Cache Optimizations

COE 501

Computer Architecture

Prof. Muhamed Mudawar

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Presentation Outline

- **Improving Cache Performance**
- Software Optimizations to reduce Miss Rate
- Hardware Cache Optimizations

Improving Cache Performance

- Average Memory Access Time (AMAT)

  AMAT = Hit time + Miss rate × Miss penalty

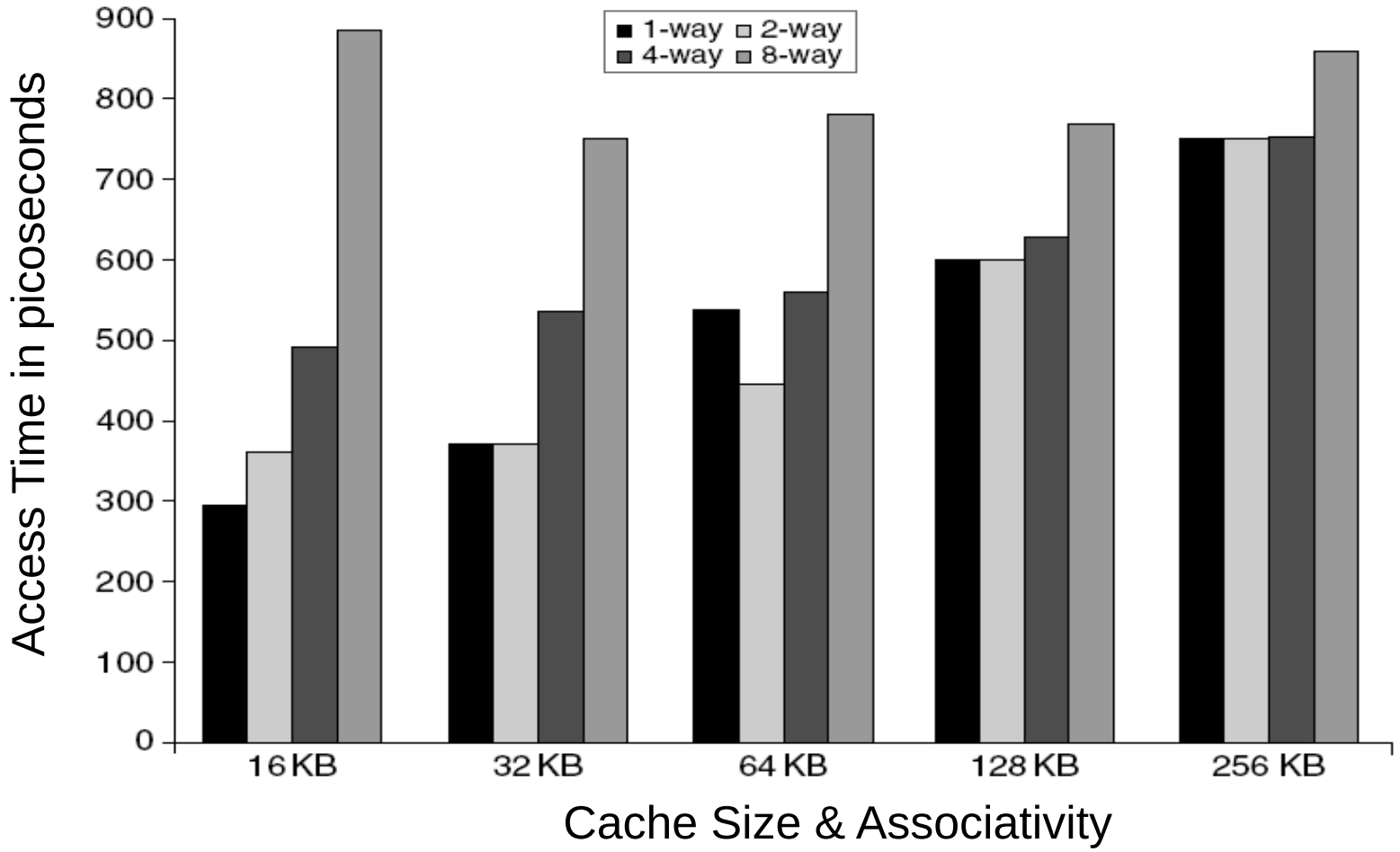
- Used as a framework for optimizations
- Reduce the Hit time
  - Small and simple caches
- Reduce the Miss Rate
  - Larger block size, Larger cache size, and Higher associativity
- Reduce the Miss Penalty
  - Multilevel caches, and giving reads priority over writes

