Memory hierarchy review

ECE 154B
Dmitri Strukov
Outline

• Cache motivation
• Cache basics
• Six basic optimizations
• Virtual memory
• Cache performance
• Opteron example
Q1. How to deal with memory wall problem?
Q1. How to deal with memory wall problem?

A1. Memory hierarchy \(\rightarrow\) will be discussing next
- Out-of-order execution
- Multithreading
- Vector processors

Discussed during ILP, DLP and TLP
Memory Hierarchy Motivation

- Problem: large memory access time >> clock cycle time
- Ideally, want to have fast and big memory simultaneously, cannot have both $\rightarrow$ build memory hierarchy
Memory Hierarchy Levels

**CPU Registers**
- 100s Bytes
- 300 - 500 ps (0.3-0.5 ns)

**L1 and L2 Cache**
- 10s-100s K Bytes
- ~1 ns - ~10 ns
- ~ $100s/ GByte

**Main Memory**
- G Bytes
- 80ns- 200ns
- ~ $10/ GByte

**Disk**
- 10s T Bytes, 10 ms
- (10,000,000 ns)
- ~ $0.1 / GByte

**Tape**
- infinite sec-min
- ~$0.1 / GByte

**Q2: why does hierarchy help with memory wall problem?**

Note the price/bit difference!
Memory Hierarchy Motivation

- A2: Fortunately, there is plenty of temporal and spatial locality in data so that one can take advantage of memory hierarchy and so create illusion of fast and large memory
Real Memory Reference Patterns

Types of Memory and Storage in a Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Registers</td>
<td>Cache</td>
<td>Main memory</td>
<td>Disk storage</td>
</tr>
<tr>
<td>Typical size</td>
<td>&lt;1 KB</td>
<td>32 KB–8 MB</td>
<td>&lt;512 GB</td>
<td>&gt;1 TB</td>
</tr>
<tr>
<td>Implementation technology</td>
<td>Custom memory with multiple ports, CMOS</td>
<td>On-chip CMOS SRAM</td>
<td>CMOS DRAM</td>
<td>Magnetic disk</td>
</tr>
<tr>
<td>Access time (ns)</td>
<td>0.15–0.30</td>
<td>0.5–15</td>
<td>30–200</td>
<td>5,000,000</td>
</tr>
<tr>
<td>Bandwidth (MB/sec)</td>
<td>100,000–1,000,000</td>
<td>10,000–40,000</td>
<td>5000–20,000</td>
<td>50–500</td>
</tr>
<tr>
<td>Managed by</td>
<td>Compiler</td>
<td>Hardware</td>
<td>Operating system</td>
<td>Operating system/operator</td>
</tr>
<tr>
<td>Backed by</td>
<td>Cache</td>
<td>Main memory</td>
<td>Disk</td>
<td>Other disks and DVD</td>
</tr>
</tbody>
</table>

Figure B.1 The typical levels in the hierarchy slow down and get larger as we move away from the processor for a large workstation or small server. Embedded computers might have no disk storage and much smaller memories and caches. The access times increase as we move to lower levels of the hierarchy, which makes it feasible to manage the transfer less responsively. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 2006; these times will decrease over time. Bandwidth is given in megabytes per second between levels in the memory hierarchy. Bandwidth for disk storage includes both the media and the buffered interfaces.

Note that bandwidth (bw) number!
Q3: Why does the bw decrease as we go down in hierarchy? A related question - what determines proper bw?
The simplest case: Direct-mapped cache

Cache is a small but fast memory which is sitting on a datapath containing the data which might (inclusive) or may not (exclusive) be duplicated elsewhere.

Block index = (Block address) MOD (Number of blocks in cache)
Main questions to answer when considering cache

- Where can a block be placed in the upper level? (block placement)
- How is a block found if it is in the upper level? (block identification)
- Which block should be replaced on a miss? (block replacement)
- What happens on a write? (write strategy)
Block Placement

Block Number

Memory

Set Number

Cache

Fully Associative

(2-way) Set Associative

Direct Mapped

block 12 can be placed anywhere

anywhere in set 0 (12 mod 4)

only into block 4 (12 mod 8)
2-Way Set-Associative Cache

Set index = (Block address) MOD (Number of sets in cache)
Fully Associative Cache
Replacement Policy

• In associative cache, which block from a set should be evicted when the set becomes full?
  – Random
  – Least recently used (LRU)
    • LRU cache state must be updated on every access
    • True implementation only feasible for small sets (2-ways)
    • Psedo-LRU often used for 4-8 way
  – First In, First OUT (FIFO) a.k.a Round-Robin
    • Used in highly associative caches
Write Policy

• Cache hit:
  – Write through: write both cache & memory
  – Write-back: write cache only
  
  Q4: Write through vs write-back – cons and pros?
  Q5: Is it always necessary to write back to main memory for write back policy on a miss?

