Cache Optimizations

COE 501
Computer Architecture
Prof. Muhamed Mudawar

Computer Engineering Department
King Fahd University of Petroleum and Minerals
Improving Cache Performance

- Software Optimizations to reduce Miss Rate

- Hardware Cache Optimizations
Improving Cache Performance

- **Average Memory Access Time (AMAT)**
  \[ AMAT = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \]

- **Used as a framework for optimizations**

- **Reduce the Hit time**
  - Small and simple caches

- **Reduce the Miss Rate**
  - Larger block size, Larger cache size, and Higher associativity

- **Reduce the Miss Penalty**
  - Multilevel caches, and giving reads priority over writes
Small and Simple Caches

- Reduce Hit time and Energy consumption

- Hit time is critical: affects the processor clock cycle
  - Indexing a cache represents a time consuming portion
  - Tag comparison in the tag array (hit or miss)
  - Selecting the data (way) in set-associative cache

- Direct-mapped overlaps tag check with data transfer
  - Associative cache uses additional mux and increases hit time

- Size of L1 caches has not increased much
  - I-Cache and D-Cache are about 64KB in recent processors
Access Time vs Size/Associativity

CACTI, 40 nm technology, Single Bank, 64-Byte blocks

Results depend on technology and detailed design assumptions.
Energy Consumption Per Read

CACTI, 40 nm technology, 64-Byte blocks

Tags + Data are read in parallel.

Energy per read is higher for multi-way set-associative caches
Classifying Cache Misses - Three Cs

- **Conditions under which cache misses occur**

  - **Compulsory**: program starts with no block in cache
    - Also called *cold start misses* or *first-reference misses*
    - Misses that would occur even if a cache has infinite size

  - **Capacity**: misses happen because cache size is small
    - Blocks are replaced and then later retrieved
    - Misses that would occur even if cache is fully associative

  - **Conflict**: misses happen because of limited associativity
    - Limited number of blocks per set and non-optimal replacement

- **4th C: Coherence misses** (discussed later)
Classifying Cache Misses

Compulsory misses are independent of cache size

Very small for long-running programs

Capacity misses decrease as capacity increases

Data were collected using LRU replacement

Conflict misses decrease as associativity increases

Miss Rate

1-way
2-way
4-way
8-way
Capacity
Compulsory

Miss Rate

0 2% 4% 6% 8% 10% 12% 14%

Cache Size

1 2 4 8 16 32 64 128 KB
Larger Block to Reduce Miss Rate

- Simplest way to reduce miss rate is to increase block size
- Large block size takes advantage of spatial locality

![Graph showing the relationship between block size and miss rate]

- Increased Conflict Misses
- Reduced Compulsory Misses

64-byte blocks are common in caches
Block Size Impact on AMAT

- Given: miss rates for different cache sizes & block sizes
- Memory latency = 80 cycles + 1 cycle per 8 bytes
  - Latency of 16-byte block = 80 + 2 = 82 clock cycles
  - Latency of 32-byte block = 80 + 4 = 84 clock cycles
  - Latency of 256-byte block = 80 + 32 = 112 clock cycles
- Which block has smallest AMAT for each cache size?

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Cache = 4 KB</th>
<th>Cache = 16 KB</th>
<th>Cache = 64 KB</th>
<th>Cache = 256 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>8.57%</td>
<td>3.94%</td>
<td>2.04%</td>
<td>1.09%</td>
</tr>
<tr>
<td>32 bytes</td>
<td>7.24%</td>
<td>2.87%</td>
<td>1.35%</td>
<td>0.70%</td>
</tr>
<tr>
<td>64 bytes</td>
<td>7.00%</td>
<td>2.64%</td>
<td>1.06%</td>
<td>0.51%</td>
</tr>
<tr>
<td>128 bytes</td>
<td>7.78%</td>
<td>2.77%</td>
<td>1.02%</td>
<td>0.49%</td>
</tr>
<tr>
<td>256 bytes</td>
<td>9.51%</td>
<td>3.92%</td>
<td>1.15%</td>
<td>0.49%</td>
</tr>
</tbody>
</table>
Block Size Impact on AMAT

- **Solution:** assume hit time = 1 clock cycle
  - Regardless of block size and cache size