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

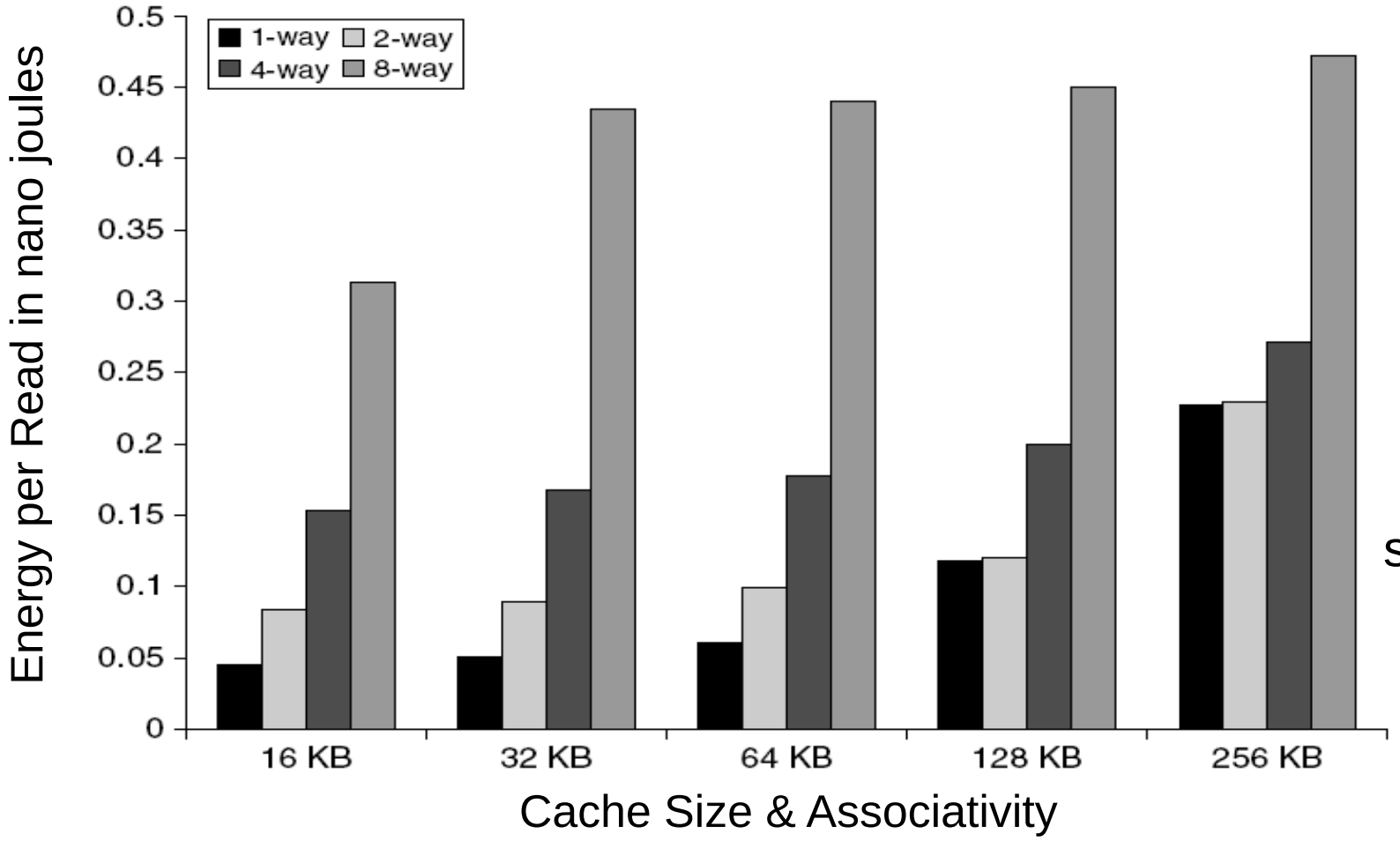
CACTI, 40 nm technology, Single Bank, 64-Byte blocks



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

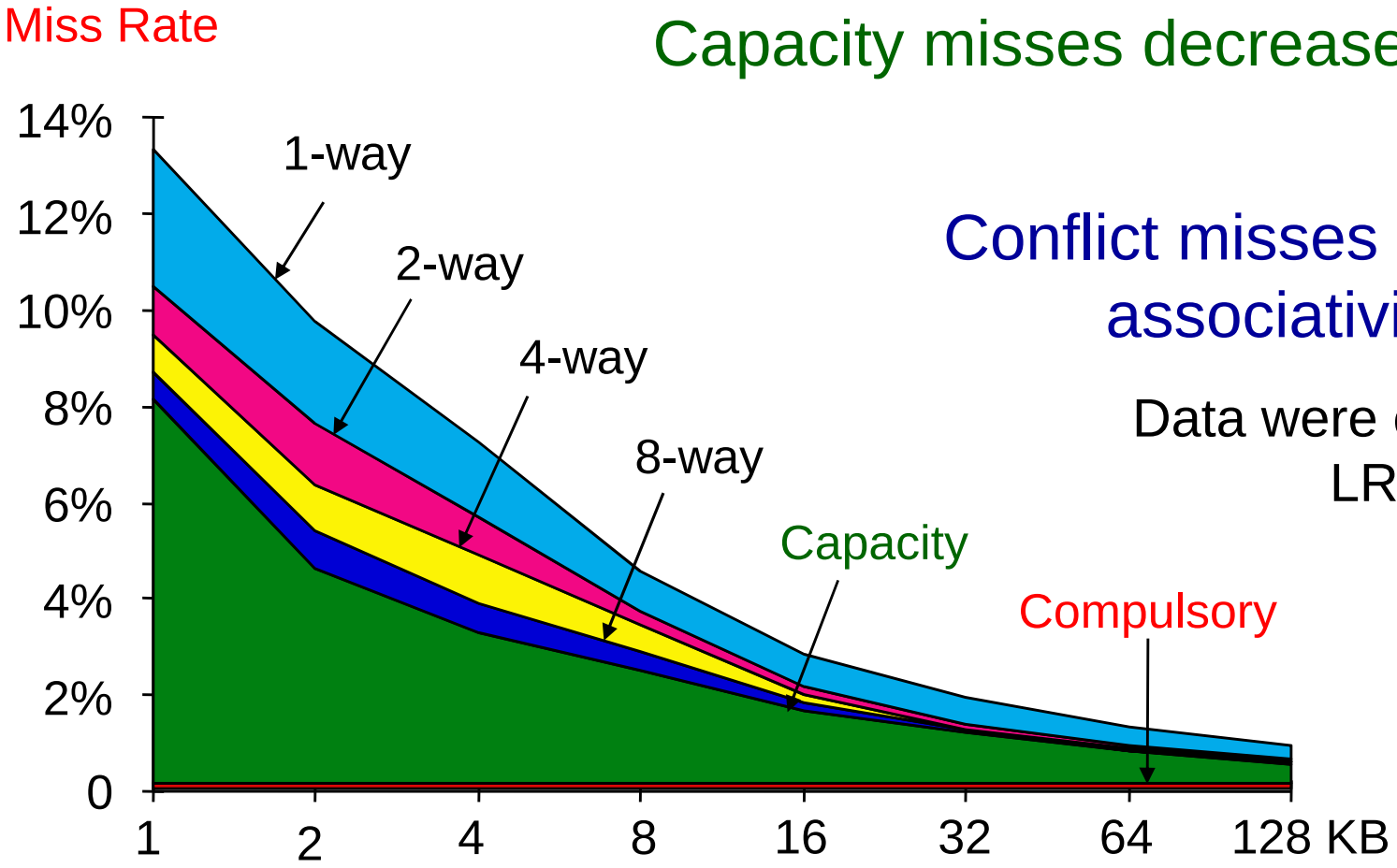
Compulsory misses are independent of cache size

