Memory Hierarchy Design

- Memory hierarchy design becomes more crucial with recent multi-core processors:
  - Aggregate peak bandwidth grows with # cores:
    - Intel Core i7 can generate two references per core per clock
    - Four cores and 3.2 GHz clock
      - 25.6 *10^9 64-bit data references/second +
      - 12.8 *10^9 128-bit instruction references
      - = 409.6 GB/s!
  - DRAM bandwidth is only 6% of this (25 GB/s)
  - Requires:
    - Multi-port, pipelined caches
    - Two levels of cache per core
    - Shared third-level cache on chip
Memory Hierarchy Basics

- \( n \) sets => \( n \)-way set associative
  - Direct-mapped cache => one block per set
  - Fully associative => one set

- Writing to cache: two strategies
  - Write-through
    - Immediately update lower levels of hierarchy
  - Write-back
    - Only update lower levels of hierarchy when an updated block is replaced
  - Both strategies use write buffer to make writes asynchronous

Memory Hierarchy Basics

- Miss rate
  - Fraction of cache access that result in a miss

- Causes of misses
  - Compulsory
    - First reference to a block
  - Capacity
    - Blocks discarded and later retrieved
  - Conflict
    - Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions

- Miss cycles per instruction
  - I-cache: \( 0.02 \times 100 = 2 \)
  - D-cache: \( 0.36 \times 0.04 \times 100 = 1.44 \)

- Actual CPI = 2 + 2 + 1.44 = 5.44
- Ideal CPU is 5.44/2 = 2.72 times faster

Introduction

Note 1: miss rate/penalty are often different for reads and writes
Average memory access time = Hit time + Miss rate \( \times \) Miss penalty

Note 2: speculative and multithreaded processors may execute other instructions during a miss
- Reduces performance impact of misses
Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9
- Now add L-2 cache …

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
  - CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
  - Performance ratio = 9/3.4 = 2.6

Memory Hierarchy Basics

- Six basic cache optimizations:
  - Larger block size
    - Reduces compulsory misses
  - Larger total cache capacity to reduce miss rate
    - Increases hit time, increases power consumption
  - Higher associativity
    - Reduces conflict misses
  - Multilevel caches to reduce miss penalty
    - Reduces overall memory access time
  - Giving priority to read misses over writes
    - Reduces miss penalty
  - Avoiding address translation in cache indexing
    - Reduces hit time
### 3-Level Cache Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 caches</strong> (per core)</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td></td>
<td>L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles</td>
</tr>
<tr>
<td><strong>L2 unified cache</strong> (per core)</td>
<td>256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
</tr>
<tr>
<td><strong>L3 unified cache</strong> (shared)</td>
<td>8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a</td>
<td>2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles</td>
</tr>
</tbody>
</table>

n/a: data not available

### Ten Advanced Optimizations

- Reducing the hit time
  - small & simple first-level caches
  - way-prediction
- Increase cache bandwidth
  - pipelined cache access
  - nonblocking caches
  - multibanked caches
- Reducing the miss penalty
  - critical word first
  - merging write buffers
- Reducing the miss rate
  - compiler optimizations
- Reducing the miss penalty or miss rate via parallelism
  - hardware prefetching of instructions and data
  - compiler-controlled prefetching

### Multilevel On-Chip Caches

**Intel Nehalem 4-core processor**

Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

### 1. Small and simple 1st level caches

- Small and simple first level caches
  - Critical timing path:
    - addressing tag memory, then
    - comparing tags, then
    - selecting correct set
  - Direct-mapped caches can overlap tag compare and transmission of data
  - Lower associativity reduces power because fewer cache lines are accessed
2. Way Prediction

- To improve hit time, predict the way to pre-set mux
  - Mis-prediction gives longer hit time
  - Prediction accuracy
    - > 90% for two-way
    - > 80% for four-way
    - L-cache has better accuracy than D-cache
  - First used on MIPS R10000 in mid-90s
  - Used on ARM Cortex-A8
- Extend to predict block as well
  - “Way selection”
  - Increases mis-prediction penalty

