New-School Machine Structures
(It’s a bit more complicated!)

- Parallel Requests
  Assigned to computer
e.g., Search “Katz”

- Parallel Threads
  Assigned to core
e.g., Lookup, Ads

- Parallel Instructions
  >1 instruction @ one time
e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
e.g., Add of 4 pairs of words

- Hardware descriptions
  All gates @ one time

- Programming Languages

How do we know?
Agenda

• Defining Performance
• Administrivia
• Memory Hierarchy
• Technology Break
• Direct Mapped Caches
• And in Conclusion …
What is Performance?

- *Latency* (or *response time* or *execution time*)
  - Time to complete one task
- *Bandwidth* (or *throughput*)
  - Tasks completed per unit time

---

Cloud Performance: Why Application Latency Matters

<table>
<thead>
<tr>
<th>Server Delay (ms)</th>
<th>Increased time to next click (ms)</th>
<th>Queries/user</th>
<th>Any clicks/user</th>
<th>User satisfaction</th>
<th>Revenue/Value User</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>200</td>
<td>500</td>
<td>--</td>
<td>-0.3%</td>
<td>-0.4%</td>
<td>--</td>
</tr>
<tr>
<td>500</td>
<td>1200</td>
<td>--</td>
<td>-1.0%</td>
<td>-0.9%</td>
<td>-1.2%</td>
</tr>
<tr>
<td>1000</td>
<td>1900</td>
<td>-0.7%</td>
<td>-1.9%</td>
<td>-1.6%</td>
<td>-2.8%</td>
</tr>
<tr>
<td>2000</td>
<td>3100</td>
<td>-1.8%</td>
<td>-4.4%</td>
<td>-3.8%</td>
<td>-4.3%</td>
</tr>
</tbody>
</table>

Figure 6.10 Negative impact of delays at Bing search server on user behavior [Brutlag and Schurman 2009].

- Key figure of merit: application responsiveness
  - Longer the delay, the fewer the user clicks, the less the user happiness, and the lower the revenue per user
Defining CPU Performance

- What does it mean to say X is faster than Y?
- Ferrari vs. School Bus?
- 2013 Ferrari 599 GTB
  - 2 passengers, 11.1 secs in quarter mile
- 2013 Type D school bus
  - 54 passengers, quarter mile time? [Link to YouTube video]
- Response Time/Latency: e.g., time to travel ¼ mile
- Throughput/Bandwidth: e.g., passenger-mi in 1 hour

Defining Relative CPU Performance

- \( \text{Performance}_x = 1/\text{Program Execution Time}_x \)
- \( \text{Performance}_x > \text{Performance}_y \Rightarrow 1/\text{Execution Time}_x > 1/\text{Execution Time}_y \Rightarrow \text{Execution Time}_y > \text{Execution Time}_x \)
- Computer X is N times faster than Computer Y
  - \( \text{Performance}_x / \text{Performance}_y = N \) or \( \text{Execution Time}_y / \text{Execution Time}_x = N \)
- Bus is to Ferrari as 12 is to 11.1: Ferrari is 1.08 times faster than the bus!
Measuring CPU Performance

- Computers use a clock to determine when events takes place within hardware
- *Clock cycles*: discrete time intervals
  - aka clocks, cycles, clock periods, clock ticks
- *Clock rate* or *clock frequency*: clock cycles per second (inverse of clock cycle time)
- 3 GigaHertz clock rate
  => clock cycle time = \(1/(3 \times 10^9)\) seconds
  clock cycle time = 333 picoseconds (ps)

CPU Performance Factors

- To distinguish between processor time and I/O, *CPU time* is time spent in processor
- CPU Time/Program
  = Clock Cycles/Program
  \(\times\) Clock Cycle Time
- Or
  CPU Time/Program
  = Clock Cycles/Program \(\div\) Clock Rate
CPU Performance Factors

• But a program executes instructions
• CPU Time/Program
  = Clock Cycles/Program x Clock Cycle Time
  = Instructions/Program
    x Average Clock Cycles/Instruction
    x Clock Cycle Time
• 1st term called Instruction Count
• 2nd term abbreviated CPI for average Clock Cycles Per Instruction
• 3rd term is 1 / Clock rate

Restating Performance Equation

• Time = Seconds
  Program
  = Instructions
  Program
  x Clock cycles
  Instruction
  x Seconds
  Clock Cycle
What Affects Each Component? 
Instruction Count, CPI, Clock Rate

<table>
<thead>
<tr>
<th>Hardware or software component?</th>
<th>Affects What?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td></td>
</tr>
<tr>
<td>Programming Language</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td></td>
</tr>
<tr>
<td>Instruction Set Architecture</td>
<td></td>
</tr>
</tbody>
</table>

Computer A clock cycle time 250 ps, $\text{CPI}_A = 2$
Computer B clock cycle time 500 ps, $\text{CPI}_B = 1.2$
Assume A and B have same instruction set
Which statement is true?