Very small for long-running programs

Capacity misses decrease as capacity increases

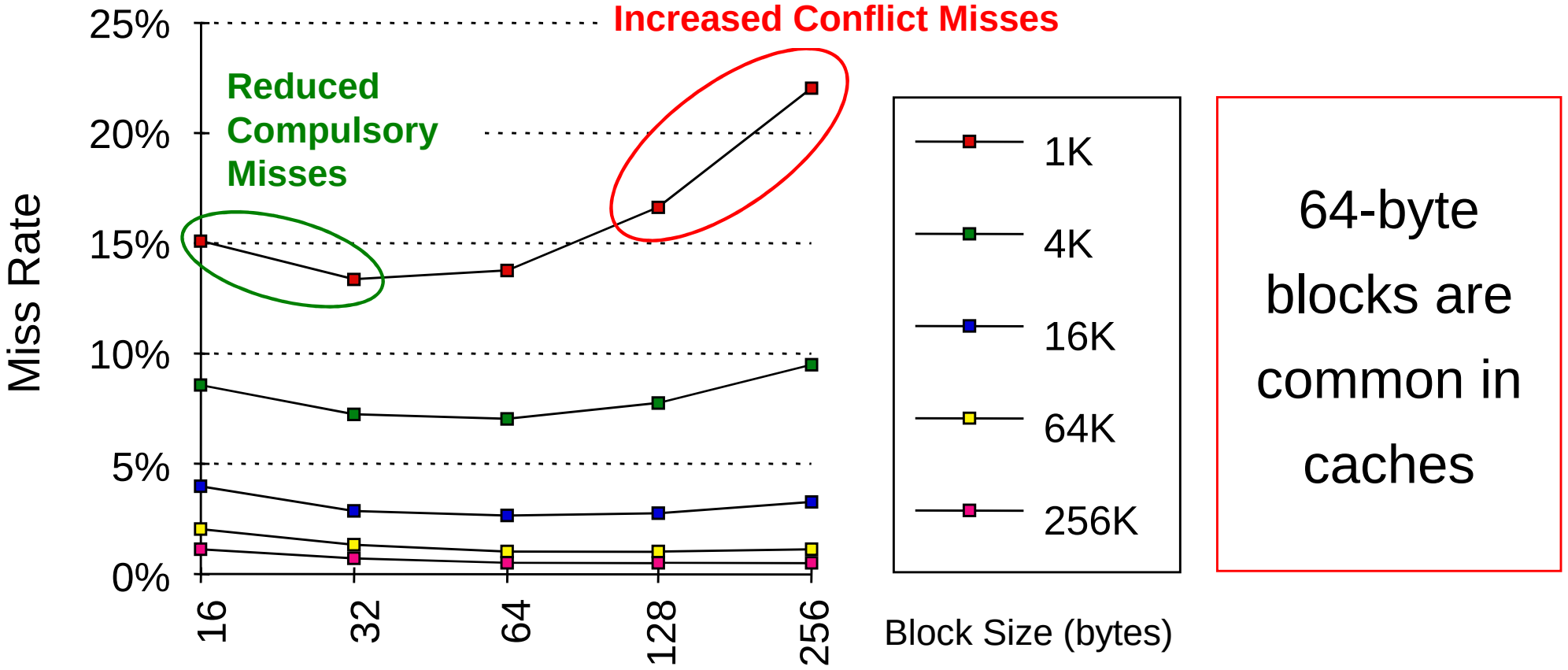
Conflict misses decrease as associativity increases

Data were collected using LRU replacement



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



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?

Block Size	Cache = 4 KB	Cache = 16 KB	Cache = 64 KB	Cache = 256 KB
16 bytes	8.57%	3.94%	2.04%	1.09%
32 bytes	7.24%	2.87%	1.35%	0.70%
64 bytes	7.00%	2.64%	1.06%	0.51%
128 bytes	7.78%	2.77%	1.02%	0.49%
256 bytes	9.51%	3.92%	1.15%	0.49%

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% × 82 = 8.027 clock cycles
- Cache Size = 256 KB, Block Size = 256 bytes
  - AMAT = 1 + 0.49% × 112 = 1.549 clock cycles

Block Size	Cache = 4 KB	Cache = 16 KB	Cache = 64 KB	Cache = 256 KB
16 bytes	AMAT = 8.027	AMAT = 4.231	AMAT = 2.673	AMAT = 1.894
32 bytes	AMAT = **7.082**	AMAT = 3.411	AMAT = 2.134	AMAT = 1.588
64 bytes	AMAT = 7.160	AMAT = **3.323**	AMAT = **1.933**	AMAT = **1.449**
128 bytes	AMAT = 8.469	AMAT = 3.659	AMAT = 1.979	AMAT = 1.470
256 bytes	AMAT = 11.65	AMAT = 4.685	AMAT = 2.288	AMAT = 1.549

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 2^{nd} and 3^{rd} level caches

Next . . .