3. Pipelining Cache

- Pipeline cache access to improve bandwidth
  - Examples:
    - Pentium: 1 cycle
    - Pentium Pro – Pentium III: 2 cycles
    - Pentium 4 – Core i7: 4 cycles
- Increases branch mis-prediction penalty
- Makes it easier to increase associativity
4. Nonblocking Caches

- Allow hits before previous misses complete
  - “Hit under miss”
  - “Hit under multiple miss”
- L2 must support this
- In general, processors can hide L1 miss penalty but not L2 miss penalty

5. Multibanked Caches

- Organize cache as independent banks to support simultaneous access
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2
- Interleave banks according to block address

6. Critical Word First, Early Restart

- Critical word first
  - Request missed word from memory first
  - Send it to the processor as soon as it arrives
- Early restart
  - Request words in normal order
  - Send missed work to the processor as soon as it arrives
- Effectiveness of these strategies depends on block size and likelihood of another access to the portion of the block that has not yet been fetched

7. Merging Write Buffer

- When storing to a block that is already pending in the write buffer, update write buffer
- Reduces stalls due to full write buffer
- Do not apply to I/O addresses
8. Compiler Optimizations

- **Loop Interchange**
  - Swap nested loops to access memory in sequential order

- **Blocking**
  - Instead of accessing entire rows or columns, subdivide matrices into blocks
  - Requires more memory accesses but improves locality of accesses

9. Hardware Prefetching

- Fetch two blocks on miss (include next sequential block)

### Pentium 4 Pre-fetching

<table>
<thead>
<tr>
<th>Technique</th>
<th>Hit time</th>
<th>Bandwidth</th>
<th>Miss penalty</th>
<th>Miss rate</th>
<th>Power consumption</th>
<th>Hardware cost/complexity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small and simple caches</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Trivial; widely used</td>
</tr>
<tr>
<td>Way-predicting caches</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>1</td>
<td>Used in Pentium 4</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Piped cached access</td>
<td>-</td>
<td>+</td>
<td>1</td>
<td>3</td>
<td>Widely used</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Non-blocking caches</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>1</td>
<td>Used in L2 of both 37 and Cortex-A5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Banked caches</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>1</td>
<td>Widely used with write through</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Critical word first and early request</td>
<td>+</td>
<td>2</td>
<td>Widely used</td>
<td>+</td>
<td>3</td>
<td>Widely used with write through</td>
<td>-</td>
</tr>
<tr>
<td>Merge write buffer</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>1</td>
<td>Widely used with write through</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Compiler techniques to reduce cache misses</td>
<td>+</td>
<td>0</td>
<td>Software is a challenge, but many compilers handle common cases and reduce cache misses</td>
<td>-</td>
<td>0</td>
<td>Most provide prefetching instructions; modern high-end processors also automatically prefetch in hardware.</td>
<td>-</td>
</tr>
<tr>
<td>Hardware prefetching of instructions and data</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>2 more, 3 data</td>
<td>-</td>
<td>Most provide prefetching instructions; modern high-end processors also automatically prefetch in hardware.</td>
<td>-</td>
</tr>
<tr>
<td>Compiler-controlled prefetching</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td>Reads non-blocking cache; possible instruction overhead, in many CPUs</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

10. Compiler Prefetching

- Insert prefetch instructions before data is needed
- Non-faulting: prefetch doesn’t cause exceptions
- Register prefetch
  - Loads data into register
- Cache prefetch
  - Loads data into cache
- Combine with loop unrolling and software pipelining

### Summary

- **Technique**
- **Hit time**
- **Bandwidth**
- **Miss penalty**
- **Miss rate**
- **Power consumption**
- **Hardware cost/complexity**
- **Comment**

- Small and simple caches
- Way-predicting caches
- Piped cached access
- Non-blocking caches
- Banked caches
- Critical word first and early request
- Merge write buffer
- Compiler techniques to reduce cache misses
- Hardware prefetching of instructions and data
- Compiler-controlled prefetching