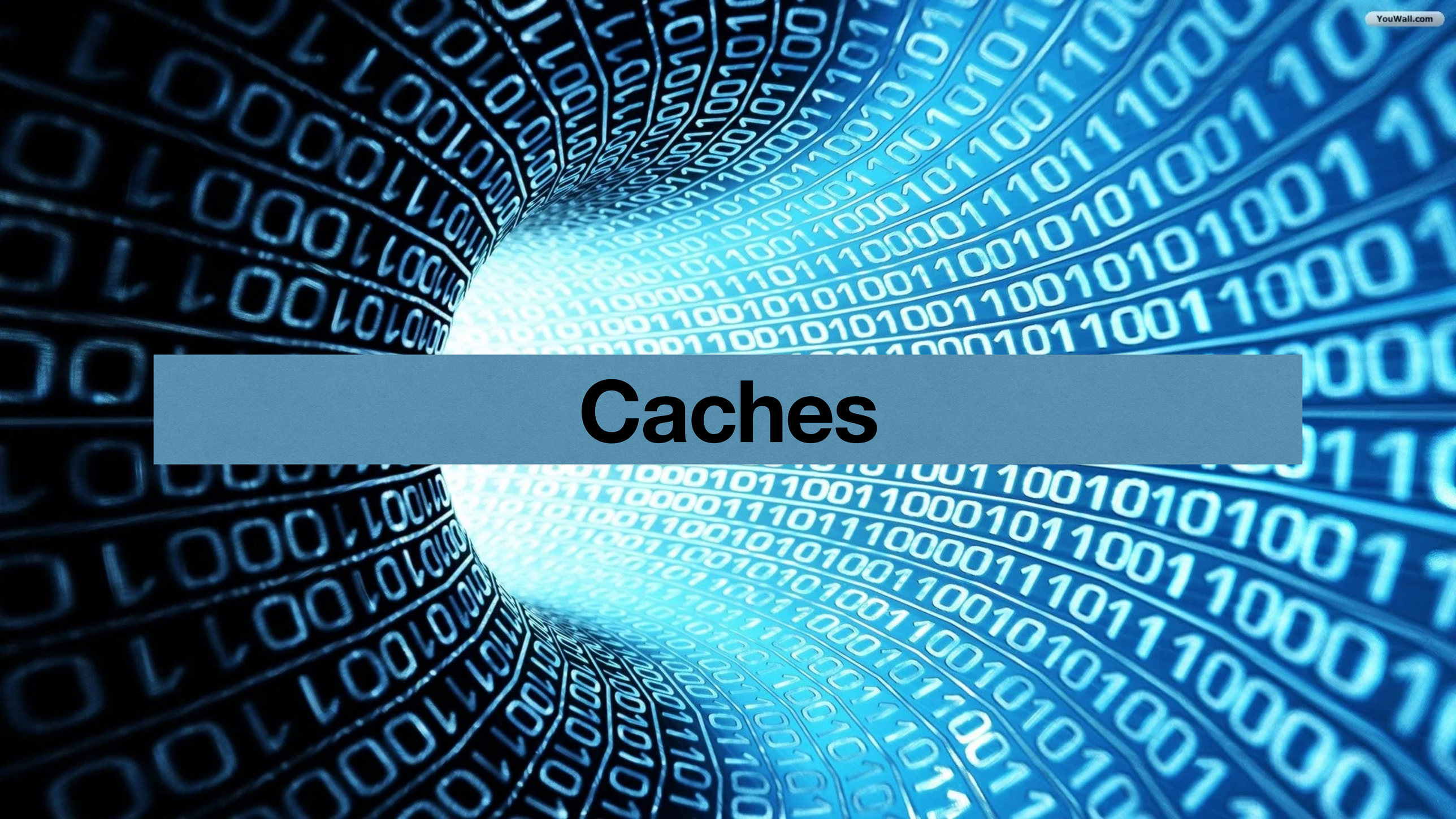
Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay



