CSE 502 Graduate Computer Architecture

Lec 5-6 – Memory Hierarchy Review

Larry Wittie
Computer Science, StonyBrook University

http://www.cs.sunysb.edu/~cse502 and ~lw

Slides adapted from David Patterson, UC-Berkeley cs252-s06
Review from last lecture

• Quantify and summarize performance
  – Ratios, Geometric Mean, Multiplicative Standard Deviation
• F&P: Benchmarks age, disks fail, 1 point fail danger
• Control VIA State Machines and Microprogramming
• Just overlap tasks; easy if tasks are independent
• Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

• Hazards limit performance on computers:
  – Structural: need more HW resources
  – Data (RAW, WAR, WAW): need forwarding, compiler scheduling
  – Control: delayed branch, prediction
• Exceptions, Interrupts add complexity
Outline

- Review
- Memory hierarchy
- Locality
- Cache design
- Virtual address spaces
- Page table layout
- TLB design options
- Conclusion
Memory Hierarchy Review
Since 1980, CPU has outpaced DRAM ...

Q. How do architects address this gap?
   A. Put smaller, faster “cache” memories between CPU and DRAM. Create a “memory hierarchy”.

Performance (1/latency)

Year

Latency gap grew 50% per year

CPU 60% per yr 2X in 1.5 yrs

DRAM 9% per yr 2X in 10 yrs
1977: DRAM faster than microprocessors

Apple II (1977)
Latencies
CPU clock: 1000 ns
DRAM access: 400 ns

<table>
<thead>
<tr>
<th>RAM Complement</th>
<th>Apple II System</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>$1,298.00</td>
</tr>
<tr>
<td>48K</td>
<td>$2,638.00</td>
</tr>
</tbody>
</table>

Steve Jobs
Steve Wozniak
Levels of the Memory Hierarchy

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU Registers</strong></td>
<td>100s Bytes</td>
<td>&lt;10s ns</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>K Bytes</td>
<td>10-100 ns</td>
</tr>
<tr>
<td><strong>Main Memory</strong></td>
<td>M Bytes</td>
<td>200ns-500ns</td>
</tr>
<tr>
<td><strong>Disk</strong></td>
<td>G Bytes, 10 ms (10,000,000 ns)</td>
<td>10^{-5} - 10^{-6} cents/bit</td>
</tr>
<tr>
<td><strong>Tape</strong></td>
<td>Almost infinite sec-min</td>
<td>10^{-8} cents/bit</td>
</tr>
</tbody>
</table>

**Upper Level**
- Registers
- Instr. Operands
- Cache
- Blocks
- Memory
- Pages
- Disk
- Files
- Tap

**Lower Level**
- Staging
- Xfer Unit
- faster
- Larger
- prog./compiler 1-8 bytes
- cache cntl 8-128 bytes
- OS 512-4K bytes
- user/operator Mbytes
### Memory Hierarchy: Apple iMac G5 (2004-5)

**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.

<table>
<thead>
<tr>
<th>1977+27yr</th>
<th>Reg</th>
<th>L1 Inst</th>
<th>L1 Data</th>
<th>L2</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size in Bytes</td>
<td>1K</td>
<td>64K</td>
<td>32K</td>
<td>512K</td>
<td>256M</td>
<td>80G</td>
</tr>
<tr>
<td>Latency in Cycles, Time</td>
<td>1 cyc, 0.6 ns</td>
<td>3 cyc, 1.9 ns</td>
<td>3 cyc, 1.9 ns</td>
<td>11 cyc, 6.9 ns</td>
<td>88 cyc, 55 ns</td>
<td>(10^7) cyc, 12 ms</td>
</tr>
</tbody>
</table>

**iMac G5 1.6 GHz**
1600 (mem: 7.3) x Apple II

iMac G5 1.6 GHz clock, 55 ns DRAM vs Apple II 1 MHz, 400 ns DRAM

Perform: CPU 1600 X, DRAM 7.3 X faster in 27 yrs => 2X/ 2.5y, 9.3y

---

CSE502-S11, Lec 05+6-cache VM TLB

2/15-17/2011
iMac’s PowerPC 970 (G5): All caches on-chip

L1 (64K Instruction) ↓ ↓ ↓ ↓ ↓

1/2 KB Registers
1/2 KB

L1 (32K Data) ↑ ↑ ↑ ↑

512K L2

Original = 64-bit single-core
The Principle of Locality

• The Principle of Locality:
  – Program access a relatively small portion of the address space at any instant of time.

