Computer Structure

Cache Memory

Lihu Rappoport and Adi Yoaz
Processor – Memory Gap

Performance vs. Time

CPU Performance Gap:
(grows 50% / year)

DRAM Performance


Performance

Time
Memory Trade-Offs

- Large (dense) memories are slow
- Fast memories are small, expensive and consume high power
- Goal: give the processor a feeling that it has a memory which is large (dense), fast, consumes low power, and cheap
- Solution: a Hierarchy of memories

Speed: Fastest → Slowest
Size: Smallest → Biggest
Cost: Highest → Lowest
Power: Highest → Lowest
Why Hierarchy Works

- **Temporal Locality (Locality in Time):**
  - If an item is referenced, it will tend to be referenced again soon
  - Example: code and variables in loops
  ⇒ Keep recently accessed data closer to the processor

- **Spatial Locality (Locality in Space):**
  - If an item is referenced, nearby items tend to be referenced soon
  - Example: scanning an array
  ⇒ Move contiguous blocks closer to the processor

- Locality + smaller HW is faster + Amdahl’s law
  ⇒ memory hierarchy
Memory Hierarchy: Terminology

- For each memory level define the following
  - **Hit**: data available in the memory level
  - **Hit Rate**: the fraction of accesses found in that level
  - **Hit Time**: time from data request till data received when hitting in the memory level; includes also the time to determine hit/miss
  - **Miss**: data not present in memory level
  - **Miss Rate** = \(1 - \text{(Hit Rate)}\)
  - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block to the processor

- **Average memory-access time** = 
  \[
  t_{\text{effective}} = (\text{Hit time} \times \text{Hit Rate}) + (\text{Miss Time} \times \text{Miss rate})
  = (\text{Hit time} \times \text{Hit Rate}) + (\text{Miss Time} \times (1 - \text{Hit rate}))
  \]
  - If hit rate is close to 1, \(t_{\text{effective}}\) is close to Hit time
Effective Memory Access Time

- **Cache** – holds a subset of the memory
  - Hopefully – the subset being used now

- **Effective memory access time**
  - \( t_{\text{effective}} = (t_{\text{cache}} \times \text{Hit Rate}) + (t_{\text{mem}} \times (1 - \text{Hit rate})) \)
  - \( t_{\text{mem}} \) includes the time it takes to detect a cache miss

- **Example**
  - Assume \( t_{\text{cache}} = 10 \) nsec and \( t_{\text{mem}} = 100 \) nsec

<table>
<thead>
<tr>
<th>Hit Rate</th>
<th>( t_{\text{eff}} ) (nsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>90</td>
<td>20</td>
</tr>
<tr>
<td>99</td>
<td>10.9</td>
</tr>
<tr>
<td>99.9</td>
<td>10.1</td>
</tr>
</tbody>
</table>

- \( t_{\text{mem}}/t_{\text{cache}} \) goes up \( \Rightarrow \) more important that hit-rate closer to 1
The cache holds a small part of the entire memory
- Need to map parts of the memory into the cache

Main memory is (logically) partitioned into blocks
- Typical block size is 32 to 64 bytes
- Blocks are aligned

Cache partitioned to cache lines
- Each cache line holds a block
- Only a subset of the blocks is mapped to the cache at a given time
- The cache views an address as
Cache Lookup

- **Cache hit**
  - Block is mapped to the cache – return data according to block’s offset

- **Cache miss**
  - Block is not mapped to the cache ⇒ do a cache line fill
    - Fetch block into fill buffer
      - may require few bus cycle
    - Write fill buffer into cache
  - May need to evict another block from the cache
    - Make room for the new block
Fully Associative Cache

- An address is partitioned to:
  - offset within block
  - block number

- Each block may be mapped to each of the cache lines:
  - Lookup block in all lines

- Each cache line has a tag:
  - All tags are compared to the block# in parallel
  - Need a comparator per line
  - If one of the tags matches the block#, we have a hit
    - Supply data according to offset
Valid Bit

- Initially cache is empty
  - Need to have a “line valid” indication – line \textit{valid bit}

- A line may also be invalidated
Direct Map Cache

- `#block l.s.bits` map block to a specific cache line
  - A given block is mapped to a specific cache line
    - Also called *set*
  - If two blocks are mapped to the same line, only one can reside in the cache

- The rest of the `#block bits` are used as *tag*
  - Compared to the tag stored in the cache for the appropriate set
Line Size: 32 bytes ⇒ 5 Offset bits
Cache Size: 16KB = 2^{14} Bytes