- 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**

Loop Fusion

```
// 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

```
// 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

`x[i][j]` = `y[i][k]` × `z[k][j]`

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 ➔ cache misses.

Matrix Z accessed by column. No spatial locality. Matrix Z is reused. However, blocks are replaced ➔ misses.

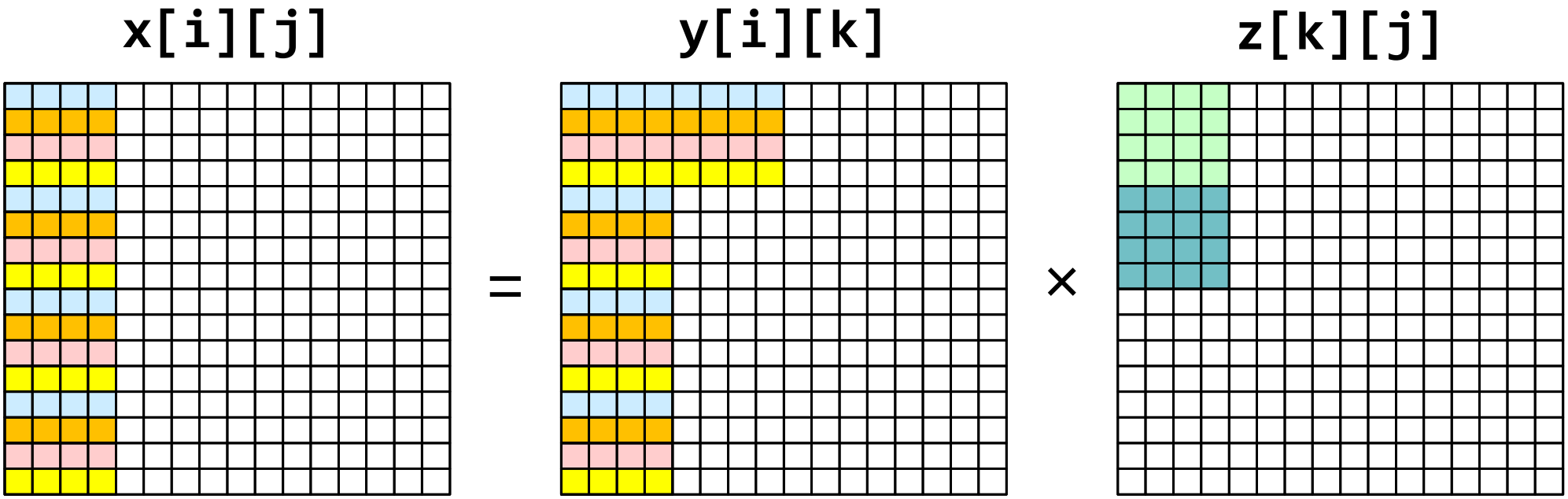
Restructuring Code with Blocking

```
// 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×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:

```
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?

Next . . .

