CPE 631 Lecture 05: CPU Caches

Aleksandar Milenkovic, milenka@ece.uah.edu
Electrical and Computer Engineering
University of Alabama in Huntsville

Outline
- Memory Hierarchy
- Four Questions for Memory Hierarchy
- Cache Performance

Solution: The Memory Hierarchy (MH)
- User sees as much memory as is available in cheapest technology and access it at the speed offered by the fastest technology

Levels in Memory Hierarchy
- Processor
- Control
- Datapath

Processor-DRAM Latency Gap
- Processor-Memory Performance Gap grows 50% / year
- DRAM: 2x/10 years
- CPU: 2x/1.5 year

Processor - Memory Gap grows 50% / year
Generations of Microprocessors

<table>
<thead>
<tr>
<th>Time of a full cache miss in instructions executed:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Alpha: 340 ns / 5.0 ns = 68 clks x 2 or 136</td>
<td></td>
</tr>
<tr>
<td>2nd Alpha: 266 ns / 3.3 ns = 80 clks x 4 or 320</td>
<td></td>
</tr>
<tr>
<td>3rd Alpha: 180 ns / 1.7 ns = 108 clks x 6 or 648</td>
<td></td>
</tr>
</tbody>
</table>

1/2X latency x 3X clock rate x 3X Instr/clock ⇒ -5X

Why hierarchy works?

- Principal of locality
  - Programs spend 90% of their execution time in only 10% of code
  - Rule of thumb: Temporal locality; recently accessed items are likely to be accessed in the near future ⇒ Keep them close to the processor
  - Spatial locality: items whose addresses are near one another tend to be referenced close together in time ⇒ Move blocks consisted of contiguous words to the upper level

Cache Measures

- Hit: data appears in some block in the upper level (Bl. X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: time to access the upper level (RAM access time + Time to determine hit/miss)
- Miss: data needs to be retrieved from the lower level (Bl. Y)
  - Miss rate: 1 - (Hit Rate)
  - Miss penalty: time to replace a block in the upper level + time to retrieve the block from the lower level
- Average memory access time
  - = Hit time + Miss rate x Miss penalty (ns or clocks)

Levels of the Memory Hierarchy

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>100s Bytes</td>
<td>1 ns</td>
</tr>
<tr>
<td>Cache</td>
<td>10s-100s K Bytes</td>
<td>1-10 ns</td>
</tr>
<tr>
<td>Main Memory</td>
<td>M Bytes</td>
<td>100ns-300ns</td>
</tr>
<tr>
<td>Disk</td>
<td>10s G Bytes, 10 ms</td>
<td>10,000,000 ns</td>
</tr>
<tr>
<td>Tape</td>
<td>infinite</td>
<td>sec-min</td>
</tr>
</tbody>
</table>

Staging Xfer Unit

- prog./compiler: 1-8 bytes
- cache cntl: 8-128 bytes
- OS: 512-4K bytes
- user/operator: Mbytes

Upper Level: faster
Lower Level: Larger
Four Questions for Memory Heir.

Q#1: Where can a block be placed in the upper level?
⇒ Block placement
- direct-mapped, fully associative, set-associative
Q#2: How is a block found if it is in the upper level?
⇒ Block identification
Q#3: Which block should be replaced on a miss?
⇒ Block replacement
- Random, LRU (Least Recently Used)
Q#4: What happens on a write?
⇒ Write strategy
- Write-through vs. write-back
- Write allocate vs. No write allocate

Direct-Mapped Cache

In a direct-mapped cache, each memory address is associated with one possible block within the cache
- Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
- Block is the unit of transfer between cache and memory

Q1: Where can a block be placed in the upper level?

Block 12 placed in 8 block cache:
- Fully associative, direct mapped,
  2-way set associative
- S.A. Mapping = Block Number Modulo Number Sets

Full Mapped
Direct Mapped
(12 mod 8) = 4
2-Way Assoc
(12 mod 4) = 0

Direct-Mapped Cache (cont’d)
Direct-Mapped Cache (cont’d)

- Since multiple memory addresses map to the same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Result: divide memory address into three fields:

<table>
<thead>
<tr>
<th>Block Address</th>
<th>TAG: to check if have the correct block</th>
<th>INDEX: to select block</th>
<th>OFFSET: to select byte within the block</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttttttttttttttt</td>
<td>11111111</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

Direct-Mapped Cache Terminology

- **INDEX**: specifies the cache index (which “row” of the cache we should look in)
- **OFFSET**: once we have found the correct block, specifies which byte within the block we want
- **TAG**: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