- **Cache Size = 4 KB, Block Size = 16 bytes**
  - AMAT = $1 + 8.57\% \times 82 = 8.027$ clock cycles

- **Cache Size = 256 KB, Block Size = 256 bytes**
  - AMAT = $1 + 0.49\% \times 112 = 1.549$ clock cycles

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Cache = 4 KB</th>
<th>Cache = 16 KB</th>
<th>Cache = 64 KB</th>
<th>Cache = 256 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>AMAT = 8.027</td>
<td>AMAT = 4.231</td>
<td>AMAT = 2.673</td>
<td>AMAT = 1.894</td>
</tr>
<tr>
<td>32 bytes</td>
<td>AMAT = <strong>7.082</strong></td>
<td>AMAT = 3.411</td>
<td>AMAT = 2.134</td>
<td>AMAT = 1.588</td>
</tr>
<tr>
<td>64 bytes</td>
<td>AMAT = 7.160</td>
<td>AMAT = <strong>3.323</strong></td>
<td>AMAT = 1.933</td>
<td>AMAT = <strong>1.449</strong></td>
</tr>
<tr>
<td>128 bytes</td>
<td>AMAT = 8.469</td>
<td>AMAT = 3.659</td>
<td>AMAT = 1.979</td>
<td>AMAT = 1.470</td>
</tr>
<tr>
<td>256 bytes</td>
<td>AMAT = 11.65</td>
<td>AMAT = 4.685</td>
<td>AMAT = 2.288</td>
<td>AMAT = 1.549</td>
</tr>
</tbody>
</table>
Larger Cache & Higher Associativity

- Increasing cache size reduces capacity misses
- It also reduces conflict misses
  - Larger cache size spreads out references to more blocks
- Drawback: longer hit time and higher cost
- Higher associativity also improves miss rates
  - Eight-way set associative is as effective as a fully associative
- Drawback: longer hit time and more energy to access
- Larger caches are popular as 2nd and 3rd level caches
 Improving Cache Performance

 Software Optimizations to reduce Miss Rate

 Hardware Cache Optimizations
Software Optimizations

- Can be done by the programmer or optimizing compiler

- Restructuring code affects data access
  - Improves spatial locality
  - Improves temporal locality

- Three optimizations
  1. Loop Interchange
  2. Loop Fusion
  3. Blocking (also called Tiling)

- In addition, software prefetching helps streaming data
  - Prefetch array data in advance to eliminate cache misses
Loop Interchange

Modern compilers optimize loops to reduce cache misses

// Original Code
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    x[i][j] = 2 * y[i][j]; // stride = N

Original code traverses matrix by column

// After Loop Interchange
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    x[i][j] = 2 * y[i][j]; // stride = 1

Revised version takes advantage of spatial locality
// Original Code
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];

for (i = 0; i < N; i++)
    d[i] = a[i] + b[i] * c[i];

Blocks are replaced in first loop then accessed in second

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

Revised version takes advantage of **temporal locality**
Blocking (or Tiling)

Original code deals with multiple matrices

Matrix Y is accessed by row, while Z is accessed by column

Loop interchange does not help

```c
// Original Code for Matrix Multiplication
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        sum = 0;
        for (k = 0; k < N; k++) {
            sum = sum + y[i][k] * z[k][j];
        }
        x[i][j] = sum;
    }
```
Access Pattern for Matrix Multiply

Matrix X is accessed by row. Exploits Spatial locality.

Matrix Y is accessed by row. Rows are reused. If large N then row blocks are replaced \( \Rightarrow \) cache misses.

Matrix Z accessed by column. No spatial locality. Matrix Z is reused. However, blocks are replaced \( \Rightarrow \) misses.
Restructuring Code with Blocking

// Blocking or Tiling (B = Block Size)
for (jj = 0; jj < N; jj = jj + B) {
    for (kk = 0; kk < N; kk = kk + B) {
        for (i = 0; i < N; i++)
            for (j = jj; j < min(jj+B,N); j++) {
                sum = 0;
                for (k = kk; k < min(kk+B,N); k++) {
                    sum = sum + y[i][k] * z[k][j];
                }
                x[i][j] = x[i][j] + sum;
            }
    }
}

Matrix X should be initialized to zero

Block size is chosen such that blocks can fit in D-Cache
Access Pattern with Blocking

Sub-row of Matrix Y (consisting of B elements) is multiplied by a sub-block of Matrix Z (consisting of $B \times B$ elements) to compute (partially) a sub-row of Matrix X.