• Two Different Types of Locality:
  – Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  – Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)

• For last 20 years, HW has relied on locality for speed by using fast cache copies of memory parts to lower average memory access time (AMAT) for programs

  Locality is a property of programs which is exploited in machine design.
Programs with locality cache well ...

Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper level (example: Block X)
  - **Hit Rate**: the fraction of memory accesses found in the upper level
  - **Hit Time**: Time to access the upper level which consists of
    RAM access time + Time to determine hit/miss

- **Miss**: data needs to be retrieved from a block in the lower level (Block Y)
  - **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!)
CSE502: Administrivia

• Instructor: Prof Larry Wittie
• Office/Lab: 1308 CompSci, lw AT icDOTsunysbDOTedu
• Office Hrs: TuTh, 3:45 - 5:15 pm, if door open, or appt.
• T. A.: None
• Class: TuTh, 2:20 - 3:40 pm 2120 Comp Sci
• Text: *Computer Architecture: A Quantitative Approach, 4th Ed.* (Oct, 2006), ISBN 0123704901 or 978-0123704900, $65/($50?) Amazon; $77used SBU, S11

• Current reading: Appendix C (back of CAQA4) (Chap 1 was first week, App A was last week)
• Tue 2/22 or Th 2/24: After review, Quiz (Appendices A & C) Reading: Memory Hierarchy, Appendix C this week Reading assignment: Chapter 2 for Thur 2/24
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 average memory access time in memory

- **Average memory-access time**
  \[ \text{= Hit time} + \text{Miss rate} \times \text{Miss penalty} \]
  (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”}
  - **replacement time**: time to make upper-level room for new block
  - **access time**: time to lower level
    \[ = f(\text{latency to lower level}) \]
  - **transfer time**: time to transfer block
    \[ = f(\text{BW between upper \& lower levels}) \]
Four Questions for Memory Hierarchy
(relative to both caches and DRAM for virtual memory)

• 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 can a block be placed in the upper level?

- Memory block 12 placed in each 8-block cache:
  - 3 cache organizations are shown below:
  - Fully associative, direct mapped, 2-way set associative (S.A.)
  - S.A. Mapping = Block Number Modulo (Number of Sets)
  - (Cache blocks allowed to hold block 12, shown in blue.)

<table>
<thead>
<tr>
<th>Cache</th>
<th>Full Associative</th>
<th>Direct Mapped</th>
<th>2-Way Set Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01234567</td>
<td>01234567</td>
<td>01234567</td>
</tr>
</tbody>
</table>

Memory:

```
01234567890123456789012345678901
```

```
1111111111122222222223
```

```
01234567890123456789012345678901
```
Q2: How tell if block is in upper level cache?
Bits = 18b: tag  8b index: 256 entries/cache  (4b: 16 wds/block  2b: 4 Byte/wd)

IMPORTANT SLIDE!

Bits: (1-way) Direct Mapped Cache
16KB = $2^8$=index sets x 1 blk/set x $2^6$=4+2=offset B/blk

Index => cache set
Location of all possible blocks

Must check tag for each block:
No need to check index, offset bits

Increasing associativity:
Shrinks index & expands tag size

Bit Fields in Memory Address Used to Access “Cache” Word
Q3: Which block to replace after a miss? 
(After start up, cache is nearly always full)

- Easy if Direct Mapped (only 1 block “1 way” per set index)
- If Set Associative or Fully Associative, must choose:
  - Random (“Ran”) Easy to implement, but not best. If only 2-way: 1 bit/way
  - LRU (Least Recently Used) LRU is best, but hard to implement if > 8-way
    Also other LRU approximations better than Random

Miss Rates for 3 Cache Sizes & Associativities

<table>
<thead>
<tr>
<th>Associativity</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSize</td>
<td>LRU</td>
<td>Ran</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>

Random picks =⇒ same low miss rate as LRU for large caches
Q4: Write policy: What happens if write \((\text{store})\) data?