- 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

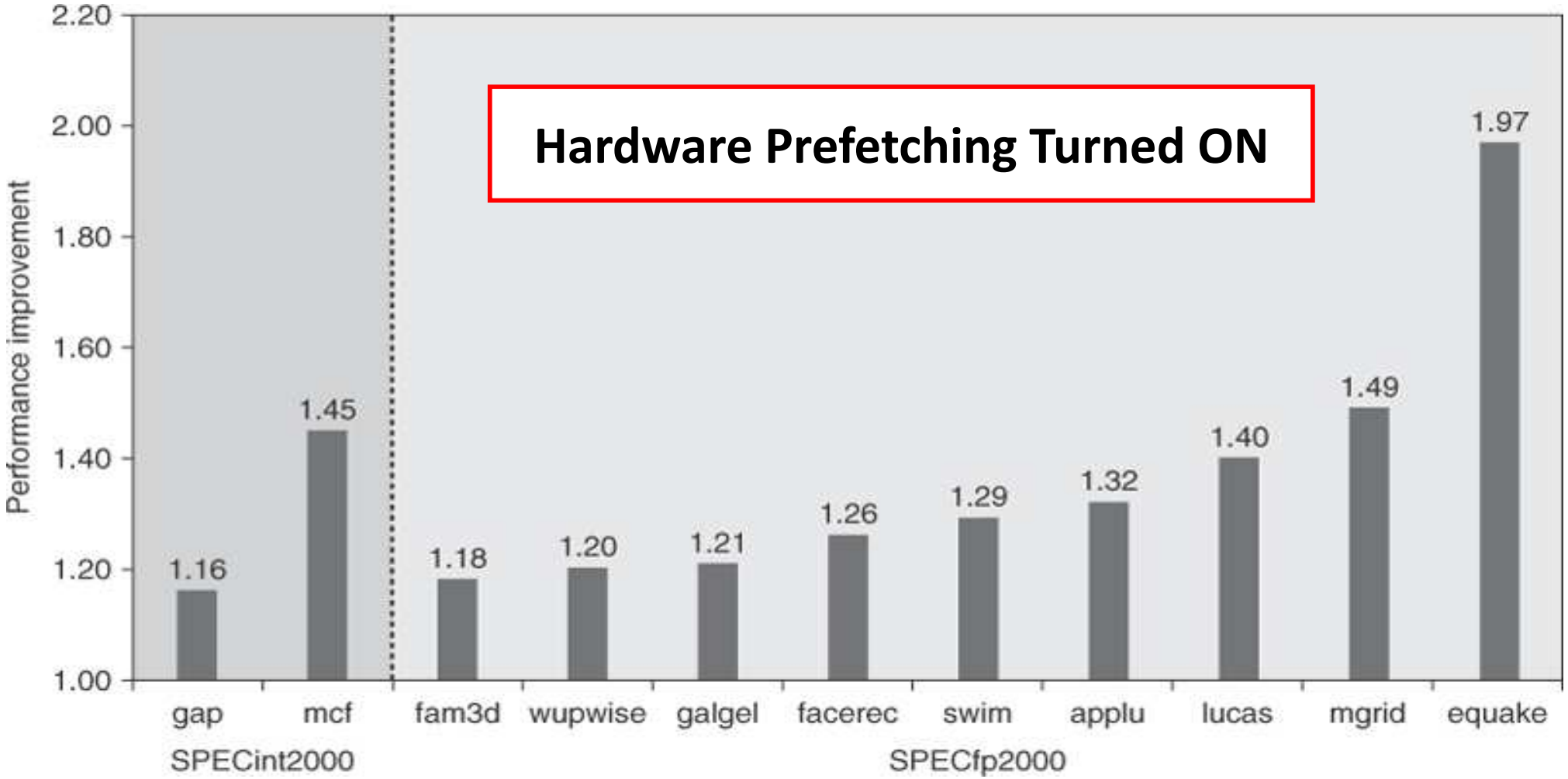
Priority to Read Misses over Writes

- 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

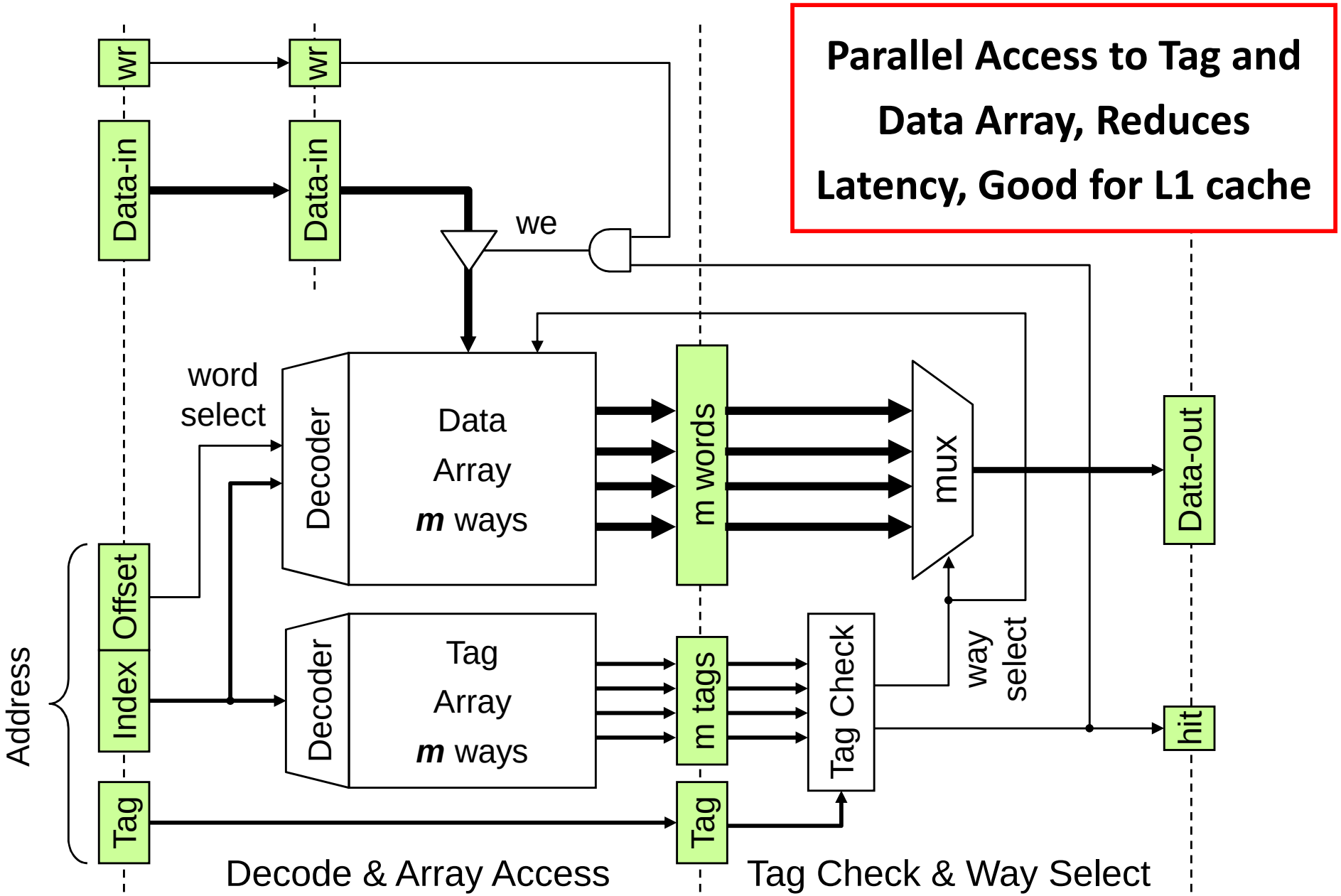
Speedup due to Hardware Prefetching



Pipelined Cache Access

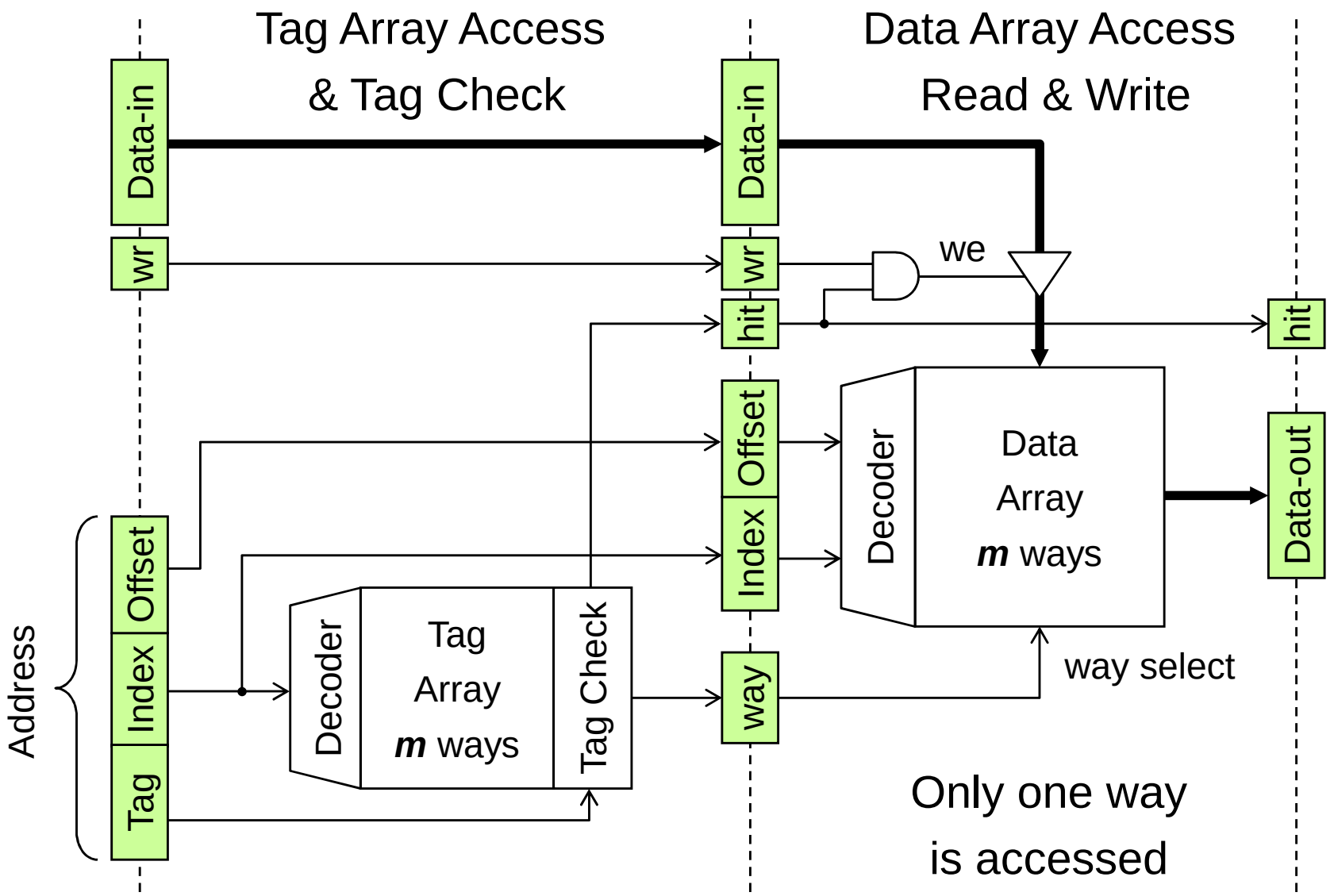
- Used mainly in the L1 Instruction and Data caches
- L1 cache latency is multiple clock cycles (2 to 4 cycles)
- However, L2 and L3 cache accesses are not pipelined
- Advantages of Pipelined Cache Access
  - Faster clock rate and higher bandwidth
  - Better for larger associativity
- Disadvantages
  - 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