Exploits **spatial and temporal** localities in X, Y, and Z.
Compiler-Controlled Prefetching

- Cache prefetch: load data into the cache only
- Processor offers non-faulting cache prefetch instruction
- Overlap execution with the prefetching of data
- Goal is to hide the miss penalty & reduce cache misses
- Example:

  ```c
  for (i=0; i<N; i++) {
      prefetch(&a[i+P]);
      prefetch(&b[i+P]);
      sum = sum + a[i] * b[i];
  }
  ```

- How to estimate P?
  Cost of Prefetch Instructions?
- Can prefetching be done by hardware transparently?
Improving Cache Performance

Software Optimizations to reduce Miss Rate

Hardware Cache Optimizations
Hardware Cache Optimizations

Five hardware cache optimizations are considered:

1. Priority to Cache Read Misses over Writes
2. Hardware Prefetching of Instructions and Data
3. Pipelined Cache Access
4. Non-Blocking Caches
5. Multi-Ported and Multi-Banked Caches
Priority to Read Misses over Writes

- Reduces: Miss Penalty
- Serve read misses **before** writes have completed
- Write-Through Cache ➔ Write Buffer
  - Read miss is served before completing writes in write buffer
  - Problem: write buffer might hold updated data on a read miss
    - Solution: lookup write buffer and forward data (if buffer hit)
- Write-Back Cache ➔ Victim Buffer
  - Read miss is served before writing back modified blocks
  - Modified blocks that are evicted are moved into a victim buffer
  - Problem: victim buffer might hold block on a read miss
    - Solution: lookup victim buffer and forward block (if buffer hit)
Hardware Prefetching

- Hardware observes instruction and data access patterns
  - Prefetch instruction/data blocks before they are requested
- Prefetch two blocks on a cache miss (most common)
  - The requested block and the next consecutive block
  - The requested block is placed in the cache
  - The prefetched block is placed into a stream buffer
- If the requested block is present in the stream buffer
  - Read block from the stream buffer & issue next prefetch request
- Multiple stream buffers for instruction & data prefetching
  - Prefetching utilizes memory bandwidth and consumes energy
  - If prefetched data is not used ➔ negative impact on performance
Speedup due to Hardware Prefetching

Hardware Prefetching Turned ON

Copyright, Elsevier Inc. All rights reserved.
Pipelined Cache Access

- Used mainly in the L1 Instruction and Data caches
- L1 cache latency is multiple clock cycles (2 to 4 cycles)
- However, L2 and L3 cache accesses are not pipelined

Advantages of Pipelined Cache Access

- Faster clock rate and higher bandwidth
- Better for larger associativity

Disadvantages

- Increases latency of I-Cache and D-Cache
- Increases branch penalty due to increased I-Cache latency
- Increases load delay due to increased D-Cache latency
Example of Pipelined Cache Access

Parallel Access to Tag and Data Array, Reduces Latency, Good for L1 cache
Serial Access to Tag and Data Arrays

- Tag array is examined first for hit, then only one way is accessed.

Serial Access to Tag and Data Array, Reduces Energy, Good for L2 and L3 caches.
Non-Blocking Cache

- Allows a cache to continue to supply hits under a miss
  - The processor need not stall on a cache miss
  - Useful for out-of-order execution and multithreaded processors

- Hit under a Miss
  - Reduces the effective miss penalty
  - Increases cache bandwidth

- Hit under Multiple Misses
  - Multiple outstanding cache misses
  - May further lower the effective miss penalty
  - Increases the complexity of the cache controller
  - Beneficial if the memory system can service multiple misses
Non-Blocking Cache Timeline

Blocking Cache

Execution Time

Miss Penalty

Hit Under 1 Miss

M H S

Execution Time

Miss Penalty

Miss Penalty

Hit Under 2 Misses

M M H S

Execution Time

Miss Penalty

Miss Penalty

Miss Penalty

M = Cache Miss = Stall

M = Cache Miss, H = Hit, S = Stall
Effectiveness of Non-Blocking Cache

Hit-under-1-miss reduces the miss penalty by 9% (SPECINT) and 12.5% (SPECFP)
Hit-under-2-misses reduces the miss penalty by 10% (SPECINT) and 16% (SPECFP)
Miss Status Holding Register (MSHR)