• Cache miss:
  – No write allocate: only write to main memory
  – Write allocated (aka fetch on write): fetch into cache

• Write buffer helps with write stalls
  – no need to wait till data are updated in memory.
  – Start as soon as data at in buffer
Cache Performance

\[
\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} \quad = \quad \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}
\]

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

• Still may not be the best metric of the performance
  • Q6: Why and which metric is better?
    • speculative and multithreaded processors may execute other instructions during a miss which reduces performance impact of misses
Six Basic Cache Optimizations

• Reduce the miss rate – 1) larger block size, 2) larger cache size, 3) higher associativity
• Reduce the miss penalty – 4) multilevel caches and 5) giving reads priority over writes
• Reducing the time to hit in the cache – 6) avoiding address translation when indexing the cache
Three Cs Model

• **Compulsory** cold start or process migration, first reference):
  – First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant
  – Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)

• **Capacity**:
  – Cache cannot contain all blocks accessed by the program
  – Solution: increase cache size (may increase access time)

• **Conflict** (collision):
  – Multiple memory locations mapped to the same cache location
  – Solution 1: increase cache size
  – Solution 2: increase associativity (stay tuned) (may increase access time)
FIGURE 5.31 The miss rate can be broken into three sources of misses. This graph shows the total miss rate and its components for a range of cache sizes. This data is for the SPEC2000 integer and floating-point benchmarks and is from the same source as the data in Figure 5.30. The compulsory miss component is 0.006% and cannot be seen in this graph. The next component is the capacity miss rate, which depends on cache size. The conflict portion, which depends both on associativity and on cache size, is shown for a range of associativities from one-way to eight-way. In each case, the labeled section corresponds to the increase in the miss rate that occurs when the associativity is changed from the next higher degree to the labeled degree of associativity. For example, the section labeled two-way indicates the additional misses arising when the cache has associativity of two rather than four. Thus, the difference in the miss rate incurred by a direct-mapped cache versus a fully associative cache of the same size is given by the sum of the sections marked eight-way, four-way, two-way, and one-way. The difference between eight-way and four-way is so small that it is difficult to see on this graph. Copyright © 2009 Elsevier, Inc. All rights reserved.
Optimization #1: Large Block Size to Reduce Miss Rate

Q: What effect does it have on miss rate and other metrics?
Optimization #1:
Large Block Size to Reduce Miss Rate

+ Reduce compulsory misses
  (by better utilizing spatial locality)
- Increase miss penalty
- Could increase conflict misses

(no benefit if miss rate decrease is offset by increase of miss penalty)
Optimization #1:

Figure B.11 Actual miss rate versus block size for the five different-sized caches in Figure B.10. Note that for a 4 KB cache, 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KB in order for a 256-byte block to decrease misses.

Figure B.12 Average memory access time versus block size for five different-sized caches in Figure B.10. Block sizes of 32 and 64 bytes dominate. The smallest average time per cache size is boldfaced.
Optimization #1

• Optimal block size depends on the latency and bandwidth of lower-level memory
  – Good balance between the bandwidth and block size
Optimization #2: Increase Cache Size

Q: What effect does it have on miss rate and other metrics?
Optimization #2: Larger Caches to Reduce Miss Rate

+ Large cache reduce capacity misses
- Longer hit time
- Higher power and cost
Optimization #2:
Larger Caches to Reduce Miss Rate

- Longer hit time
- Higher power and cost

Access time estimate for 90 nm using CACTI model 4.0
Optimization #3: Higher Associativity

Q: What effect does it have on miss rate and other metrics?
Optimization #3: Higher Associativity to Reduce Miss Rate

+ Reduce conflict misses
- Increase hit time
- Increase power

2:1 rule of thumb: direct mapped cache of size N has the same miss rate as two-way associative cache of size N/2
Hit Time Increase

Directed mapped vs. set-associative cache

Access time estimate for 90 nm using CACTI model 4.0
Optimization #3:

Clock cycle time_{2-way} = 1.36 \times \text{Clock cycle time}_{1-way}
Clock cycle time_{4-way} = 1.44 \times \text{Clock cycle time}_{1-way}
Clock cycle time_{8-way} = 1.52 \times \text{Clock cycle time}_{1-way}