**BLOCK ADDRESS**: TAG + INDEX

Direct-Mapped Cache Example

- Conditions
  - 32-bit architecture (word=32bits), address unit is byte
  - 8KB direct-mapped cache with 4 words blocks
- Determine the size of the Tag, Index, and Offset fields
  - OFFSET (specifies correct byte within block): cache block contains 4 words = 16 (2^4) bytes ⇒ 4 bits
  - INDEX (specifies correct row in the cache): cache size is 8KB = 2^13 bytes, cache block is 2^4 bytes ⇒ 9 bits in cache (1 block = 1 row): 2^13/2^4 = 2^9 bits
  - TAG: Memory address length - offset - index = 32 - 4 - 9 = 19 ⇒ tag is leftmost 19 bits

1 KB Direct Mapped Cache, 32B blocks

- For a 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)

- Conditions
  - 32-bit architecture (word=32bits), address unit is byte
  - 8KB direct-mapped cache with 4 words blocks
- Determine the size of the Tag, Index, and Offset fields
  - OFFSET (specifies correct byte within block): cache block contains 4 words = 16 (2^4) bytes ⇒ 4 bits
  - INDEX (specifies correct row in the cache): cache size is 8KB = 2^13 bytes, cache block is 2^4 bytes ⇒ 9 bits
  - TAG: Memory address length - offset - index = 32 - 4 - 9 = 19 ⇒ tag is leftmost 19 bits

- Diagrams and tables illustrating cache memory address fields and their usage.
Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operate in parallel (N typically 2 to 4)
  - Example: Two-way set associative cache
    - Cache Index selects a "set" from the cache
    - The two tags in the set are compared in parallel
    - Data is selected based on the tag result

Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss

- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.

Q2: How is a block found if it is in the upper level?

- Tag on each block
  - No need to check index or block offset
  - Increasing associativity shrinks index, expands tag

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
  - Set Associative or Fully Associative:
    - Random
    - LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Assoc</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</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>
Q4: What happens on a write?

- **Write through**—The information is written to both the block in the cache and to the block in the lower-level memory.
- **Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
- Is block clean or dirty?
- **Pros and Cons of each?**
  - **WT:** read misses cannot result in writes
  - **WB:** no repeated writes to same location
- WT always combined with write buffers so that don’t wait for lower level memory

Write stall in write through caches

- When the CPU must wait for writes to complete during write through, the CPU is said to write stall
- **Common optimization**
  => Write buffer which allows the processor to continue as soon as the data is written to the buffer, thereby overlapping processor execution with memory updating
- However, write stalls can occur even with write buffer (when buffer is full)

Write Buffer for Write Through

- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) <= 1 / DRAM write cycle
- Memory system designer’s nightmare:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

What to do on a write-miss?

- **Write allocate (or fetch on write)**
  The block is loaded on a write-miss, followed by the write-hit actions
- **No-write allocate (or write around)**
  The block is modified in the memory and not loaded into the cache
- Although either write-miss policy can be used with write through or write back, write back caches generally use write allocate and write through often use no-write allocate
An Example: The Alpha 21264 Data Cache (64KB, 64-byte blocks, 2w)

Cache Performance

- **Hit Time** = time to find and retrieve data from current level cache
- **Miss Penalty** = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- **Hit Rate** = % of requests that are found in current level cache
- **Miss Rate** = 1 - Hit Rate

Cache Performance (cont’d)

- Average memory access time (AMAT)
  
  \[
  \text{AMAT} = \text{Hit Time} \times \text{Miss Rate} \times \text{Miss Penalty} 
  = \% \text{ instructions} \times (\text{Hit time}_{\text{inst}} + \text{Miss Rate}_{\text{inst}} \times \text{Miss Penalty}_{\text{inst}}) 
  = \% \text{ data} \times (\text{Hit time}_{\text{data}} + \text{Miss Rate}_{\text{data}} \times \text{Miss Penalty}_{\text{data}})
  \]

An Example: Unified vs. Separate I&D

- Compare 2 design alternatives (ignore L2 caches)?
  - 16KB I&D: Inst misses=3.82 /1K, Data miss rate=40.9 /1K
  - 32KB unified: Unified misses = 43.3 misses/1K
- Assumptions:
  - Ideal frequency is 36% -> 74% accesses from instructions (1.01.36)
  - hit time = 1clock cycle, miss penalty = 100 clock cycles
  - Data hit has 1 stall for unified cache (only one port)
Unified vs. Separate I&D (cont’d)

Miss rate (L1I) = (# L1I misses) / (IC)

#L1I misses = (L1I misses per 1k) * (IC /1000)