Caches

A Few Words About Performance

Performance: Time (Iron Law)

Time/Program =

Instructions/program X cycles/instruction X Time/cycle

Source code	ISA	microarch.
Compiler	microarch.	technology
ISA		

Performance

- *Latency* (execution/response time): time to finish one task. It is additive (Performance = 1/latency)
- *Throughput* (bandwidth): number of tasks/unit time. It is not additive

Performance — In our words

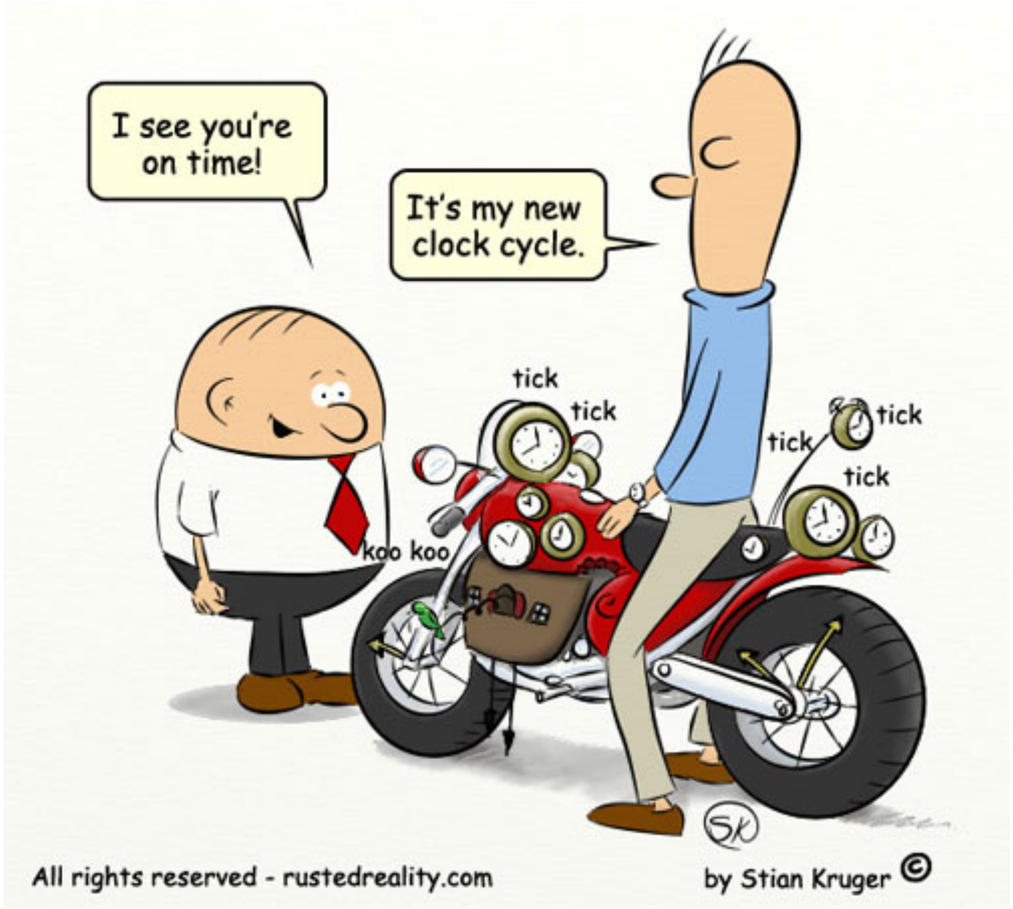
Which computer is faster?

The measure is *execution time*

“X is n times faster than Y”

$$n = \frac{Exetime_Y}{Exetime_X} = \frac{Perfromance_X}{Perfromance_Y}$$

Empirical Evaluation



- Benchmarks
- Metrics
- Simulators
- Latency and bandwidth

Evaluation

- To compare Processor A with Processor B by running programs
- How many programs?
- The programs that you care.
- What if I want to build a new one (processor, caches, DRAM) ?