<table>
<thead>
<tr>
<th>Cache size (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.44</td>
<td>3.25</td>
<td>3.22</td>
<td>3.28</td>
</tr>
<tr>
<td>8</td>
<td>2.69</td>
<td>2.58</td>
<td>2.55</td>
<td>2.62</td>
</tr>
<tr>
<td>16</td>
<td>2.23</td>
<td>2.40</td>
<td>2.46</td>
<td>2.53</td>
</tr>
<tr>
<td>32</td>
<td>2.06</td>
<td>2.30</td>
<td>2.37</td>
<td>2.45</td>
</tr>
<tr>
<td>64</td>
<td>1.92</td>
<td>2.14</td>
<td>2.18</td>
<td>2.25</td>
</tr>
<tr>
<td>128</td>
<td>1.52</td>
<td>1.84</td>
<td>1.92</td>
<td>2.00</td>
</tr>
<tr>
<td>256</td>
<td>1.32</td>
<td>1.66</td>
<td>1.74</td>
<td>1.82</td>
</tr>
<tr>
<td>512</td>
<td>1.20</td>
<td>1.55</td>
<td>1.59</td>
<td>1.66</td>
</tr>
</tbody>
</table>

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. Boldface type means that this time is higher than the number to the left, that is, higher associativity increases average memory access time.

Miss rate for fully set associative cache with LRU policy is always better compared to direct mapped cache – true or false?
Optimization #4: Multilevel Caches to Reduce Miss Penalty

• Fast 1\textsuperscript{st} level cache (matching the clock cycle time of fast processor)
• Large 2\textsuperscript{nd} level cache (to catch misses in the 1\textsuperscript{st} level cache)

Q: What effect does it have on miss rate and other metrics?
Optimization #4: Multilevel Caches to Reduce Miss Penalty

- Fast 1\textsuperscript{st} level cache (matching the clock cycle time of fast processor)
- Large 2\textsuperscript{nd} level cache (to catch misses in the 1\textsuperscript{st} level cache)

+ Reduce miss penalty

- High complexity

\[
AMAT = \text{Hit Time L1} + \text{Miss Rate L1} \times \text{Miss Penalty L1}
\]
\[
\text{Miss Penalty L1} = \text{Hit Time L2} + \text{Miss Rate L2} \times \text{Miss Penalty L2}
\]

Local miss rate: \# misses / \# local number of accesses
Global miss rate: \# misses / \# total number of access

( i.e. product of local miss rates to itself and miss rates to upper levels of cache)
Optimization #4:

Figure B.15 Relative execution time by second-level cache size. The two bars are for different clock cycles for an L2 cache hit. The reference execution time of 1.00 is for an 8192 KB second-level cache with a 1-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure B.14, using a simulator to imitate the Alpha 21264.
Optimization #4:

Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches smaller than the sum of the two 64 KB first-level caches make little sense, as reflected in the high miss rates. After 256 KB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32 KB first-level cache. The L2 caches (unified) were two-way set associative with replacement. Each had split L1 instruction and data caches that were 64 KB two-way set associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data were collected as in Figure B.4.
Optimization #5: Giving Priority to Read Misses over Writes to Reduce Miss Penalty

- Check the write buffer on a read miss

```
SW R3, 512(R0)   ; M[512] ← R3  (cache index 0)
LW R1, 1024(R0)  ; R1 ← M[1024]  (cache index 0)
LW R2, 512(R0)   ; R2 ← M[512]   (cache index 0)
```
Optimization #6: Avoiding Address Translation during Indexing of the Cache to Reduce Hit Time
Logic behind virtual memory mechanism - I

• Need mechanism to cache storage in main memory (MM), i.e. similar to caching MM in cache ($)

• Can we have the same HW mechanism as for $\leftrightarrow$ MM
  – Prefer to use off-the-shelf-components for MM (so answer is no)

• If we have to build it in SW how can what and how it can be done?
Step 1: On every request to MM (i.e. miss to $) generate exception (O/S code)
   → some minimal HW support is also always necessary (not pure SW mechanism)
Step 2: Implement $ in software in memory in that O/S code
   → if implemented exactly as hw cache (case A) comes with many problems, e.g. reading value of the tag could also result in miss
   → additionally, miss could be instruction miss what happens on instruction miss for exception procedure
Step 3: Partition MM into O/S and user space
   → store only tags in a table and have very large pages – see case B
   → keep O/S code and all tables in O/S portion
      → whenever miss is to O/S portion do not generate exception but simply read from MM, i.e. assume that these data are certainly in MM and not in storage (for that all O/S procedures should be preloaded)
Step 4: Code relocation is awkward which may lead to inefficient memory usage
→ perform address translation (case C), i.e. work with virtual memory
→ natural for implementation of sharing and protection mechanisms