<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. \text{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>

Additional option -- let writes to an un-cached address allocate a new cache line ("write-allocate"), else just Write-Through.
Write Buffers for Write-Through Caches

Write buffer holds (addresses&) data awaiting write-through to lower levels

Q. Why a write buffer?
A. So CPU not stall for writes

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

Q. Are Read After Write (RAW) hazards an issue for write buffer?
A. Yes! Drain buffer before next read or check buffer addresses before read-miss.
5 Basic Cache Optimizations

• Reducing Miss Rate
  1. Larger Block size (reduce **Compulsory, “cold”, misses**)
  2. Larger Cache size (reduce **Capacity misses**)
  3. Higher Associativity (reduce **Conflict misses**) (... and multiprocessors have cache **Coherence misses**) (4 Cs)

• Reducing Miss Penalty
  4. Multilevel Caches \{total miss rate = \( \prod \) (local miss rate\(_k\)),
     where \( \prod \) means product of all items\(_k\), for \( k = 1 \) to max. \}

• Reducing Hit Time (minimize cache latency)
  5. Giving Reads Priority over Writes, since CPU is waiting. Read completes before earlier writes in write buffer
Outline

• Review
• Memory hierarchy
• Locality
• Cache design
• Virtual address spaces
• Page table layout
• TLB design options
• Conclusion
The Limits of Physical Addressing

Programs use “Physical addresses” of memory locations

CPU
A0-A31
D0-D31

Simple addressing method of archaic pre-1978 computers

Memory
A0-A31
D0-D31

All programs shared one address space: The **physical** address space

Machine language programs had to be aware of the machine organization

No way to prevent a program from accessing any machine resource in memory
Solution: Add a Layer of Indirection

All user programs run in an standardized \textbf{virtual} address space starting at zero.

Needs fast(!) \textbf{Address Translation} hardware, managed by the operating system (OS), to map each virtual address to physical memory.

Hardware supports “modern” OS features: \textbf{Memory protection, Address translation, Sharing}
Three Advantages of Virtual Memory

• Translation:
  – Program can be given consistent view of memory, even though physical memory is scrambled (pages of programs in any order in physical RAM)
  – Makes multithreading reasonable (now used a lot!)
  – Only the most important part of each program (“the Working Set”) must be in physical memory at any one time.
  – Contiguous structures (like stacks) use only as much physical memory as necessary, yet still can grow later as needed, without recopying.

• Protection (most important now):
  – Different threads (or processes) protected from each other.
  – Different pages can be given special behavior
    Examples (Read Only, Invisible to user programs, Not cached).
  – Kernel and OS data are protected from access by User programs
  – Very important for protection from malicious programs

• Sharing:
  – Can map same physical page to multiple users
    (“Shared memory” holding C++ compiler for many users at once.)
Page tables encode mappings of code’s virtual addresses to physical memory address space.

A virtual address space (V.A.S.) is divided into blocks of memory called pages.

A machine usually supports pages of a few sizes (MIPS R4000).

A page table is indexed by a virtual address.

A valid page table entry codes the present physical memory “frame” address for the page.

OS manages the page table for each V.A.S. ID.
Details of Page Table

- Page table maps virtual page numbers to physical frames ("PTE" = Page Table Entry)
- Virtual memory treats main memory = cache for disk
All page tables may not fit in memory!

A table for 4KB pages for a 32-bit physical address space (max 4GB) has 1M entries
Each process needs its own address space tables!

Two-level Page Tables

32 bit virtual address

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>21</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 index</td>
<td>P2 index</td>
<td>Page Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Single top-level table **wired** (stays) in main memory

Only a subset of the 1024 second-level tables are in main memory; rest are on disk or unallocated
VM and Disk: Page replacement policy

Set of all pages in Memory

Head pointer:
Place page on free list if used bit is still clear (=0).

Schedule freed pages with dirty bit set to be written to disk.

Tail pointer:
Clear (=0) “used bits" in page table, says maybe not used recently.

Architect’s role:
support setting dirty and used bits

Dirty bit: page written.

Used bit: set to 1 on any reference

Page Table

<table>
<thead>
<tr>
<th>dirty</th>
<th>used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Freelist

Free Pages
TLB Design Concepts
MIPS Address Translation: How does it work?