World of Benchmarks

- SPEC CPU 2017 (https://www.spec.org/cpu2017/)

The **SPEC CPU® 2017** benchmark package contains SPEC's next-generation, industry-standardized, CPU intensive suites for measuring and comparing compute intensive performance, stressing a system's processor, memory subsystem and compiler.

SPECspeed: used for comparing time for a computer to complete single tasks

SPECrate: measure the throughput or work per unit of time.

What are there? From a GCC compiler, Gaming, Video compression, Chess(AI), Differential Equation solver, Numerical programs, searching genome sequence, quantum computer simulation and many more (SPEC 2006)

World of Benchmarks

CloudSuite (https://www.cloudsuite.ch/)

CloudSuite is a benchmark suite for cloud services. The benchmarks are based on real-world software stacks and represent real-world setups.

PARSEC (https://parsec.cs.princeton.edu/)

Benchmark suite composed of multithreaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors.

World of Benchmarks

MobileBench (https://mobilebench.engineering.asu.edu/) comprising a selection of representative smart phone applications.

Many more application domain specific: Graph processing, ML perf,

Rules for Measuring Performance with Benchmarks

- No Source code modifications are allowed, or it is impossible
- Use one compiler, one language for all the benchmark programs and don’t play with the flags

- Else, you can cheat!!!
- Also, use *benchmark suites*, not a single benchmark

Pitfalls of Benchmarks

Benchmark not representative of all

Your workload is I/O bound → SPECCPU is useless

Benchmark is too old

Need to be periodically refreshed

Non-benchmarks

- Application kernels: A small code fragment or part of the program
- Synthetic benchmark : Not part of any real program!!
- Micro-benchmark

World of Simulators

- Functional Simulator: Used to verify the correct execution of the program. Can not be used for performance evaluation.

- Performance simulators:

(i) Trace-driven: ChampSim (https://github.com/ChampSim/ChampSim)

(ii) Execution-driven: gem5, Multi2sim

Functional simulator is part of the performance simulators.

Evaluation Continued

Pick a *relevant* benchmark suite

Measure IPC of each program

Summarize the performance using:

- Arithmetic Mean (AM)
- Geometric Mean (GM)
- Harmonic Mean (HM)

Which one to choose?

Example

	IMTEL	ABM	AND
App. one	10	20	30
App. two	20	30	40
App. three	30	40	10

Which machine performs better over IMTEL and why?

Example

	ABM	AND
App. one	2	3
App. two	1.5	2
App. three	1.3	0.3
A.M.	1.60	1.76
G.M.	1.57	1.21
H.M.	1.54	0.72

AM on ratios

	X	Y
App. 1	1	100
App. 2	1000	10

Normalized to X	X	Y
App. 1	1	100
App. 2	1	0.01
AM	1	50.005

Y is 50 times faster than X

Normalized to Y	X	Y
App. 1	0.01	1
App. 2	100	1
AM	50.005	1

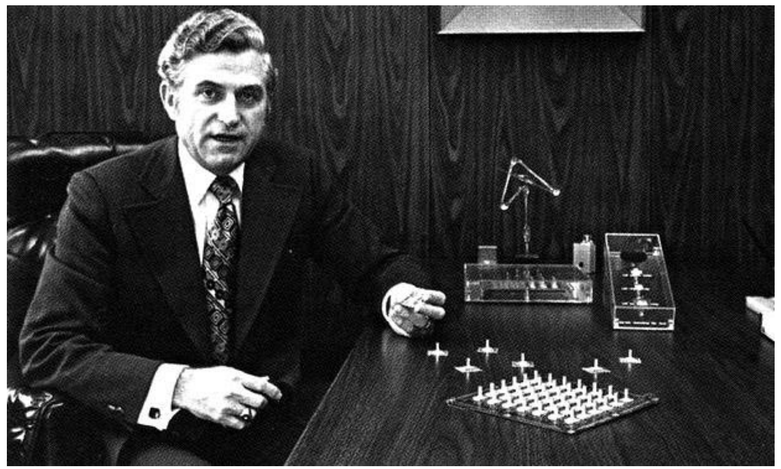
X is 50 times faster than Y

AM vs. GM

- GM of ratios is same as the ratio of the GMs
- Due to the aforementioned fact, the choice of reference does not matter if you go with GM.