Serial Access to Tag and Data Arrays

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

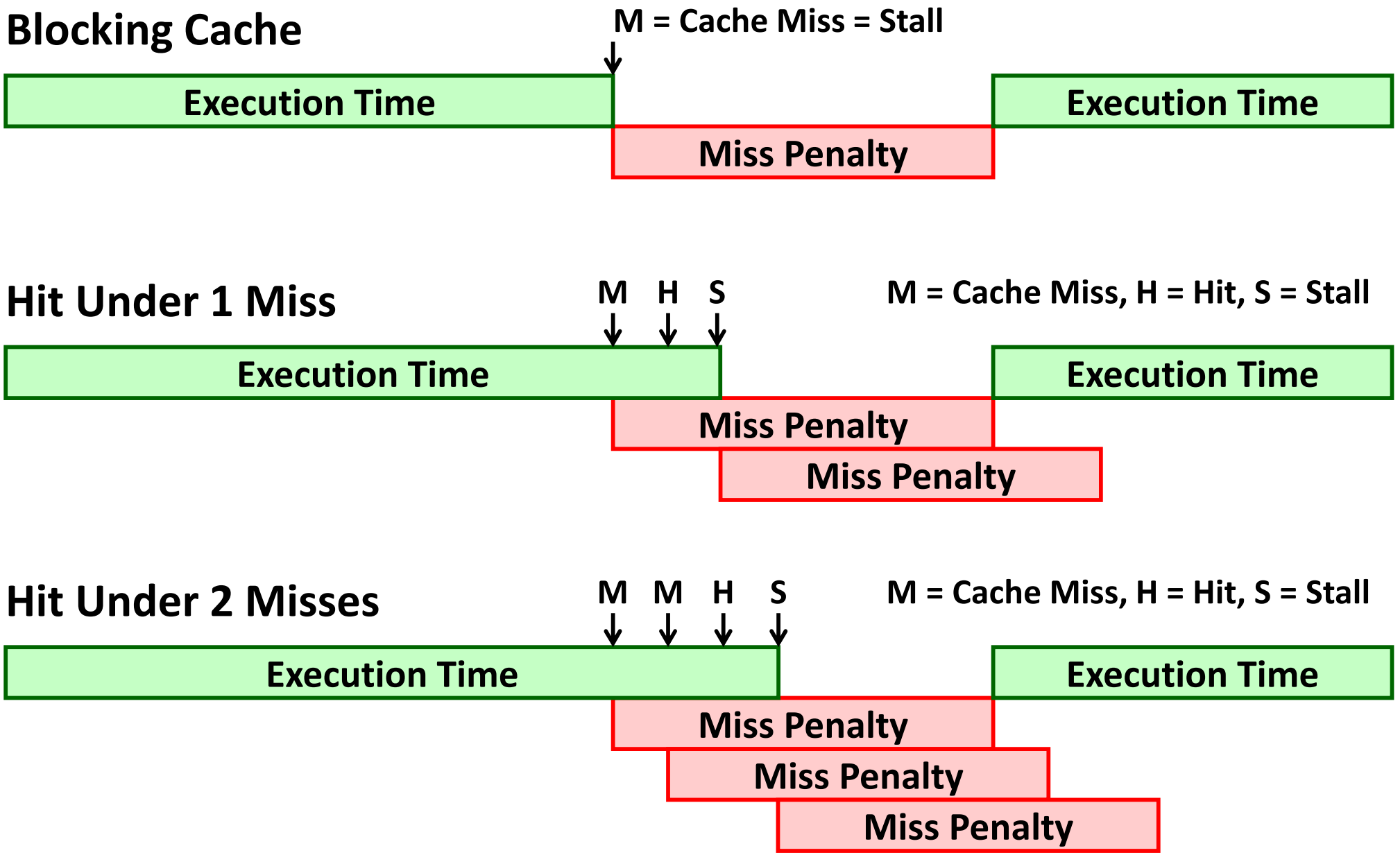


Serial Access to Tag and Data Array, Reduces Energy, Good for L2 and L3 caches

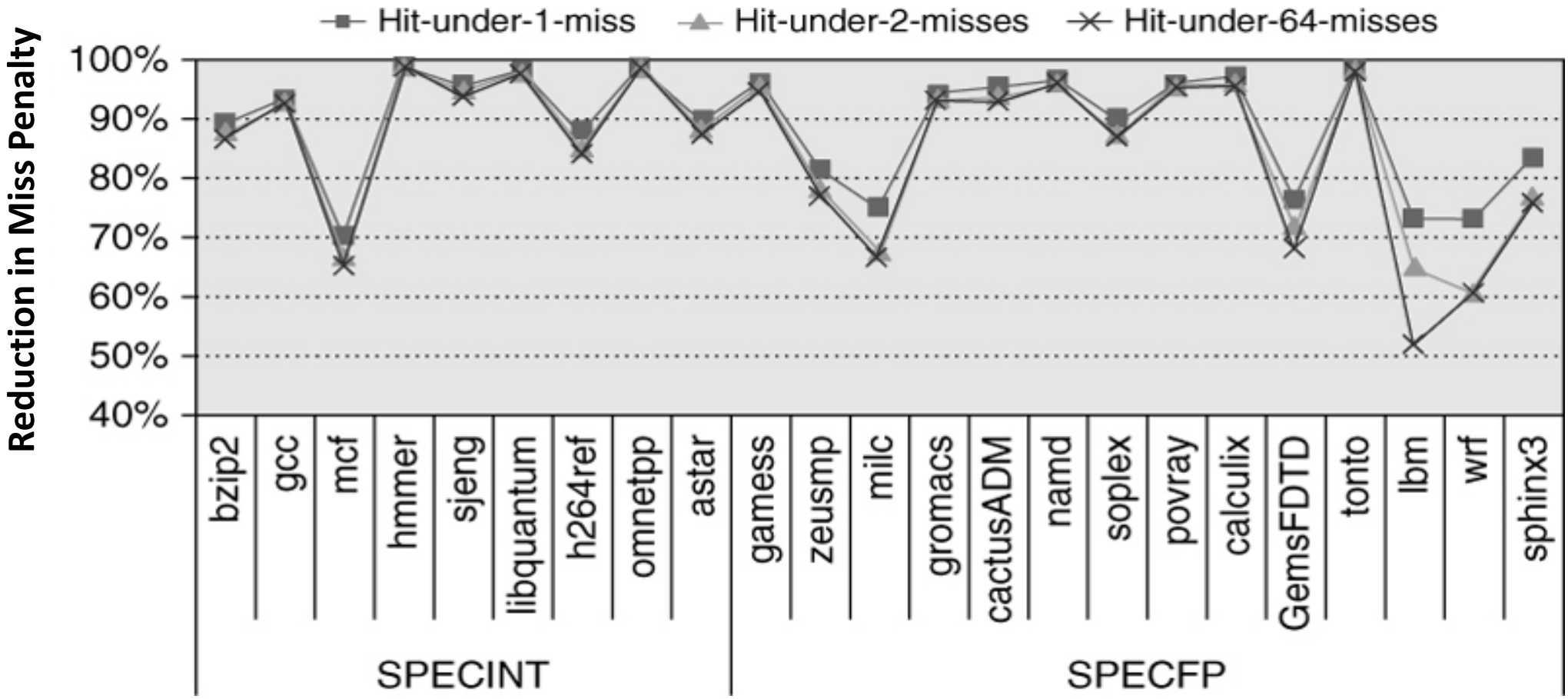
Non-Blocking Cache

- 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



Effectiveness of Non-Blocking Cache

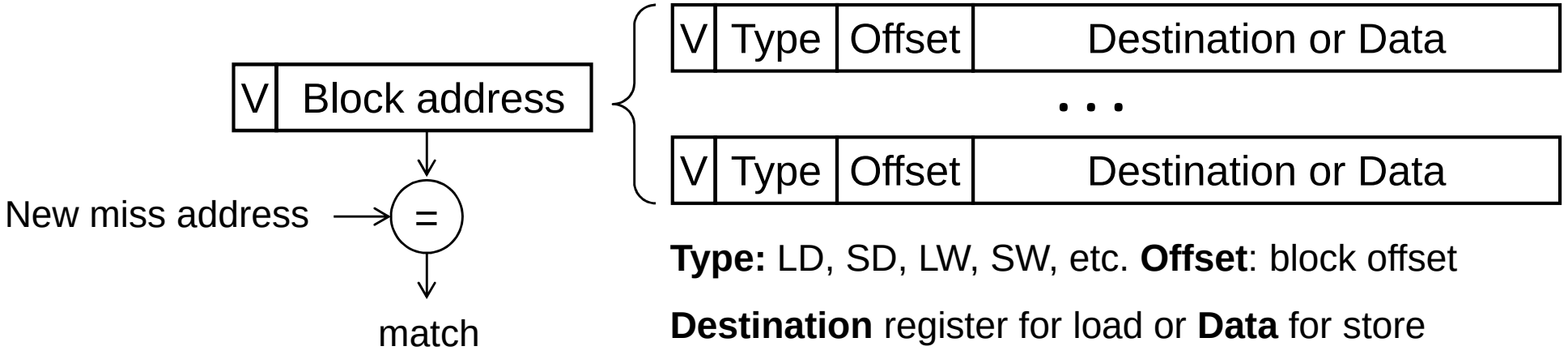


Hit-under-1-miss reduces the miss penalty by 9% (SPECINT) and 12.5% (SPECFP)

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



Non-Blocking Cache Operation

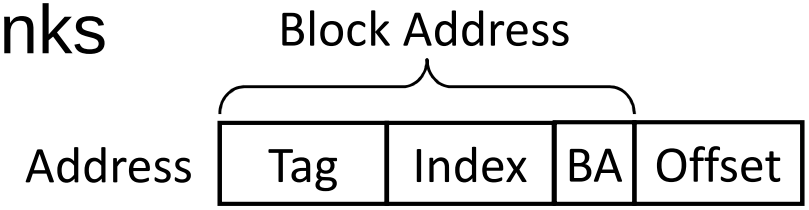
- On Cache Miss, check MSHR for matched block address
  - If found: allocate new load/store entry for matched block
  - If not found: allocate new MSHR and load/store entry