Miss rate (L1I) = 3.82/1000 = 0.0038

Miss rate (L1D) = (# L1D misses) / (# Mem. Refs)

#L1D misses = (L1D misses per 1k) * (IC /1000)

Miss rate (L1D) = 40.9 * (IC/1000) / (0.36*IC) = 0.1136

Miss rate (L1U) = (# L1U misses) / (IC + Mem. Refs)

#L1U misses = (L1U misses per 1k) * (IC /1000)

Miss rate (L1U) = 43.3*(IC / 1000) / (1.36 * IC) = 0.0318

AMAT (split) = (% instr.) * (hit time + L1I miss rate * Miss Pen.) + (% data) * (hit time + L1D miss rate * Miss Pen.) = .74(1 + .0038*100) + .26(1+.1136*100) = 4.2348 clock cycles

AMAT (unif.) = (% instr.) * (hit time + L1Umiss rate * Miss Pen.) + (% data) * (hit time + L1U miss rate * Miss Pen.) = .74(1 + .0318*100) + .26(1 + 1 + .0318*100) ≈ 4.44 clock cycles

AMAT and Processor Performance

Miss-oriented Approach to Memory Access
– CPI_{Exec} includes ALU and Memory instructions

\[
\text{CPU time} = \frac{IC \times (\text{CPI}_{\text{Exec}} + \text{MemAccessInst} \times \text{MissRateInst} \times \text{MissPenalty})}{\text{Clockrate}}
\]

\[
\text{CPU time} = \frac{IC \times (\text{CPI}_{\text{Exec}} + \text{MemMissesInst} \times \text{MissPenalty})}{\text{Clockrate}}
\]

Separating out Memory component entirely
– AMAT = Average Memory Access Time
– CPI_{ALU} does not include memory instructions

\[
\text{CPU time} = \frac{IC \times (\text{ALUop} \times \text{CPI}_{\text{ALUop}} + \text{MemAccessInst} \times \text{AMAT})}{\text{Clockrate}}
\]

\[
\% \text{instructions} \times (\text{Hit time instr} + \text{Miss Rate instr} \times \text{Miss Penalty instr}) + \% \text{data} \times (\text{Hit time data} + \text{Miss Rate data} \times \text{Miss Penalty data})
\]
Summary: 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 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

- **Write Policy**:
  - Write Through: needs a write buffer.
  - Write Back: control can be complex

**How to Improve Cache Performance?**

- **AMAT** = \( \text{HitTime} + \text{MissRate} \times \text{MissPenalty} \)

- **Cache optimizations**
  1. Reduce the miss rate
  2. Reduce the miss penalty
  3. Reduce the time to hit in the cache

Summary: The Cache Design Space

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

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

**Where Misses Come From?**

- Classifying Misses: 3 Cs
  - Compulsory — The first access to a block is not in the cache, so the block must be brought into the cache. Also called cold start misses or first reference misses. (Misses in even an Infinite Cache)
  - Capacity — If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. (Misses in Fully Associative Size X Cache)
  - Conflict — If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses. (Misses in N-way Associative, Size X Cache)

- More recent, 4th “C”:
  - Coherence — Misses caused by cache coherence.
Cache Organization?

- Assume total cache size not changed
- What happens if:
  1) Change Block Size
  2) Change Cache Size
  3) Change Cache Internal Organization
  4) Change Associativity
  5) Change Compiler

- Which of 3Cs is obviously affected?

3Cs Absolute Miss Rate (SPEC92)

- 8-way: conflict misses due to going from fully associative to 8-way assoc.
- 4-way: conflict misses due to going from 8-way to 4-way assoc.
- 2-way: conflict misses due to going from 4-way to 2-way assoc.
- 1-way: conflict misses due to going from 2-way to 1-way assoc. (direct mapped)

3Cs Relative Miss Rate

1st Miss Rate Reduction Technique: Larger Block Size

- Larger Block Size: Reduced compulsory misses, Increased conflict misses
1st Miss Rate Reduction Technique:
Larger Block Size (cont’d)

- Example:
  - Memory system takes 40 clock cycles of overhead, and then delivers 16 bytes every 2 clock cycles.
  - Miss rate vs. block size (see table); hit time is 1 cc.
  - AMAT = Hit Time + Miss Rate x Miss Penalty.

- Block size depends on both latency and bandwidth of lower level memory.
  - Low latency and bandwidth => decrease block size.
  - High latency and bandwidth => increase block size.