Principles of Computer Design

Amdahl's Law (common case fast)

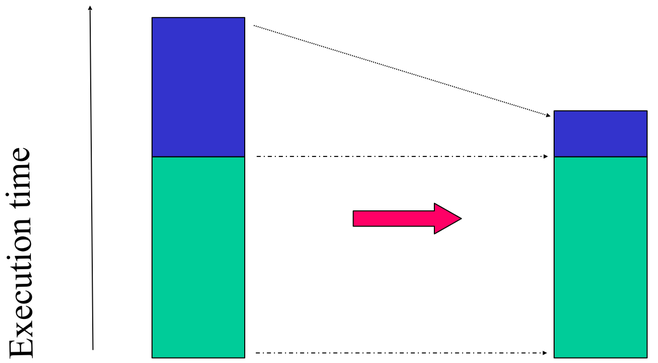


$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}}$$

$$= \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Amdahl's Law and Speedup

- Speedup: How much faster a machine will run due to an enhancement?
- Need to consider two things while using Amdahl's law:
  - 1st… **Fraction of the computation time that can use the enhancement**
    - If a program executes in 30 seconds and 15 seconds of exec. uses enhancement, fraction = ½
  - 2nd… **Improvement gained by enhancement**
    - If enhanced task takes 3.5 seconds and original task took 7secs, we say the speedup is 2.



Amdahl's Law: Example

- Floating point instructions improved to run 2 times faster.
- But, only 10% of actual instructions are FP

  - $ExTime_{new} = ?$
  - $Speedup_{new} = ?$

Example: Answer

- Floating point instructions improved to run 2X faster.
  - But only 10% of actual instructions are FP.

$$\text{ExTime}_{new} = \text{ExTime}_{old} \times (0.9 + 0.1/2) = 0.95 \times \text{ExTime}_{old}$$

$$\text{Speedup}_{overall} = \frac{1}{0.95} = 1.053$$

Example 2

A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FSQRT) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FSQRT hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

Example 2

We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FSQRT}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Improving the performance of the FP operations overall is slightly better because of the higher frequency.

Amdahl's Law

Which one will provide better overall speedup?

A. Small speedup on the large fraction of execution time.
B. Large speedup on the small fraction of execution time.
C. Does not matter.