  - If all MSHR resources are allocated then Stall (Structural)
- When cache block is transferred from lower-level memory
  - Process 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 ➔ **Bank Conflict**
  - One request is allowed to proceed, while second request waits

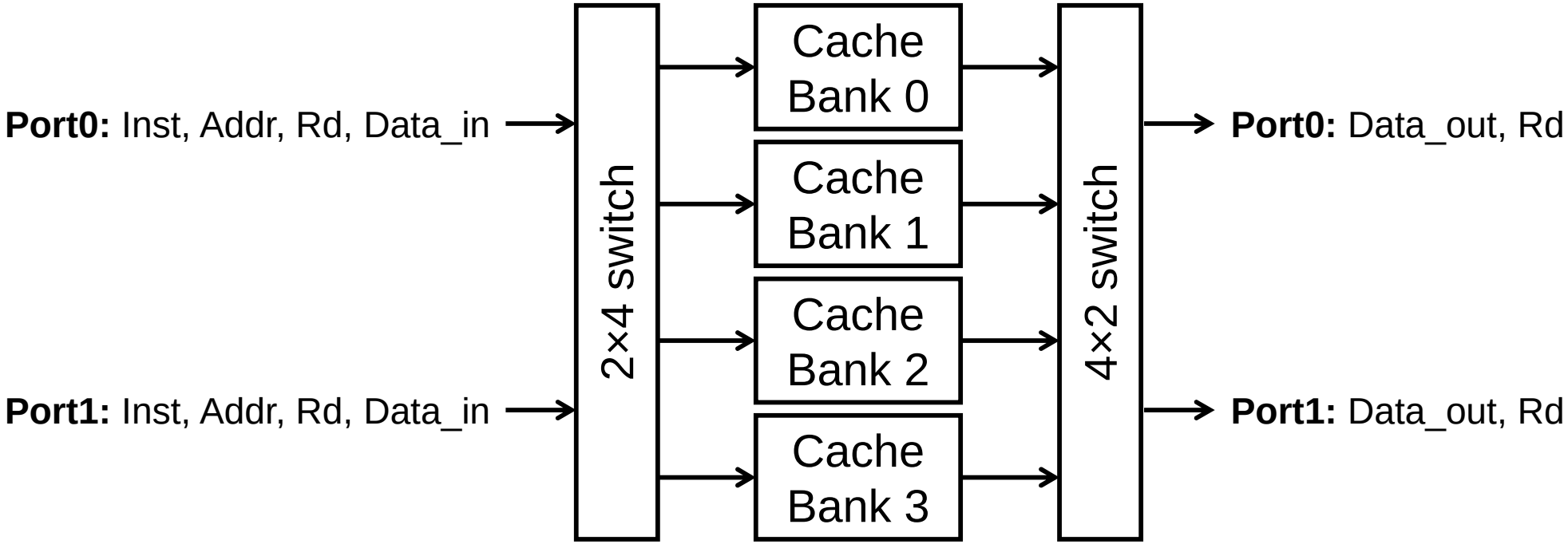


- Example: Sequential interleaving of blocks across 4 cache banks
  - Each cache bank is implemented using a tag array and a data array

Index	Bank 0	Bank 1	Bank 2	Bank 3
0	Block 0, 16, …	Block 1, 17, …	Block 2, 18, …	Block 3, 19, …
1	Block 4, 20, …	Block 5, 21, …	Block 6, 22, …	Block 7, 23, …
2	Block 8, 24, …	Block 9, 25, …	Block 10, 26, …	Block 11, 27, …
3	Block 12, 28, …	Block 13, 29, …	Block 14, 30, …	Block 15, 31, …

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



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