial Locality

• Using the cache improves performance by taking advantage of temporal locality
  » When a word in memory is accessed it is loaded into cache memory
  » It is then available quickly if it is needed again soon

• This does nothing for spatial locality
Memory Blocks

- Divide memory into blocks
- If any word in a block is accessed, then load an entire block into the cache
  » Usually called a cache line

<table>
<thead>
<tr>
<th>Block 0</th>
<th>0x00000000–0x0000003F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>0x00000040–0x0000007F</td>
</tr>
<tr>
<td>Block 2</td>
<td>0x00000080–0x000000BF</td>
</tr>
</tbody>
</table>

Cache line for 16 word block size

<table>
<thead>
<tr>
<th>tag</th>
<th>valid</th>
<th>w0</th>
<th>w1</th>
<th>w2</th>
<th>w3</th>
<th>w4</th>
<th>w5</th>
<th>w6</th>
<th>w7</th>
<th>w8</th>
<th>w9</th>
<th>w10</th>
<th>w11</th>
<th>w12</th>
<th>w13</th>
<th>w14</th>
<th>w15</th>
</tr>
</thead>
</table>

4/26/2009

cse410-13-cache ©2006-09 Perkins, DW Johnson and University of Washington
Address Tags Revisited

- A cache block size > 1 word requires the address to be divided differently
- Instead of a byte offset into a word, we need a byte offset into the block
- Assume we have 10-bit addresses, 8 cache lines, and 4 words (16 bytes) per cache line block…

<table>
<thead>
<tr>
<th>10 bit Address</th>
<th>Tag (3)</th>
<th>Index (3)</th>
<th>Block Offset (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101100111</td>
<td>010</td>
<td>110</td>
<td>0111</td>
</tr>
</tbody>
</table>
The Effects of Block Size

• Big blocks are good
  » Fewer first time misses
  » Exploits spatial locality

• Small blocks are good
  » Don’t evict as much data when bringing in a new entry
  » More likely that all items in the block will turn out to be useful
Reads vs. Writes

- Caching is essentially making a copy of the data
- When you read, the copies still match the original after you’ve read the data
- When you write, the results must eventually propagate to both copies
  - Especially at the lowest (slowest, largest) level of the hierarchy, which is in some sense the permanent copy or the “truth”
Write-Through Caches

• Write all updates to both cache and memory

• Advantages
  » The cache and the memory are always consistent
  » Evicting a cache line is cheap because no data needs to be written out to memory at eviction
  » Easy to implement

• Disadvantages
  » Runs at memory speeds when writing (can use write buffer to reduce this problem)
Write-Back Caches

- Write the update to the cache only. Write to memory only when cache block is evicted
- Advantage
  - Runs at cache speed rather than memory speed
  - Some writes never go all the way to memory
  - When a whole block is written back, can use high bandwidth transfer
- Disadvantage
  - Complexity required to maintain consistency – what if another processor tries to read the same memory location?
Dirty bit

• When evicting a block from a write-back cache, we could
  » always write the block back to memory
  » write it back only if we changed it
• Caches use a “dirty bit” to mark if a line was changed
  » the dirty bit is 0 when the block is loaded
  » it is set to 1 if the block is modified
  » when the line is evicted, it is written back only if the dirty bit is 1
i-Cache and d-Cache

• There usually are two separate caches for instructions and data.
  » Avoids structural hazards in pipelining
  » The combined cache is twice as big but still has an access time of a small cache
  » Allows both caches to operate in parallel, for twice the bandwidth
Cache Line Replacement

• How do you decide which cache block to replace?

• If the cache is direct-mapped, it’s easy
  » only one slot per index

• Otherwise, common strategies:
  » Random
  » Least Recently Used (LRU)
LRU Implementations

• LRU is very difficult to implement for high degrees of associativity

• 4-way approximation:
  » 1 bit to indicate least recently used pair
  » 1 bit per pair to indicate least recently used item in this pair

• We will see this issue again at the operating system level for virtual memory pages
Multi-Level Caches

- Use each level of the memory hierarchy as a cache over the next lowest level
- Inserting level 2 between levels 1 and 3 allows:
  - level 1 to have a higher miss rate (so can be smaller and cheaper)
  - level 3 to have a larger access time (so can be slower and cheaper)
Summary: Classifying Caches

• Where can a block be placed?
  » Direct mapped, N-way Set or Fully associative

• How is a block found?
  » Direct mapped: by index
  » Set associative: by index and search
  » Fully associative: by search

• What happens on a write access?
  » Write-back or Write-through

• Which block should be replaced?
  » Random
  » LRU (Least Recently Used)
Not Explored (Yet?)

- Cache Coherency in multiprocessor systems
- Want each processor to have its own cache
  - Fast local access
  - No interference with/from other processors
- But: now what happens if more than one processor accesses a cache line at the same time?
  - How do we keep multiple copies consistent?
  - What about synchronization with main storage?