<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>64 K</td>
<td>0.49</td>
<td>2.39</td>
<td>4.50</td>
<td>7.46</td>
</tr>
<tr>
<td>128 K</td>
<td>0.51</td>
<td>2.64</td>
<td>4.87</td>
<td>7.82</td>
</tr>
<tr>
<td>256 K</td>
<td>0.70</td>
<td>2.87</td>
<td>5.10</td>
<td>8.04</td>
</tr>
</tbody>
</table>

2nd Miss Rate Reduction Technique:
Larger Caches

- Reduce Capacity misses.
- Drawbacks: Higher cost, Longer hit time.

- Miss rates improve with higher associativity.
  - Two rules of thumb:
    - 8-way set-associative is almost as effective in reducing misses as fully-associative cache of the same size.
    - 2:1 Cache Rule: Miss Rate 2-way associative cache size X = Miss Rate 8-way associative cache size X/2.

- Beware: Execution time is only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%.

<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>64 K</td>
<td>1.02</td>
<td>1.51</td>
<td>1.91</td>
<td>2.31</td>
</tr>
<tr>
<td>128 K</td>
<td>1.06</td>
<td>1.33</td>
<td>1.62</td>
<td>1.90</td>
</tr>
<tr>
<td>256 K</td>
<td>1.15</td>
<td>1.56</td>
<td>1.87</td>
<td>2.16</td>
</tr>
</tbody>
</table>

3rd Miss Rate Reduction Technique:
Higher Associativity

- Miss rates improve with higher associativity.

- Two rules of thumb:
  - 8-way set-associative is almost as effective in reducing misses as fully-associative cache of the same size.
  - 2:1 Cache Rule: Miss Rate DM cache size N = Miss Rate 2-way cache size N/2.

- Beware: Execution time is only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%.

- Block size depends on both latency and bandwidth of lower level memory.
  - Low latency and bandwidth => decrease block size.
  - High latency and bandwidth => increase block size.
3rd Miss Rate Reduction Technique: Higher Associativity (cont'd)

- Example
  - \( \text{CCT}_{2-\text{way}} = 1.10 \times \text{CCT}_{1-\text{way}} \)
  - \( \text{CCT}_{4-\text{way}} = 1.12 \times \text{CCT}_{1-\text{way}} \)
  - \( \text{CCT}_{8-\text{way}} = 1.14 \times \text{CCT}_{1-\text{way}} \)
  - Hit time = 1 cc, Miss penalty = 50 cc
  - Find AMAT using miss rates from Fig 5.9 (old textbook)

<table>
<thead>
<tr>
<th>Cache (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.44</td>
<td>6.64</td>
<td>6.20</td>
<td>6.44</td>
</tr>
<tr>
<td>2</td>
<td>5.90</td>
<td>4.90</td>
<td>4.62</td>
<td>4.99</td>
</tr>
<tr>
<td>4</td>
<td>3.90</td>
<td>3.90</td>
<td>3.67</td>
<td>3.99</td>
</tr>
<tr>
<td>8</td>
<td>3.45</td>
<td>3.35</td>
<td>3.19</td>
<td>3.44</td>
</tr>
<tr>
<td>16</td>
<td>3.00</td>
<td>3.00</td>
<td>2.77</td>
<td>3.03</td>
</tr>
<tr>
<td>32</td>
<td>2.70</td>
<td>2.70</td>
<td>2.55</td>
<td>2.78</td>
</tr>
<tr>
<td>64</td>
<td>2.50</td>
<td>2.50</td>
<td>2.37</td>
<td>2.58</td>
</tr>
<tr>
<td>128</td>
<td>2.30</td>
<td>2.30</td>
<td>2.18</td>
<td>2.38</td>
</tr>
</tbody>
</table>

4th Miss Rate Reduction Technique: Way Prediction, “Pseudo-Associativity”

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
  - **Way Prediction**: extra bits are kept to predict the way or block within a set
    - Mux is set early to select the desired block
    - Only a single tag comparison is performed
    - What if miss?
      -> check the other blocks in the set
  - Used in Alpha 21264 (1 bit per block in IC$)
    - 1 cc if predictor is correct, 3 cc if not
    - Effectiveness: prediction accuracy is 85%
  - Used in MIPS 4300 embedded proc. to lower power

Pseudo-Associative Cache

- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)
- Accesses proceed just as in the DM cache for a hit
- On a miss, check the second entry
  - Simple way is to invert the MSB bit of the INDEX field to find the other block in the “pseudo set”