- Contains the block address of the pending miss
  - Same block can have multiple outstanding load/store misses
  - Can also have multiple outstanding block addresses

- Misses can be classified into:
  - Primary: first miss to a cache block that initiates a fetch request
  - Secondary: subsequent miss to a cache block in transition
  - Structural Stall miss: the MSHR hardware resource is fully utilized

\[
\begin{array}{|c|c|c|}
\hline
V & \text{Block address} & \text{New miss address} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
V & \text{Type} & \text{Offset} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
V & \text{Type} & \text{Offset} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
V & \text{Type} & \text{Offset} \\
\hline
\end{array}
\]

- Type: LD, SD, LW, SW, etc.
- Offset: block offset
- Destination register for load or Data for store
Non-Blocking Cache Operation

- On Cache Miss, check MSHR for matched block address
  - If found: allocate new load/store entry for matched block
  - If not found: allocate new MSHR and load/store entry
  - If all MSHR resources are allocated then Stall (Structural)

- When cache block is transferred from lower-level memory
  - Process the load and store instructions that missed in the block
  - Load data from the specified block offset into destination register
  - Store data in the data cache at the specified block offset
  - De-allocate MSHR entry after completing all missed loads/stores
Multi-Banked Cache

- Banks were originally used in main memory and DRAM chips.
- They are now commonly used in cache memory (L1, L2, and L3).
- The cache is divided into multiple banks.
- Multiple banks can be accessed independently and in parallel.
- Intel core i7 has 4 banks in L1 and 8 banks in L2.
  - L1 cache banks can support 2 memory accesses per cycle.
    - To support high instruction execution rate in superscalar processors.
  - L2 cache banks can handle multiple outstanding L1 cache misses.
    - To support non-blocking caches.
  - L2 and L3 cache banks also reduce energy per access ➔ smaller arrays.
Multi-Banked Cache (cont'd)

Partition address space into multiple banks

- Block-interleaved cache banks

- Bank Address (BA) = Block Address $\mod N$ banks

- When two requests map to same cache bank $\Rightarrow$ Bank Conflict

- One request is allowed to proceed, while second request waits

Example: Sequential interleaving of blocks across 4 cache banks

- Each cache bank is implemented using a tag array and a data array

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Block 0, 16, ...</td>
<td>Block 1, 17, ...</td>
<td>Block 2, 18, ...</td>
</tr>
<tr>
<td>0</td>
<td>Block 4, 20, ...</td>
<td>Block 5, 21, ...</td>
<td>Block 6, 22, ...</td>
</tr>
<tr>
<td>1</td>
<td>Block 8, 24, ...</td>
<td>Block 9, 25, ...</td>
<td>Block 10, 26, ...</td>
</tr>
<tr>
<td>2</td>
<td>Block 12, 28, ...</td>
<td>Block 13, 29, ...</td>
<td>Block 14, 30, ...</td>
</tr>
</tbody>
</table>
Multi-Ported, Multi-Banked Cache

- Example: Dual-Ported Data Cache with four cache banks
  - Two address ports → Two load / store instructions per cycle
  - Four cache banks to reduce bank conflict
  - Each cache bank is set-associative with a tag array and data array
  - Crossbar switches map addresses to cache banks and back to the ports

Diagram:

Port0: Inst, Addr, Rd, Data_in

Port1: Inst, Addr, Rd, Data_in

2×4 switch

Cache Bank 0

Cache Bank 1

Cache Bank 2

Cache Bank 3

4×2 switch

Port0: Data_out, Rd

Port1: Data_out, Rd
In Summary

- **Reducing Hit Time and Energy**
  - Smaller and simpler L1 caches

- **Reducing Miss Rate**
  - Larger block size, larger capacity, and higher associativity
  - Software (and compiler) optimizations
  - Software and Hardware prefetching of instructions and data

- **Reducing Miss Penalty**
  - Multi-level caches
  - Priority to read misses over writes, non-blocking cache

- **Increasing Cache Bandwidth**
  - Pipelined, non-blocking, multi-ported, and multi-banked cache