#lines = cache size / line size
  = 2^{14}/2^5=2^9=512
#sets = #lines
#set bits = 9 bits

#Tag bits
  = 32 − (#set bits + #offset bits)
  = 32 − (9+5) = 18 bits

Lookup Address: 0x12345678
0001 0010 0011 0100 0101 0110 0111 1000
  tag= 0x048B1
  set= 0x0B3
  offset= 0x18
**Direct Map Cache (cont)**

- **Partition memory into slices**
  - slice size = cache size

- **Partition each slice to blocks**
  - Block size = cache line size
  - Distance of block from slice start indicates position in cache (set)

- **Advantages**
  - Easy hit/miss resolution
  - Easy replacement algorithm
  - Lowest power

- **Disadvantage**
  - Line replacements due to “set conflict misses”
The page illustrates a 2-Way Set Associative Cache with the following points:

- Each set holds two lines (way 0 and way 1).
- Each block can be mapped into one of two lines in the appropriate set.

**Example:**

- **Line Size:** 32 bytes
- **Cache Size:** 16KB
- **# of lines:** 512 lines
- **# sets:** 256
- **Offset bits:** 5 bits
- **Set bits:** 8 bits
- **Tag bits:** 19 bits

The address of 0x12345678 corresponds to:

```
0001 0010 0011 0100
0101 0110 0111 1000
```

- **Offset:** 11000 = 0x18 = 24
- **Set:** 1011 0011 = 0x0B3 = 179
- **Tag:** 000 1001 0001 1010 0010 = 0x091A2
### 2-Way Cache – Hit Decision

- **Tag**: 39
- **Set**: 12
- **Offset**: 5

- **Way 0**:
  - Tag
  - Set
  - Offset
- **Way 1**:
  - Tag
  - Set
  - Offset

- **Data**:
  - Way 0
  - Way 1

- **MUX**:

- **Hit/Miss**:

- **Data**:
2-Way Set Associative Cache (cont)

- **Partition memory into slices**
  - slice size = way size = ½ cache size

- **Partition each slice to blocks**
  - Block size = cache line size
  - Distance of block from slice start indicates position in cache (set)

- **Compared to direct map cache**
  - Half size slice ⇒ 2× number of slices
    ⇒ 2× number of blocks mapped to each set in the cache
  - But in each set we can have 2 blocks at a given time
Cache Read Miss

♦ On a read miss – perform a cache line fill
  ▶ Fetch entire block that contains the missing data from memory

♦ Block is fetched into the cache line fill buffer
  ▶ May take a few bus cycles to complete the fetch
    • e.g., 64 bit (8 byte) data bus, 32 byte cache line ⇒ 4 bus cycles
    • Can stream the critical chunk into the core before the line fill ends

♦ Once the entire block fetched into the fill buffer
  ▶ it is moved from the fill buffer into the cache
Cache Replacement

- **Direct map**
  - A new block is mapped to a single line in the cache
  - Old line is evicted (re-written to memory if needed)

- **N-way set associative cache**
  - Choose a victim from all ways in the appropriate set

- **Replacement algorithms**
  - Optimal replacement algorithm
  - FIFO – First In First Out
  - Random
  - LRU – Least Recently used

- **LRU is the best**
  - but not that much better even than random
LRU Implementation

- **2 ways**
  - 1 bit per set to mark latest way accessed in set
  - Evict way not pointed by bit

- **k-way set associative LRU**
  - Requires full ordering of way accesses
  - Hold a log₂k bit counter per line
  - When way \( i \) is accessed
    \[ X = \text{Counter}[i] \]
    \[ \text{Counter}[i] = k-1 \]
    for (\( j = 0 \) to \( k-1 \))
      \[ \text{if } ((j \neq i) \text{ AND } (\text{Counter}[j] > X)) \text{ Counter}[j]--; \]
  - When replacement is needed
    - evict way with counter = 0
  - Expensive for even small \( k \)'s

**Initial State**

<table>
<thead>
<tr>
<th>Way</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Access way 2**

<table>
<thead>
<tr>
<th>Way</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**Access way 0**

<table>
<thead>
<tr>
<th>Way</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Pseudo LRU (PLRU)

- PLRU records a partial order using a tree structure
  - The ways of a set are in the leaves; the PLRU bits are the internal nodes
    - For $n$ leaves (ways) there are $n-1$ internal nodes (PLRU bits)

- Example: 4-ways, using 3 bits per-set:
  - Bit$_0$ specifies if LRU in ways {0,1} or in ways {2,3}
  - Bit$_1$ specifies the LRU between {0,1}
  - Bit$_2$ specifies the LRU between {2,3}

