Cache Impact On Performance: \textit{An Example}

Assuming the following execution and cache parameters:

- Cache miss penalty = 50 cycles
- Normal instruction execution CPI ignoring memory stalls = 2.0 cycles
- Miss rate = 2%
- Average memory references/instruction = 1.33

\[\text{CPU time} = IC \times [\text{CPI}_{\text{execution}} + \text{Memory accesses/instruction} \times \text{Miss rate} \times \text{Miss penalty}] \times \text{Clock cycle time}\]

\[\text{CPU time with cache} = IC \times (2.0 + (1.33 \times 2\% \times 50)) \times \text{clock cycle time}\]

\[= IC \times 3.33 \times \text{Clock cycle time}\]

\[\rightarrow \text{Lower CPI}_{\text{execution}} \text{ increases the impact of cache miss clock cycles}\]

\[\rightarrow \text{CPUs with higher clock rate, have more cycles per cache miss and more memory impact on CPI}\]
Impact of Cache Organization: An Example

Given:

• A perfect CPI with cache = 2.0 Clock cycle = 2 ns
• 1.3 memory references/instruction Cache size = 64 KB with
• Cache miss penalty = 70 ns, no stall on a cache hit
• One cache is direct mapped with miss rate = 1.4%
• The other cache is two-way set-associative, where:
  – CPU time increases 1.1 times to account for the cache selection multiplexor
  – Miss rate = 1.0%

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

Average memory access time \( t_{1\text{-way}} \) = \( 2.0 + (.014 \times 70) = 2.98 \) ns

Average memory access time \( t_{2\text{-way}} \) = \( 2.0 \times 1.1 + (.010 \times 70) = 2.90 \) ns

CPU time = IC x [CPI\(_{\text{execution}}\) + Memory accesses/instruction x Miss rate x Miss penalty ] x Clock cycle time

CPU\(_{\text{time \ 1-way}}\) = IC x (2.0 x 2 + (1.3 x .014 x 70)) = 5.27 x IC

CPU\(_{\text{time \ 2-way}}\) = IC x (2.0 x 2 x 1.10 + (1.3 x 0.01 x 70)) = 5.31 x IC

→ In this example, 1-way cache offers slightly better performance with less complex hardware.
Types of Cache Misses: "The Three C’s"

1. **Compulsory:** On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.

2. **Capacity:** Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).

3. **Conflict:** In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.
The 3 Cs of Cache:
Absolute Miss Rates (SPEC92)
The 3 Cs of Cache: Relative Miss Rates (SPEC92)
Improving Cache Performance

How?

• Reduce Miss Rate
• Reduce Cache Miss Penalty
• Reduce Cache Hit Time
Improving Cache Performance

• **Miss Rate Reduction Techniques:**
  * Increased cache capacity
  * Higher associativity
  * Hardware prefetching of instructions and data
  * Compiler-controlled prefetching
  * Larger block size
  * Victim caches
  * Pseudo-associative Caches
  * Compiler optimizations

• **Cache Miss Penalty Reduction Techniques:**
  * Giving priority to read misses over writes
  * Early restart and critical word first
  * Second-level cache (L2)
  * Sub-block placement
  * Non-blocking caches

• **Cache Hit Time Reduction Techniques:**
  * Small and simple caches
  * Avoiding address translation during cache indexing
  * Pipelining writes for fast write hits
Miss Rate Reduction Techniques:

Larger Block Size

- A larger block size improves cache performance by taking advantage of spatial locality.
- For a fixed cache size, larger block sizes mean fewer cache block frames.