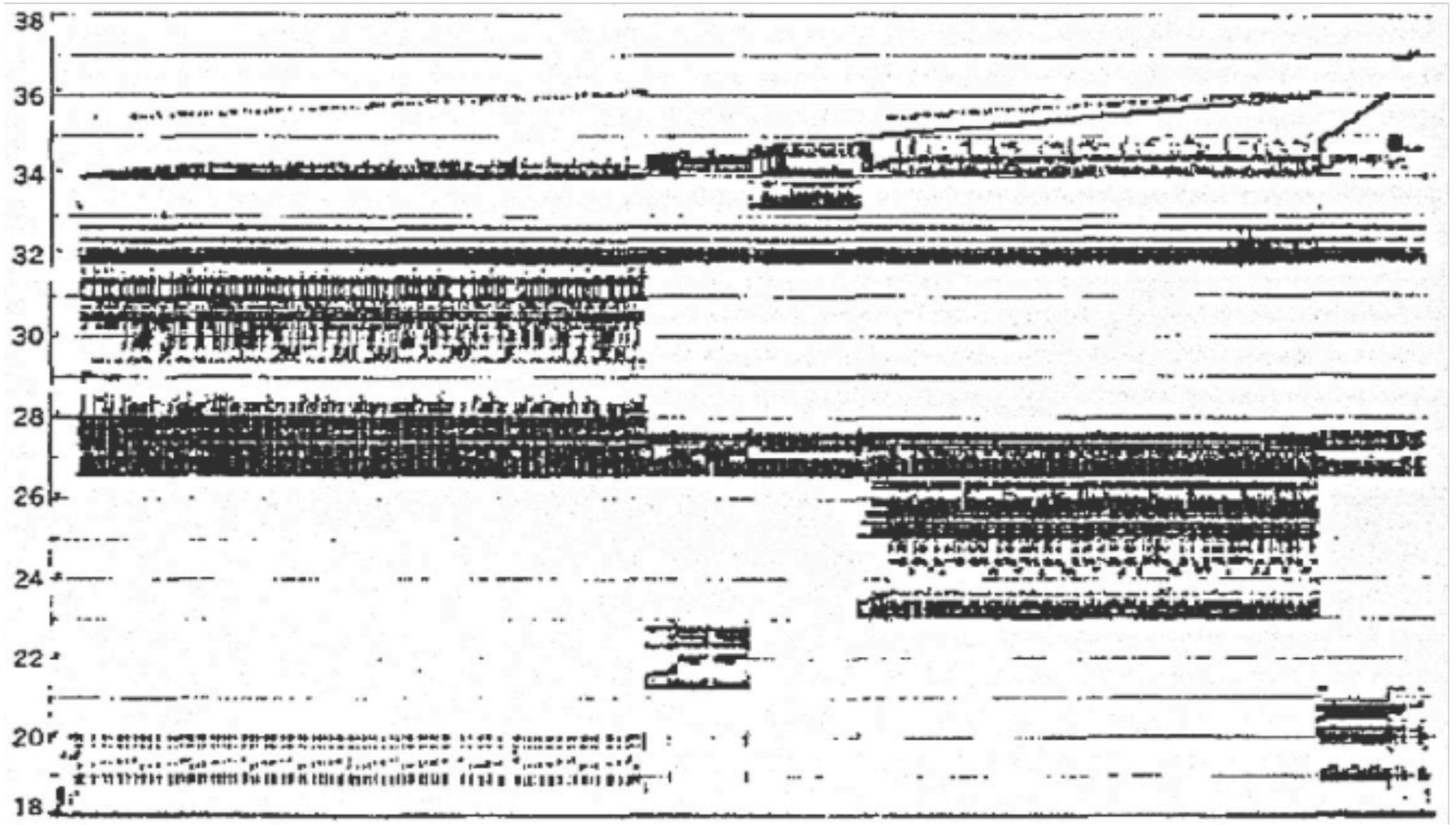
Depends on the difference between small and large. Mostly it is A.

Principle of Locality

- Programs tend to use the data and instructions they have used recently.