- ☐ Computer A is $\approx 1.2$ times faster than B
- ☐ Computer A is $\approx 4.0$ times faster than B
- ☐ Computer B is $\approx 1.7$ times faster than A
- ☐ Computer B is $\approx 3.4$ times faster than A
Workload and Benchmark

• **Workload**: Set of programs run on a computer
  – Actual collection of applications run or made from real programs to approximate such a mix
  – Specifies both programs and relative frequencies

• **Benchmark**: Program selected for use in comparing computer performance
  – Benchmarks form a workload
  – Usually standardized so that many use them

SPEC
(System Performance Evaluation Cooperative)

• Computer Vendor cooperative for benchmarks, started in 1989

• SPECCPU2006
  – 12 Integer Programs
  – 17 Floating-Point Programs

• Often turn into number where bigger is faster

• **SPECratio**: reference execution time on old reference computer divide by execution time on new computer to get an effective speed-up
### SPECINT2006 on AMD Barcelona

<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction Count (B)</th>
<th>CPI</th>
<th>Clock cycle time (ps)</th>
<th>Execution Time (s)</th>
<th>Reference Time (s)</th>
<th>SPEC-ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted string processing</td>
<td>2,118</td>
<td>0.75</td>
<td>400</td>
<td>637</td>
<td>9,770</td>
<td>15.3</td>
</tr>
<tr>
<td>Block-sorting compression</td>
<td>2,389</td>
<td>0.85</td>
<td>400</td>
<td>817</td>
<td>9,650</td>
<td>11.8</td>
</tr>
<tr>
<td>GNU C compiler</td>
<td>1,050</td>
<td>1.72</td>
<td>400</td>
<td>724</td>
<td>8,050</td>
<td>11.1</td>
</tr>
<tr>
<td>Combinatorial optimization</td>
<td>336</td>
<td>10.0</td>
<td>400</td>
<td>1,345</td>
<td>9,120</td>
<td>6.8</td>
</tr>
<tr>
<td>Go game</td>
<td>1,658</td>
<td>1.09</td>
<td>400</td>
<td>721</td>
<td>10,490</td>
<td>14.6</td>
</tr>
<tr>
<td>Search gene sequence</td>
<td>2,783</td>
<td>0.80</td>
<td>400</td>
<td>890</td>
<td>9,330</td>
<td>10.5</td>
</tr>
<tr>
<td>Chess game</td>
<td>2,176</td>
<td>0.96</td>
<td>400</td>
<td>837</td>
<td>12,100</td>
<td>14.5</td>
</tr>
<tr>
<td>Quantum computer simulation</td>
<td>1,623</td>
<td>1.61</td>
<td>400</td>
<td>1,047</td>
<td>20,720</td>
<td>19.8</td>
</tr>
<tr>
<td>Video compression</td>
<td>3,102</td>
<td>0.80</td>
<td>400</td>
<td>993</td>
<td>22,130</td>
<td>22.3</td>
</tr>
<tr>
<td>Discrete event simulation library</td>
<td>587</td>
<td>2.94</td>
<td>400</td>
<td>690</td>
<td>6,250</td>
<td>9.1</td>
</tr>
<tr>
<td>Games/path finding</td>
<td>1,082</td>
<td>1.79</td>
<td>400</td>
<td>773</td>
<td>7,020</td>
<td>9.1</td>
</tr>
<tr>
<td>XML parsing</td>
<td>1,058</td>
<td>2.70</td>
<td>400</td>
<td>1,143</td>
<td>6,900</td>
<td>6.0</td>
</tr>
</tbody>
</table>

---

### Summarizing Performance ...

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

*Flashcard Quiz: Which system is faster?*

**System A**  
**System B**  
Same performance  
Unanswerable question!
Summarizing SPEC Performance

- Varies from 6x to 22x faster than reference computer
- Geometric mean of ratios:
  \[ \sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i} \]
  - Geometric Mean gives same relative answer no matter what computer is used as reference
- Geometric Mean for Barcelona is 11.7

Agenda

- Defining Performance
- Administrivia
- Memory Hierarchy
- Technology Break
- Direct Mapped Caches
- And in Conclusion ...
Administrivia

• Lab #5, Homework #4, Project #2-1
• Midterm, 17 October, 6-9 PM

Agenda

• Defining Performance
• Administrivia
• Memory Hierarchy
• Technology Break
• Direct Mapped Caches
• And in Conclusion ...
Big Idea: Memory Hierarchy

- **Processor**
  - **Increasing distance from processor, decreasing speed**
  - **Level 1**
  - **Level 2**
  - **Level 3**
  - **...**
  - **Level n**

- **Size of memory at each level**
  - As we move to outer levels the latency goes up and price per bit goes down. Why?