• Performance keeps improving to a limit when the fewer number of cache block frames increases conflict misses and thus overall cache miss rate.
Miss Rate Reduction Techniques:

Higher Cache Associativity

Example: Average Memory Access Time (A.M.A.T) vs. Miss Rate

<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>1</td>
<td>2.33</td>
<td>2.15</td>
<td>2.07</td>
<td>2.01</td>
</tr>
<tr>
<td>2</td>
<td>1.98</td>
<td>1.86</td>
<td>1.76</td>
<td>1.68</td>
</tr>
<tr>
<td>4</td>
<td>1.72</td>
<td>1.67</td>
<td>1.61</td>
<td>1.53</td>
</tr>
<tr>
<td>8</td>
<td>1.46</td>
<td>1.48</td>
<td>1.47</td>
<td>1.43</td>
</tr>
<tr>
<td>16</td>
<td>1.29</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
</tr>
<tr>
<td>32</td>
<td>1.20</td>
<td>1.24</td>
<td>1.25</td>
<td>1.27</td>
</tr>
<tr>
<td>64</td>
<td>1.14</td>
<td>1.20</td>
<td>1.21</td>
<td>1.23</td>
</tr>
<tr>
<td>128</td>
<td>1.10</td>
<td>1.17</td>
<td>1.18</td>
<td>1.20</td>
</tr>
</tbody>
</table>

(Red means A.M.A.T. not improved by more associativity)
Miss Rate Reduction Techniques: Victim Caches

- Data discarded from cache is placed in an added small buffer (victim cache).
- On a cache miss check victim cache for data before going to main memory.
- Jouppi [1990]: A 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache.
- Used in Alpha, HP PA-RISC machines.
Miss Rate Reduction Techniques:

Pseudo-Associative Cache

- Attempts to combine the fast hit time of Direct Mapped cache and have the lower conflict misses of 2-way set-associative cache.
- Divide cache in two parts: On a cache miss, check other half of cache to see if data is there, if so have a pseudo-hit (slow hit)
- Easiest way to implement is to invert the most significant bit of the index field to find other block in the “pseudo set”.

Drawback: CPU pipelining is hard to implement effectively if L₁ cache hit takes 1 or 2 cycles.
  - Better used for caches not tied directly to CPU (L₂ cache).
  - Used in MIPS R1000 L₂ cache, also similar L₂ in UltraSPARC.
**Miss Rate Reduction Techniques:**

**Hardware Prefetching of Instructions And Data**

- Prefetch instructions and data before they are needed by the CPU either into cache or into an external buffer.

- **Example:** The Alpha APX 21064 fetches two blocks on a miss: The requested block into cache and the next consecutive block in an instruction stream buffer.

- The same concept is applicable to data accesses using a data buffer.

- Extended to use multiple data stream buffers prefetching at different addresses (four streams improve data hit rate by 43%).

- It has been shown that, in some cases, eight stream buffers that can handle data or instructions can capture 50-70% of all misses.
Miss Rate Reduction Techniques:

Compiler Optimizations

Compiler cache optimizations improve access locality characteristics of the generated code and include:

- **Reorder procedures** in memory to reduce conflict misses.
- **Merging Arrays**: Improve spatial locality by single array of compound elements vs. 2 arrays.
- **Loop Interchange**: Change nesting of loops to access data in the order stored in memory.
- **Loop Fusion**: Combine 2 or more independent loops that have the same looping and some variables overlap.
- **Blocking**: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows.
Miss Rate Reduction Techniques: Compiler-Based Cache Optimizations

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];

Merging the two arrays:
– Reduces conflicts between val and key
– Improve spatial locality
Miss Rate Reduction Techniques: Compiler-Based Cache Optimizations

Loop Interchange Example

/* 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 in this case improves spatial locality.
Miss Rate Reduction Techniques: Compiler-Based Cache Optimizations

Loop Fusion Example

/* 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];
        }

- Two misses per access to \(a\) & \(c\) versus one miss per access
- Improves spatial locality
Miss Rate Reduction Techniques: Compiler-Based Cache Optimizations

Data Access 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 z[ ]
  – Read N elements of 1 row of y[ ] repeatedly
  – Write N elements of 1 row of x[ ]

• Capacity Misses can be represented as a function of N & Cache Size:
  – 3 NxNx4 => no capacity misses; otherwise ...