Step 5: Programs (data and code) are dealing with virtual address which must be translated to physical addresses
→ To avoid vicious circles note that access of O/S does not need translation
Virtual Memory

• Motivation:
  – Caching
  – Relocation
  – Sharing
  – Protection
Address Translation Mechanisms

Virtual page # | Offset
--- | ---
| | |

Physical page #

Page table register

Page Table
(in main memory)

Main memory

Disk storage
Step 4: Code relocation is awkward which may lead to inefficient memory usage
→ perform address translation (case C), i.e. work with virtual memory
→ natural for implementation of sharing and protection mechanisms

Step 5: Programs (data and code) are dealing with virtual address which must be translated to physical addresses
→ To avoid vicious circles note that access of O/S does not need translation

→ Should $ be for virtual addresses or physical addresses?
Aliasing in Virtual-Address Caches

Two virtual pages share one physical page.

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist in cache*

Software (i.e., OS) solution for direct-mapped cache:
VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARC)
Virtual Addressing with a Cache

• Thus it takes an *extra* memory access to translate a VA to a PA

![Diagram of virtual addressing with a cache]

- This makes memory (cache) accesses very expensive (if every access was really *two* accesses)
- The hardware fix is to use a Translation Lookaside Buffer (TLB) – a small cache that keeps track of recently used address mappings to avoid having to do a page table lookup
# Making Address Translation Fast

![Diagram]

<table>
<thead>
<tr>
<th>Virtual page #</th>
<th>Physical page base addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- **TLB**
- **Page Table** (in physical memory)
- **Main memory**
- **Disk storage**

- **Virtual page #**
- **Physical page base addr**
- **Page table register**
A TLB miss – is it a page fault or merely a TLB miss?

- If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
  - Takes 10’s of cycles to find and load the translation info into the TLB
- If the page is not in main memory, then it’s a true page fault
  - Takes 1,000,000’s of cycles to service a page fault

• TLB misses are much more frequent than true page faults
Virtual Address Caches

Alternative: place the cache before the TLB

- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- aliasing problems due to the sharing of pages (-)
Opt #6: Concurrent Access to TLB & Cache: Virtually Indexed Physically Tagged

Index L is available without consulting the TLB
⇒ cache and TLB accesses can begin simultaneously
Tag comparison is made after both accesses are completed

Cases: \( L + b = k \) \( L + b < k \) \( L + b > k \)
Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage
- OS sets the Page Table Base Register whenever active user process changes
Size of Linear Page Table

With 32-bit addresses, 4-KB pages & 4-byte PTEs:

⇒ $2^{20}$ PTEs, i.e., 4 MB page table per user
⇒ 4 GB of swap needed to back up full virtual address space

Larger pages?

• Internal fragmentation (Not all memory in a page is used)
• Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

• Even 1MB pages would require $2^{44}$ 8-byte PTEs (35 TB!)

*What is the “saving grace” ?*
Hierarchical Page Table

Virtual Address
- 31 22 21 12 11 0
- \( p_1 \) \( p_2 \) offset

10-bit 10-bit
L1 index L2 index

Root of the Current Page Table
- (Processor Register)

Level 1 Page Table
- \( p_1 \)

Level 2 Page Tables

Data Pages

- page in primary memory
- page in secondary memory
- PTE of a nonexistant page
Virtual Memory Use Today

- Desktops/servers have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features

- Most embedded processors and DSPs provide physical addressing only
  - Can’t afford area/speed/power budget for virtual memory support
  - Often there is no secondary storage to swap to!
  - Programs custom written for particular memory configuration in product
  - Difficult to implement restartable instructions for exposed architectures
Acknowledgements

Some of the slides contain material developed and copyrighted by Arvind, Emer (MIT), Asanovic (UCB/MIT) and instructor material for the textbook
Extra slides not covered in lecture
Anti-Aliasing Hardware Solution

AMD Opteron checks several block tags for possible aliasing
Hashed Page Table

- Hashed Page Table is typically 2 to 3 times larger than the number of PPN’s to reduce collision probability.
- It can also contain DPN’s for some non-resident pages (not common).
- If a translation cannot be resolved in this table then the software consults a data structure that has an entry for every existing page.