- On each access to a way
  - Update PLRU bits from root to hit way to point to the way (0-right, 1-left)

- Example: ways access order: 3,0,2,1
  - bit$_0$=0 points to the pair with way 1
  - bit$_1$=1 points to way 1
  - bit$_2$=0 way 2 was accessed after way 3

- Victim selection
  - Follow the opposite directions pointed by the PLRU bits
Effect of Cache on Performance

- **MPI – miss per instruction**
  - MPI = \#cache misses / \#instructions
    = \#cache misses / \#mem access \times \#mem access / \#instructions
  - More correlative to performance than cache miss rate
    - Takes into account also frequency of memory accesses

- **Memory stall cycles**
  = \#memory accesses \times miss rate \times miss penalty
  = IC \times MPI \times miss penalty

- **CPU time**
  = (CPU execution cycles + Memory stall cycles) \times cycle time
  = IC \times (CPI_{execution} + MPI \times Miss penalty) \times cycle time
Memory Update Policy on Writes

- **Write back – cheaper writes**
  - “Store” operations that hit the cache write only to the cache
    - Main memory is not accessed in case of write hit
    - Line is marked as *modified* or *dirty*
  - Modified cache line written to memory only when it is evicted
    - Saves memory accesses when a line is updated many times
    - On eviction, the entire line must be written to memory
      - There is no indication which bytes within the line were modified

- **Write through – cheaper evictions**
  - Stores that hit the cache write both to cache and to memory
    - Need to write only the bytes that were changed (not entire line)
  - The ratio between reads and write is ~ 4:1
    - While writing to memory, the processor goes on reading from cache
  - No need to evict blocks from the cache (never dirty)
  - Use write buffers so that don’t wait for lower level memory
Write Buffer for Write Through

- A Write Buffer between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory

- Work ok if store frequency $\ll 1$/DRAM write cycle
  - Otherwise store buffer overflows no matter how big it is

- Write combining
  - combine writes in the write buffer

- On cache miss need to lookup write buffer
Cache Write Miss

- The processor is not waiting for data
  ⇒ continues its work

- Option 1:
  Write allocate: fetch the line into the cache
  - Goes with write back policy, assuming more writes are needed
  - hoping that subsequent writes to the line hit the cache

- Option 2:
  Write no allocate: do not fetch line into cache
  - Goes with write through policy
  - subsequent writes would update memory anyhow
Cache Line Size

- Larger line size takes advantage of spatial locality
  - Too big blocks: may fetch unused data
  - Possibly evicting useful data ⇒ miss rate goes up

- Larger line size means larger miss penalty
  - Takes longer time to perform a cache line fill
    - Using critical chunk first reduces the issue
  - Takes longer to evict (when using write back update policy)

\[
\text{Ave. Access Time} = \text{hit time} \times (1 - \text{miss rate}) + \text{miss penalty} \times \text{miss rate}
\]
Classifying Misses: 3 Cs

◆ Compulsory
  ❖ First access to a block which is not in the cache
    • the block must be brought into the cache
  ❖ Cache size does not matter
  ❖ Solution: prefetching

◆ Capacity
  ❖ Cache cannot contain all blocks needed during program execution
    • blocks are evicted and later retrieved
  ❖ Solution: increase cache size, stream buffers

◆ Conflict
  ❖ Occurs in set associative or direct mapped caches when too many blocks are mapped to the same set
  ❖ Solution: increase associativity, victim cache
3Cs in SPEC92

- **Conflict**
- **Capacity**

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>Miss Rate per Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.02</td>
</tr>
<tr>
<td>0.04</td>
<td></td>
</tr>
<tr>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>0.08</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>0.12</td>
<td></td>
</tr>
<tr>
<td>0.14</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2-way</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4-way</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>8-way</td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>
Victim Cache

- **Per-set pressure may be non-uniform**
  - Some sets may have more **conflict misses** than others
  - Solution: allocate ways to sets dynamically

- **Victim cache gives a 2\textsuperscript{nd} chance to evicted lines**
  - A line evicted from L1 cache is placed in the victim cache
  - If victim cache is full $\Rightarrow$ evict its LRU line
  - On L1 cache lookup, lookup victim cache in parallel

- **On victim cache hit**
  - Line is moved back to cache
  - Evicted line moved to the victim cache
  - Same access time as cache hit

- **Especially effective for direct mapped cache**
  - Enables to combine the fast hit time of a direct mapped cache and still reduce conflict misses
Stream Buffers