• Idea: compute BxB submatrix that fits in cache
Miss Rate Reduction Techniques: Compiler-Based Cache Optimizations

Blocking Example (continued)

/* 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 is called the Blocking Factor
• Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
• May also affect conflict misses
Compiler-Based Cache Optimizations

<table>
<thead>
<tr>
<th>Function</th>
<th>Performance Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpenta (nasa7)</td>
<td>2.5</td>
</tr>
<tr>
<td>gmtty (nasa7)</td>
<td>2.0</td>
</tr>
<tr>
<td>tomcatv</td>
<td>1.5</td>
</tr>
<tr>
<td>btrix (nasa7)</td>
<td>1.5</td>
</tr>
<tr>
<td>mxm (nasa7)</td>
<td>1.0</td>
</tr>
<tr>
<td>spice</td>
<td>1.0</td>
</tr>
<tr>
<td>cholesky (nasa7)</td>
<td>1.0</td>
</tr>
<tr>
<td>compress</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Legend:
- **merged arrays**
- **loop interchange**
- **loop fusion**
- **blocking**
Miss Penalty Reduction Techniques:  
Giving Priority To Read Misses Over Writes

- Write-through cache with write buffers suffers from RAW conflicts with main memory reads on cache misses:
  - Write buffer holds updated data needed for the read.
  - One solution is to simply wait for the write buffer to empty, increasing read miss penalty (in old MIPS 1000 by 50%).
  - Check write buffer contents before a read; if no conflicts, let the memory access continue.

- For write-back cache, on a read miss replacing dirty block:
  - Normally: Write dirty block to memory, and then do the read.
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write.
  - CPU stalls less since it restarts soon after the read.
Miss Penalty Reduction Techniques:

Sub-Block Placement

- Divide a cache block frame into a number of sub-blocks.
- Include a valid bit per sub-block of cache block frame to indicate validity of sub-block.
  - Originally used to reduce tag storage (fewer block frames).
- No need to load a full block on a miss just the needed sub-block.
Miss Penalty Reduction Techniques:

Early Restart and Critical Word First

• Don’t wait for full block to be loaded before restarting CPU:
  – *Early restart:* As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

  – *Critical Word First:* Request the missed word first from memory and send it to the CPU as soon as it arrives.
    • Let the CPU continue execution while filling the rest of the words in the block.
    • Also called *wrapped fetch* and *requested word first*.

• Generally useful only for caches with large block sizes.

• Programs with a high degree of spatial locality tend to require a number of sequential word, and may not benefit by early restart.
Miss Penalty Reduction Techniques:

Non-Blocking Caches

*Non-blocking cache* or *lockup-free cache* allows data cache to continue to supply cache hits during the processing of a miss:

- Requires an out-of-order execution CPU.
- “hit under miss” reduces the effective miss penalty by working during misses vs. ignoring CPU requests.
- “hit under multiple miss” or “miss under miss” may further lower the effective miss penalty by overlapping multiple misses.
- Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses.
- Requires multiple memory banks to allow multiple memory access requests.
- Example: Intel Pentium Pro/III allows up to 4 outstanding memory misses.
Value of Hit Under Miss For SPEC

Average Memory Access Time (A.M.A.T)

Hit Under i Misses

0->1
1->2
2->64
Base

EECC551 - Shaaban
Cache Miss Penalty Reduction Techniques:
Second-Level Cache (L₂)

- By adding another cache level between the original cache and memory:

  1. The first level of cache (L₁) can be small enough to be placed on-chip to match the CPU clock rate.
  2. The second level of cache (L₂) is large enough to capture a large percentage of accesses.