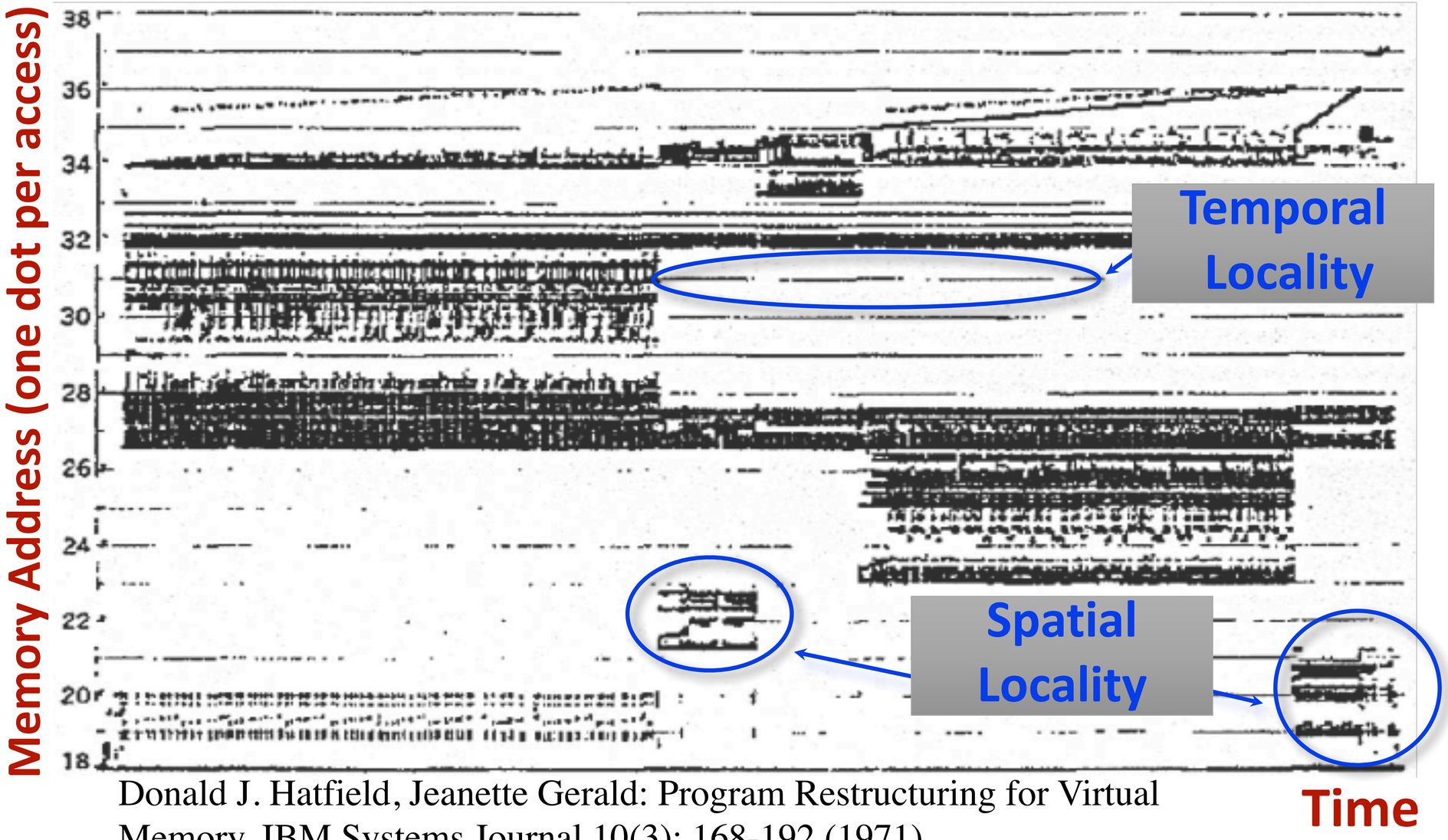
- So from the recent past, we can have a good idea of future
- Temporal Locality: Recently accessed items are to be used in near future.
- Spatial Locality: Items whose addresses are near to each other tend to referenced close together in time.

Let's look at the Applications (benchmarks)