Library Analogy

- Writing a report based on books on reserve
  - E.g., works of J.D. Salinger
- Go to library to get reserved book and place on desk in library
- If need more, check them out and keep on desk
  - But don’t return earlier books since might need them
- You hope this collection of ~10 books on desk enough to write report, despite 10 being only 0.00001% of books in UC Berkeley libraries
Principle of Locality

• *Principle of Locality*: Programs access small portion of address space at any instant of time
• What program structures lead to locality in instruction accesses?

Cache Philosophy

• Programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  – Works fine even if programmer has no idea what a cache is
  – However, performance-oriented programmers today sometimes “reverse engineer” cache design to design data structures to match cache
  – We’ll do that in Project 3
Memory Access without Cache

- Load word instruction: `lw $t0, 0($t1)`
- $t1 contains $1022_{ten}$, Memory[$1022$] = 99

1. Processor issues address $1022_{ten}$ to Memory
2. Memory reads word at address $1022_{ten}$ (99)
3. Memory sends 99 to Processor
4. Processor loads 99 into register $t1$

Memory Access with Cache

- Load word instruction: `lw $t0, 0($t1)`
- $t1$ contains $1022_{ten}$, Memory[$1022$] = 99
- With cache (similar to a hash)
  1. Processor issues address $1022_{ten}$ to Cache
  2. Cache checks to see if has copy of data at address $1022_{ten}$
     2a. If finds a match (Hit): cache reads 99, sends to processor
     2b. No match (Miss): cache sends address $1022_{ten}$ to Memory
        I. Memory reads 99 at address $1022_{ten}$
        II. Memory sends 99 to Cache
        III. Cache replaces word with new 99
        IV. Cache sends 99 to processor
  3. Processor loads 99 into register $t1$
Cache “Tags”

- Need way to tell if have copy of location in memory so that can decide on hit or miss
- On cache miss, put memory address of block in “tag address” of cache block
  - 1022 placed in tag next to data from memory (99)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>252</td>
<td>12</td>
</tr>
<tr>
<td>1022</td>
<td>99</td>
</tr>
<tr>
<td>131</td>
<td>7</td>
</tr>
<tr>
<td>2041</td>
<td>20</td>
</tr>
</tbody>
</table>

Anatomy of a 16 Byte Cache, 4 Byte Block

- Operations:
  1. Cache Hit
  2. Cache Miss
  3. Refill cache from memory
- Cache needs Address Tags to decide if Processor Address is a Cache Hit or Cache Miss
  - Compares all 4 tags
Cache Requirements

- Suppose processor now requests location 511, which contains 11?
- Doesn’t match any cache block, so must “evict” one resident block to make room
  - Which block to evict?
- Replace “victim” with new memory block at address 511

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>252</td>
<td>12</td>
</tr>
<tr>
<td>1022</td>
<td>99</td>
</tr>
<tr>
<td>511</td>
<td>11</td>
</tr>
<tr>
<td>2041</td>
<td>20</td>
</tr>
</tbody>
</table>

Block Must be Aligned in Memory

- Word blocks are aligned, so binary address of all words in cache always ends in $00_{two}$
- How to take advantage of this to save hardware and energy?
- Don’t need to compare last 2 bits of 32-bit byte address (comparator can be narrower)
  => Don’t need to store last 2 bits of 32-bit byte address in Cache Tag (Tag can be narrower)
Anatomy of a 32B Cache, 8B Block

- Blocks must be aligned in pairs, otherwise could get same word twice in cache
  ⇒ Tags only have even-numbered words
  ⇒ Last 3 bits of address always 000<sub>two</sub>
  ⇒ Tags, comparators can be narrower
- Can get hit for either word in block

Big Idea: Locality

- **Temporal Locality** (locality in time)
  - Go back to same book on desktop multiple times
  - If a memory location is referenced, then it will tend to be referenced again soon
- **Spatial Locality** (locality in space)
  - When go to book shelf, pick up multiple books on J.D. Salinger since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
Principle of Locality

• *Principle of Locality*: Programs access small portion of address space at any instant of time
• What program structures lead to temporal and spatial locality in instruction accesses?
• In data accesses?