- When adding a second level of cache:

  Average memory access time = Hit time\textsubscript{L₁} + Miss rate\textsubscript{L₁} \times Miss penalty\textsubscript{L₁}

  where:

  Miss penalty\textsubscript{L₁} = Hit time\textsubscript{L₂} + Miss rate\textsubscript{L₂} \times Miss penalty\textsubscript{L₂}

- Local miss rate: the number of misses in the cache divided by the total number of accesses to this cache (i.e. Miss rate\textsubscript{L₂} above).
- Global miss rate: The number of misses in the cache divided by the total accesses by the CPU (i.e. the global miss rate for the second level cache is Miss rate\textsubscript{L₁} \times Miss rate\textsubscript{L₂}

Example:

Given 1000 memory references 40 misses occurred in L₁ and 20 misses in L₂

The miss rate for L₁ (local or global) = 40/1000 = 4%

The global miss rate for L₂ = 20/1000 = 2%
L₂ Performance Equations

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

\[
\text{Miss Penalty}_{L1} = \begin{array}{c}
\text{Hit Time}_{L2} + \\
\text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}
\end{array}
\]

\[
\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \left(\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}\right)
\]
Cache Miss Penalty Reduction Techniques:

3 Levels of Cache, L₁, L₂, L₃

- CPU
- L₁ Cache: Hit Rate = H₁, Hit time = 1 cycle
- L₂ Cache: Hit Rate = H₂, Hit time = T₂ cycles
- L₃ Cache: Hit Rate = H₃, Hit time = T₃ cycles

Main Memory

Memory access penalty, M
L3 Performance Equations

$$AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

$$\text{Miss Penalty}_{L2} = \text{Hit Time}_{L3} + \text{Miss Rate}_{L3} \times \text{Miss Penalty}_{L3}$$

$$AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times (\text{Hit Time}_{L3} + \text{Miss Rate}_{L3} \times \text{Miss Penalty}_{L3}))$$
Hit Time Reduction Techniques:

**Pipelined Writes**

- Pipeline tag check and cache update as separate stages; current write tag check & previous write cache update
- Only STORES in the pipeline; empty during a miss

```
Store r2, (r1) Check r1
Add --
Sub --
Store r4, (r3) M[r1]<-
    r2& check r3
```

- Shaded is “Delayed Write Buffer”; which must be checked on reads; either complete write or read from buffer
Hit Time Reduction Techniques:

Avoiding Address Translation

• Send virtual address to cache: Called *Virtually Addressed Cache* or just *Virtual Cache* vs. *Physical Cache*
  – Every time process is switched logically the cache must be flushed; otherwise it will return false hits
  • Cost is time to flush + “compulsory” misses from empty cache
  – Dealing with *aliases* (sometimes called *synonyms*); Two different virtual addresses map to same physical address
  – I/O must interact with cache, so need virtual address

• Solution to aliases:
  – HW guarantees covers index field & direct mapped, they must be unique; this is called *page coloring*

• Solution to cache flush:
  – Add *process identifier tag* that identifies a process as well as address within process: can’t get a hit if wrong process
Hit Time Reduction Techniques:

Virtually Addressed Caches

Conventional Organization

Virtually Addressed Cache
Translate only on miss
Synonym Problem

Overlap $ access with VA translation:
requires $ index to remain invariant across translation
## Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>MR</th>
<th>MP</th>
<th>HT</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Block Size</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>1</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>2</td>
</tr>
<tr>
<td>Pseudo-Associative Caches</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>2</td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>2</td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>3</td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>1</td>
</tr>
<tr>
<td>Subblock Placement</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>1</td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>2</td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>3</td>
</tr>
<tr>
<td>Second Level Caches</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>2</td>
</tr>
<tr>
<td>Small &amp; Simple Caches</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Avoiding Address Translation</td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>2</td>
</tr>
<tr>
<td>Pipelining Writes</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>1</td>
</tr>
</tbody>
</table>

**Miss rate**

**Miss Penalty**

**Hit time**