"Virtual Addresses"

A0-A31
CPU
D0-D31

"Physical Addresses"

Virtual
Physical
Translation Look-Aside Buffer (TLB)

A0-A31
Memory
D0-D31

Data

Per-Process Translation Look-Aside Buffer (TLB)

A small very fast fully-associative cache of mappings from virtual to physical addresses

TLB also contains protection bits for virtual address

Fast common case: If virtual address is in TLB, process has permission to read/write it.

What is the table of mappings that it caches?

Recently used VA⇒PA entries.
V = 0 pages reside on disk or have not yet been allocated to this ASID. OS handles V = 0 as a "Page fault".

Physical and virtual pages must be the same size! Here, 1024 Bytes/page each.

MIPS handles TLB misses in software (random replacement). Other machines use hardware.

The TLB caches page table entries.
Can TLB translation overlap cache indexing? {Maybe if cache is tiny.}

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag Part of Physical Addr = Physical Page Number</td>
<td>Index</td>
</tr>
<tr>
<td></td>
<td>Byte Select</td>
</tr>
</tbody>
</table>

Virtual Translation Look-Aside Buffer (TLB)

Physical

Cache Tags Valid Cache Data

Cache Block

Cache Block

Cache Block

Virtual Page Number

Page Offset

Tag Part of Physical Addr = Physical Page Number

Index

Byte Select

Virtual Translation Look-Aside Buffer (TLB)

Physical

Cache Tags

Valid

Cache Data

Cache Block

Cache Block

Cache Block

Cache Tag

Hit

Data out

Having cache index in page offset works, but

Q. What is the downside?

A. Inflexibility. Size of cache limited by page size.
Problems With Overlapped TLB Access

Overlapped access only works so long as the address bits used to index into the cache do not change as the result of VA translation.

This usually limits overlapping to small caches, large page sizes, or high n-way set associative caches if you want a large capacity cache.

Example: suppose everything the same except that the cache is increased to 8 KBytes instead of 4 KB:

```
11  2

00

20  12

virt page #  disp
```

This bit is changed by VA translation, but it is needed for cache lookup.

Solutions:
- go to 8KByte page sizes;
- go to 2-way set associative cache; or
Can CPU use virtual addresses for cache?

If cache index in virtual address, only cache misses use TLB!

**Downside:** a subtle, fatal problem. What is it?

*(Aliasing)*

A. Synonym problem. If two address spaces share a physical frame, data may be in cache twice. Maintaining consistency is a nightmare.
Summary #1/3: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation

- The optimal choice is a compromise
  - depends on access characteristics
    » workload
    » use (I-cache, D-cache, TLB)
  - depends on technology / cost

- Simplicity often wins
Summary #2/3: Caches

• The Principle of Locality:
  – Program access a relatively small portion of the address space at any instant of time.
    » **Temporal Locality**: Locality in Time
    » **Spatial Locality**: Locality in Space

• Three Major Uniprocessor Categories of Cache Misses:
  – **Compulsory Misses**: sad facts of life. Example: cold start misses.
  – **Capacity Misses**: increase cache size
  – **Conflict Misses**: increase cache size and/or associativity.
    Nightmare Scenario: ping pong effect!
  – **Coherence Misses (in multiprocessors)**: Avoid frequent use of shared variables in parallel programs.

• Write Policy: **Write Through** vs. **Write Back**

• Today CPU time is a function of (ops, cache misses) vs. just f(ops): Increasing performance affects Compilers, Data structures, and Algorithms
Summary #3/3: TLB, Virtual Memory

- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance
  - This decade is a funny time, since most systems cannot access all of 2nd level cache without TLB misses! The answer in newer processors is 2-levels of TLB.

- Caches, TLBs, Virtual Memory all understood by examining how they deal with four questions:
  1) Where can a block be placed?
  2) How is a block found?
  3) What block is replaced on a miss?
  4) How are writes handled?

- Today VM allows many processes to share single memory without having to swap all processes to disk; today VM protection is more important than memory hierarchy benefits, but computers are still insecure

- Short in-class openbook quiz on appendices A-C & Chapter 1 near start of next (2/22 or 2/24) class. Bring a calculator.

  (Please put your best email address on your exam.)