- Before inserting a new line into cache
  - Put new line in a stream buffer

- If the line is expected to be accessed again
  - Move the line from the stream buffer into cache
  - E.g., if the line hits in the stream buffer

- Example
  - Scanning a very large array (much larger than the cache)
  - Each item in the array is accessed just once
  - If the array elements are inserted into the cache
    - The entire cache will be thrashed
  - If we detect that this is just a scan-once operation
    - E.g., using a hint from the software
    - Can avoid putting the array lines into the cache
Prefetching

- **Hardware Data Prefetching** – predict future data accesses
  - Next sequential / Streaming
    - Triggered by an ascending access to very recently loaded data
    - Fetches the next line, assuming a streaming load
  - Stride prefetcher
    - Tracks individual load instructions, detecting a regular stride
    - Prefetch address = current address + stride
    - Detects strides of up to 2K bytes, both forward and backward
  - General pattern prefetcher
    - Identifies patterns of Load address distances

- **Data has to be prefetched before it is needed**
  - The prefetcher aggressiveness has to be tuned
  - How fast and how far ahead to issue the prefetch request, such that data arrives on time, before it is needed
Prefetching (cont.)

- Prefetching relies on extra memory bandwidth
  - Too aggressive / inaccurate prefetching slows down demand fetches
    - Hurts performance

- Software Prefetching
  - Special prefetching instructions that cannot cause faults

- Instruction Prefetching
  - On a cache miss, prefetch sequential cache lines into stream buffers
  - Branch predictor directed prefetching
    - Let branch predictor run ahead
Critical Word First

- Reduce Miss Penalty
- 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*
- *Example: Pentium*
  - 64 bit = 8 byte bus, 32 byte cache line \(\Rightarrow\) 4 bus cycles to fill line
  - Fetch date from 95H
    - \[\begin{array}{|c|c|c|c|}
      \hline
      80H-87H & 88H-8FH & 90H-97H & 98H-9FH \\
      \hline
      3 & 2 & 1 & 4 \\
      \hline
    \end{array}\]
Code Optimizations: Merging Arrays

- Merge 2 arrays into a single array of compound elements

/* Before: two sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: One array of structures */
struct merge {
    int val;
    int key;
} merged_array[SIZE];

- Reduce conflicts between val and key
- Improves spatial locality
Code Optimizations: Loop Fusion

- Combine 2 independent loops that have same looping and some variables overlap
- Assume each element in `a` is 4 bytes, 32KB cache, 32 B / line

```java
for (i = 0; i < 10000; i++)
    a[i] = 1 / a[i];
for (i = 0; i < 10000; i++)
    sum = sum + a[i];
```

- First loop: hit 7/8 of iterations
- Second loop: array > cache ⇒ same hit rate as in 1st loop

- Fuse the loops

```java
for (i = 0; i < 10000; i++) {
    a[i] = 1 / a[i];
    sum = sum + a[i];
}
```

- First line: hit 7/8 of iterations
- Second line: hit all

In the example only load accesses are taken into account for hit rate calculation
Code Optimizations: Loop Interchange

- 2-dimension array in memory:
  
  \[
  x[0][0] \ x[0][1] \ldots \ x[0][99] \ x[1][0] \ x[1][1] \ldots
  \]

/* Original program: access are 100 bytes apparatus */
/* \ x[0][0] \ x[1][0] \ldots \ x[4999][0] \ x[0][1] \ x[1][1] \ldots */
for (j = 0; j < 100; j++)
  for (i = 0; i < 5000; i++)
    \[x[i][j] = 2 \times x[i][j];\]

/* Reversing the loops order: access are adjacent */
/* Improved spatial locality */
for (i = 0; i < 5000; i++)
  for (j = 0; j < 100; j++)
    \[x[i][j] = 2 \times x[i][j];\]
Improving Cache Performance

- **Reduce cache miss rate**
  - Larger cache
  - Reduce compulsory misses
    - Larger Block Size
    - HW Prefetching (Instr, Data)
    - SW Prefetching (Data)
  - Reduce conflict misses
    - Higher Associativity
    - Victim Cache
  - Stream buffers
    - Reduce cache thrashing
  - Compiler Optimizations

- **Reduce the miss penalty**
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit under Miss, Miss under Miss)
  - Second Level Cache

- **Reduce cache hit time**
  - On-chip caches
  - Smaller size cache (hit time increases with cache size)
  - Direct map cache (hit time increases with associativity)
Separate Code / Data Caches

- Parallelize data access and instruction fetch

- Code cache is a read only cache
  - No need to write back line into memory when evicted
  - Simpler to manage