- Hit time
- Pseudo-Hit Time
- Miss Penalty
- Time

- What if too many hits in the slow part?
  - swap contents of the blocks

Example: Pseudo-Associativity

- Compare 1-way, 2-way, and pseudo associative organizations for 2KB and 128KB caches
- Hit time = 1cc, Pseudo hit time = 2cc
- Parameters are the same as in the previous Exmp.
  - \( \text{AMAT}_{\text{ps}} = \text{Hit Time}_{\text{ps}} + \text{Miss Rate}_{\text{ps}} \times \text{Miss Penalty}_{\text{ps}} \)
  - \( \text{Miss Rate}_{\text{ps}} = \text{Miss Rate}_{\text{2-way}} \)
  - Hit time_{\text{ps}} = Hit time_{\text{2-way}} + Alternate hit rate_{\text{ps}} \times 2
  - Alternate hit rate_{\text{ps}} = Hit rate_{\text{2-way}} - Hit rate_{\text{1-way}} = Miss rate_{\text{1-way}} - Miss rate_{\text{2-way}}

<table>
<thead>
<tr>
<th>Cache (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>Pseudo</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5.30</td>
<td>4.90</td>
<td>4.644</td>
</tr>
<tr>
<td>128</td>
<td>1.50</td>
<td>1.45</td>
<td>1.356</td>
</tr>
</tbody>
</table>
5th Miss Rate Reduction Technique: Compiler Optimizations

- Reduction comes from software (no Hw ch.)
- McFarling [1989] reduced caches misses by 75%
  (8KB, DM, 4 byte blocks) in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - Merging Arrays: Improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop Interchange: change nesting of loops to access data in order stored in memory
  - Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
  - Blocking: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Loop Interchange

- Motivation: some programs have nested loops that access data in nonsequential order
- Solution: Simply exchanging the nesting of the loops can make the code access the data in the order it is stored => reduce misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded

Loop Interchange Example

```c
/* Before */
for (k = 0; k < 100; k = k+1)
   for (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5000; i = i+1)
         x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
         x[i][j] = 2 * x[i][j];
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality.

Reduces misses if the arrays do not fit in the cache.

Blocking

- Motivation: multiple arrays, some accessed by rows and some by columns
- Storing the arrays row by row (row major order) or column by column (column major order) does not help: both rows and columns are used in every iteration of the loop (Loop Interchange cannot help)
- Solution: instead of operating on entire rows and columns of an array, blocked algorithms operate on submatrices or blocks => maximize accesses to the data loaded into the cache before the data is replaced
**Blocking Example**

/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        {r = 0);
            for (k = 0; k < N; k = k+1){
                r = r + y[i][k]*z[k][j];
            }
        x[i][j] = r;
    }

- Two Inner Loops:
  - Read all NxN elements of y[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]

- Capacity Misses - a function of N & Cache Size:
  - $2N^3 + N^2$ (assuming no conflict; otherwise …)

- Idea: compute on BxB submatrix that fits

**Blocking Example (cont’d)**

/* After */
for (jj = 0; jj < N; jj = jj+B)
    for (kk = 0; kk < N; kk = kk+B)
        for (i = 0; i < N; i = i+1)
            for (j = jj; j < min(jj+B-1,N); j = j+1)
                {r = 0;
                    for (k = kk; k < min(kk+B-1,N); k = k+1) {
                        r = r + y[i][k]*z[k][j];
                    }
                x[i][j] = x[i][j] + r;
            }

- B called Blocking Factor
- Capacity Misses from $2N^3 + N^2$ to $N^3/B + 2N^2$
- Conflict Misses Too?

**Merging Arrays**

- Motivation: some programs reference multiple arrays in the same dimension with the same indices at the same time => these accesses can interfere with each other, leading to conflict misses

- Solution: combine these independent matrices into a single compound array, so that a single cache block can contain the desired elements

**Merging Arrays Example**

/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];
/* After: 1 array of structures */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];
Some programs have separate sections of code that access with the same loops, performing different computations on the common data.

Solution: “Fuse” the code into a single loop => the data that are fetched into the cache can be used repeatedly before being swapped out => reducing misses via improved temporal locality.

Loop Fusion Example

```c
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        { a[i][j] = 1/b[i][j] * c[i][j];
          d[i][j] = a[i][j] + c[i][j];
        }
```

2 misses per access to a & c vs. one miss per access; improve temporal locality.

Summary of Compiler Optimizations to Reduce Cache Misses (by hand)

<table>
<thead>
<tr>
<th>Performance Improvement</th>
<th>1</th>
<th>1.5</th>
<th>2</th>
<th>2.5</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>merged arrays</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>loop fusion</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>blocking</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3 Cs: Compulsory, Capacity, Conflict
- 1. Larger Cache => Reduce Capacity
- 2. Larger Block Size => Reduce Compulsory
- 3. Higher Associativity => Reduce Conflicts
- 4. Way Prediction & Pseudo-Associativity
- 5. Compiler Optimizations