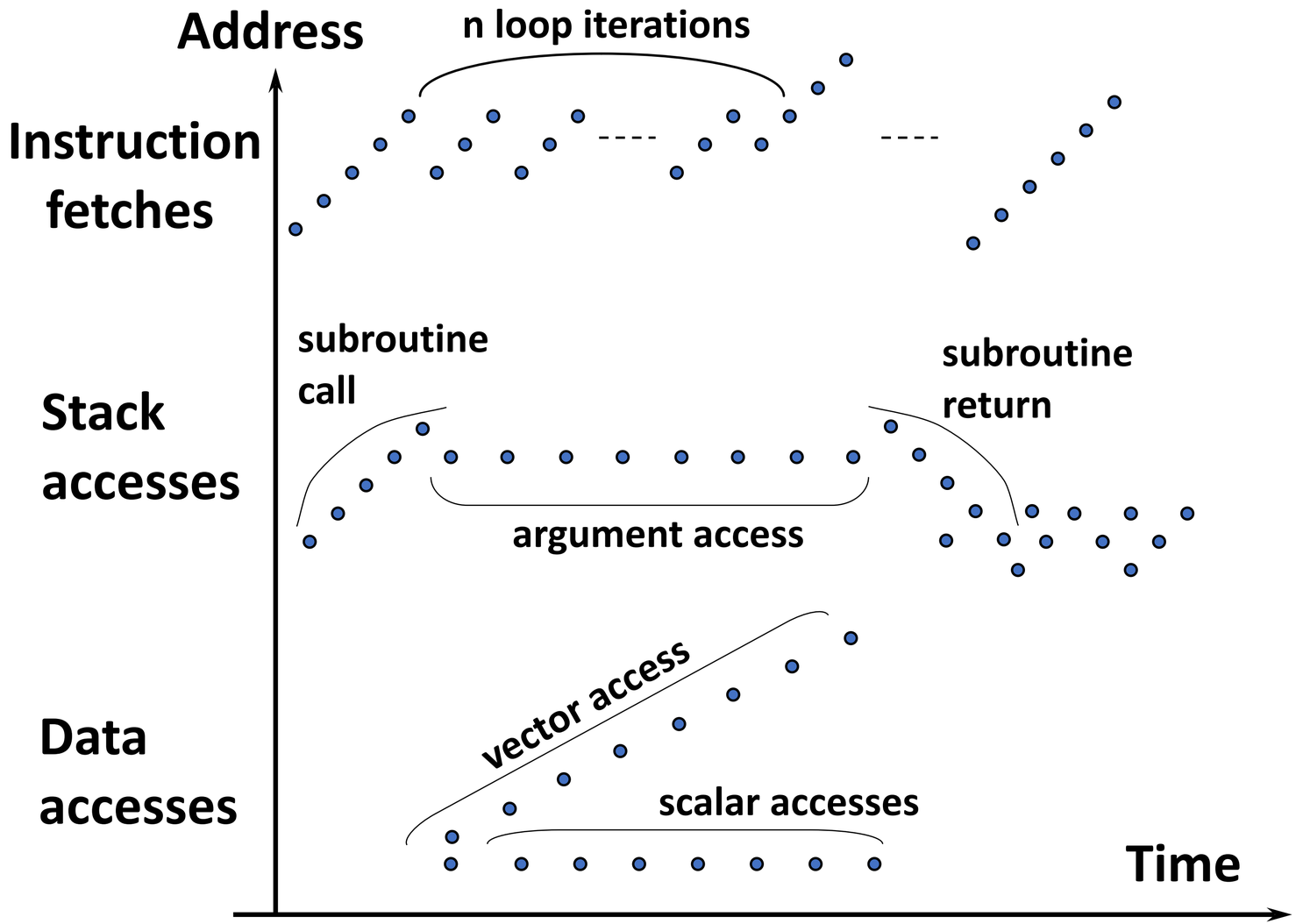
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Locality