- Self modifying code
  - Whenever executing a memory write ⇒ snoop code cache
    - Requires a dedicated snoop port: tag array read + tag match
    - Otherwise snoops would stall fetch
  - If the code cache contains the written address
    - Invalidate the line in which the address is contained
    - Flush the pipeline – it main contain stale code
Non-Blocking Cache

◆ Hit Under Miss
  ◆ Allow cache hits while one miss is in progress
  ◆ Another miss has to wait
  ◆ Relevant for an out-of-order execution CPU

◆ Miss Under Miss, Hit Under Multiple Misses
  ◆ Allow hits and misses when other misses in progress
  ◆ Memory system must allow multiple pending requests
  ◆ Manage a list of outstanding cache misses
    • When miss is served and data gets back, update list
Multi-ported Cache

- N-ported cache enables $n$ accesses in parallel
  - Parallelize cache access in different pipeline stages
  - Parallelize cache access in a super-scalar processors

- About doubles the cache die size

- Possible solution: banked cache
  - Each line is divided to $n$ banks
  - Can fetch data from $k \leq n$ different banks in possibly different lines
L2 Cache

- L2 cache is bigger, but with higher latency
  - Reduces L1 miss penalty – saves access to memory
  - L2 cache is located on-chip, but may be shared by a few cores
  - L2 is unified (code / data)

- L2 inclusive of L1
  - All addresses in L1 are also contained in L2
    - Address evicted from L2 ⇒ snoop invalidate it in L1
  - Data in L1 may be more updated than in L2
    - When evicting a modified line from L1 ⇒ write to L2
    - When evicting a line from L2 which is modified in L1
      - Snoop invalidate to L1 generates a write from L1 to L2
      - Line marked as modified in L2 ⇒ Line written to memory
  - Since L2 contains L1 it needs to be significantly larger
    - e.g., if L2 is only 2× L1, half of L2 is duplicated in L1
Core™ 2 Duo Die Photo

L2 Cache
A memory system is *coherent* if

1. If P1 writes to address X, and later on P2 reads X, and there are no other writes to X in between
   \[ \Rightarrow \text{P2's read returns the value written by P1's write} \]

2. Writes to the same location are serialized:
   two writes to location X are seen in the same order by all processors

All processors see the same order of values. E.g., if P1 sees 0 then 1, P2 cannot see 1 then 0.
MESI Protocol

- Each cache line can be in one of 4 states

  - Invalid – Line’s data is not valid
  - Shared – Line is valid and not dirty, copies may exist in other processors
  - Exclusive – Line is valid and not dirty, other processors do not have the line in their local caches
  - Modified – Line is valid and dirty, other processors do not have the line in their local caches
Multi-processor System: Example

- P1 reads 1000
- P1 writes 1000
Multi-processor System: Example

- P1 reads 1000
- P1 writes 1000
- P2 reads 1000
- L2 snoops 1000
- P1 writes back 1000
- P2 gets 1000
Multi-processor System: Example

- P1 reads 1000
- P1 writes 1000
- P2 reads 1000
- L2 snoops 1000
- P1 writes back 1000
- P2 gets 1000
- P2 requests for ownership with write intent
Multi-processor System: Inclusion

- L2 needs to keep track of the presence of each line in each of the processors
  - Determine if it needs to send a snoop to a processor
  - Determine in what state to provide a requested line (S,E)
  - Need to guarantee that the L1 caches in each processor are inclusive of the L2 cache

- When L2 evicts a line
  - L2 sends a snoop invalidate to all processors that have it
  - If the line is modified in the L1 cache of one of the processors (in which case it exist only in that processor)
    - The processor responds by sending the updated value to L2
    - When the line is evicted from L2, the updated value gets written to memory
## MESI Protocol States

<table>
<thead>
<tr>
<th>State</th>
<th>Valid</th>
<th>Modified</th>
<th>Copies may exist in other processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>No</td>
<td>N.A.</td>
<td>N.A</td>
</tr>
<tr>
<td>Shared</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- **A modified line must be exclusive**
  - Otherwise, another processor which has the line will be using stale data
  - Therefore, before modifying a line, a processor must request ownership of the line
Backup
## Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Miss Rate</th>
<th>Miss Penalty</th>
<th>Hit Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Block Size</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pseudo-Associative Caches</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-block Placement</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Second Level Caches</td>
<td>+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table above summarizes various cache optimization techniques and their impact on miss rate, miss penalty, and hit time. A '+' indicates a positive effect, while a '-' indicates a negative effect.