Common Cache Optimizations

• Reduce tag overhead by having larger blocks
  – E.g., 2 words, 4 words, 8 words
• Separate caches for instructions and data
  – Double bandwidth, don’t interfere with each other
• Bigger caches (but access time could get bigger than one clock cycle if too big)
• Divide cache into multiple sets, only search inside one set => saves comparators, energy
  – If as many sets as blocks, then only 1 comparator (aka Direct-Mapped Cache)
  – But may increase Miss Rate
Hardware Cost of Cache

- Need to compare every tag to the Processor address
- Comparators are expensive
- Optimization: 2 sets => ½ comparators
- 1 Address bit selects which set

Processor Address Fields used by Cache Controller

- **Block Offset**: Byte address within block
- **Set Index**: Selects which set
- **Tag**: Remaining portion of processor address

<table>
<thead>
<tr>
<th>Processor Address (32-bits total)</th>
<th>Tag</th>
<th>Set Index</th>
<th>Block offset</th>
</tr>
</thead>
</table>

- Size of Index = \( \log_2 \) (number of sets)
- Size of Tag = Address size – Size of Index – \( \log_2 \) (number of bytes/block)
What is limit to number of sets?

• Can save more comparators if have more than 2 sets
• Limit: As Many Sets as Cache Blocks – only needs one comparator!
• Called “Direct-Mapped” Design

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Block offset</th>
</tr>
</thead>
</table>

One More Detail: Valid Bit

• When start a new program, cache does not have valid information for this program
• Need an indicator whether this tag entry is valid for this program
• Add a “valid bit” to the cache tag entry
  – 0 => cache miss, even if by chance, address = tag
  – 1 => cache hit, if processor address = tag
Agenda

• Defining Performance
• Administrivia
• Memory Hierarchy
• Technology Break
• Direct Mapped Caches
• And in Conclusion ...
Direct-Mapped Cache Example

- One word blocks, cache size = 1K words (or 4KB)

```
Valid bit ensures something useful in cache for this index

Compare Tag with upper part of Address to see if a Hit

What kind of locality are we taking advantage of?
```

```
Comparator

Read data from cache instead of memory if a Hit

Block offset
```

```
Hit rate: fraction of access that hit in the cache
Miss rate: 1 – Hit rate
Miss penalty: time to replace a block from lower level in memory hierarchy to cache
Hit time: time to access cache memory (including tag comparison)
```

```
Abbreviation: “$” = cache (A Berkeley innovation!)
```
Mapping a 6-bit Memory Address

- In example, block size is 4 bytes/1 word (it could be multi-word)
- Memory and cache blocks are the same size, unit of transfer between memory and cache
- # Memory blocks >> # Cache blocks
  - 16 Memory blocks/16 words/64 bytes/6 bits to address all bytes
  - 4 Cache blocks, 4 bytes (1 word) per block
  - 4 Memory blocks map to each cache block
- Byte within block: low order two bits, ignore! (nothing smaller than a block)
- Memory block to cache block, aka index: middle two bits
- Which memory block is in a given cache block, aka tag: top two bits

Caching: A Simple First Example

Q: Is the mem block in cache?

Compare the cache tag to the high-order 2 memory address bits to tell if the memory block is in the cache (provided valid bit is set)

Main Memory

One word blocks.
Two low-order bits define the byte in the block (32b words).

Q: Where in the cache is the memory block?

Use next 2 low-order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)
Multiword-Block Direct-Mapped Cache

• Four words/block, cache size = 1K words

![Diagram of Multiword-Block Direct-Mapped Cache]

What kind of locality are we taking advantage of?

Cache Names for Each Organization

• "Fully Associative": Block can go anywhere
  – First design in lecture
  – Note: No Index field, but 1 comparator/block

• "Direct Mapped": Block goes one place
  – Note: Only 1 comparator
  – Number of sets = number blocks

• "N-way Set Associative": N places for a block
  – Number of sets = number of blocks / N
  – Fully Associative: N = number of blocks
  – Direct Mapped: N = 1
Range of Set-Associative Caches

• For a fixed-size cache, each increase by a factor of 2 in associativity doubles the number of blocks per set (i.e., the number of “ways”) and halves the number of sets –
  • decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

More Associativity (more ways)

| Tag | Index | Block offset |

Note: IBM persists in calling sets “ways” and ways “sets”. They’re wrong.

For S sets, N ways, B blocks, which statements hold?

A) The cache has B tags
B) The cache needs N comparators
C) B = N x S
D) Size of Index = \( \log_2(S) \)

- A only
- A and B only
- A, B, and C only
- All four statements are true
Typical Memory Hierarchy

- **Principle of locality + memory hierarchy** presents programmer with ≈ as much memory as is available in the cheapest technology at the ≈ speed offered by the fastest technology

And In Conclusion, ...

- Time (seconds/program) is measure of performance
  \[\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}\]

- Principle of Locality for Libraries /Computer Memory
- Hierarchy of Memories (speed/size/cost per bit) to Exploit Locality
- Cache – copy of data lower level in memory hierarchy
- Direct Mapped to find block in cache using Tag field and Valid bit for Hit
- Larger caches reduce Miss rate via Temporal and Spatial Locality, but can increase Hit time
- Multilevel caches help Miss penalty
- AMAT helps balance Hit time, Miss rate, Miss penalty