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Few Examples



Back to Memory

World with no caches

North pole ☹

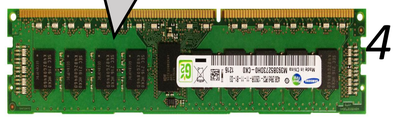
Core

32-bit Address

Data

200 to 300 cycles

Costly DRAM accesses☹



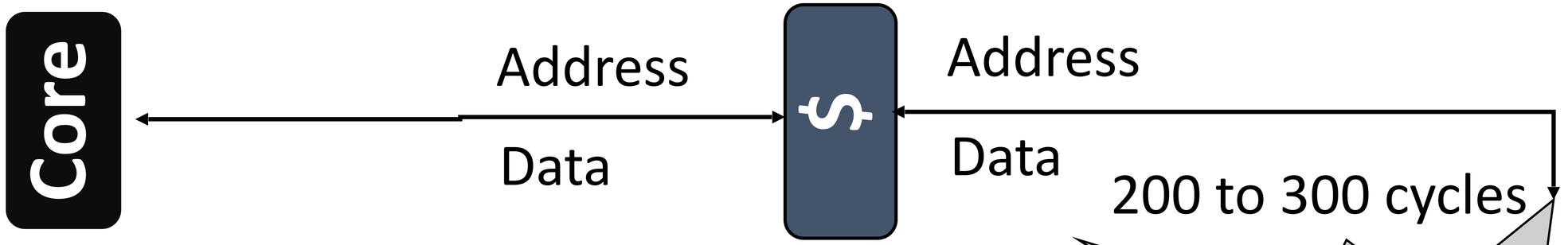
4 GB DRAM

South pole ☹

Minimizing costly DRAM accesses is critical for performance

Caching: Why does it works…?

North pole ☺



Costly DRAM accesses☹

Caching is a speculation technique ☺
Works – if locality

Do not ignore the common case

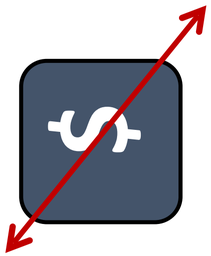
Reduction in DRAM accesses ~ Improvement in execution time

WRONG!

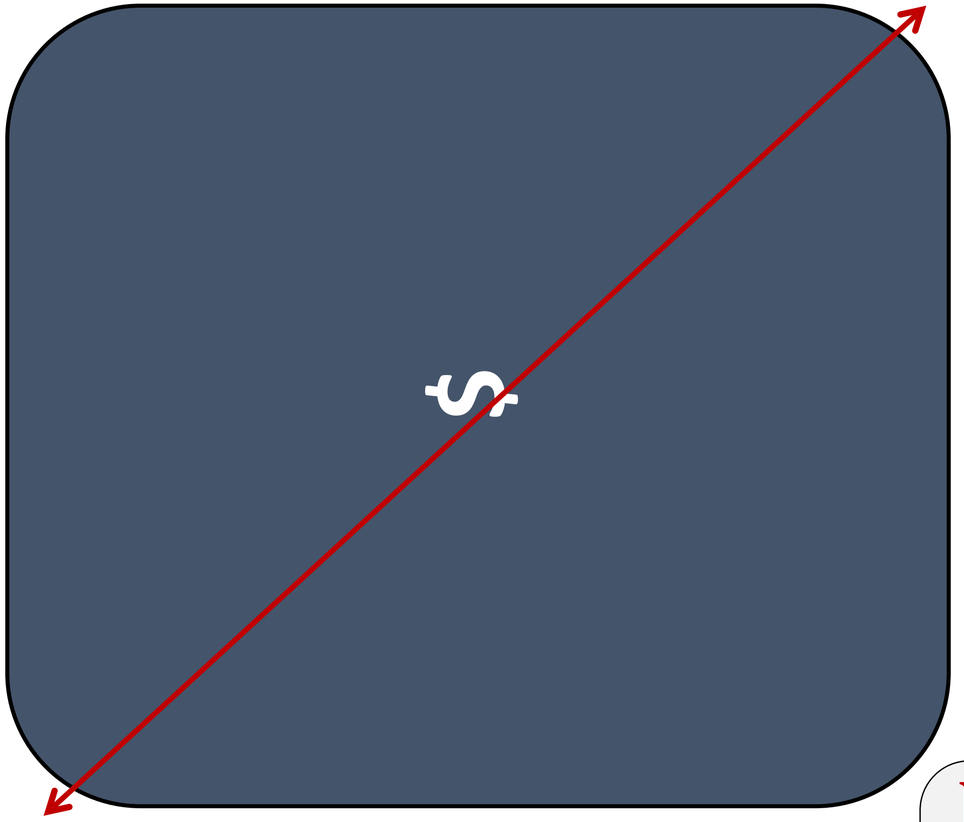
What if your program is not memory intensive

How big/small?

Core



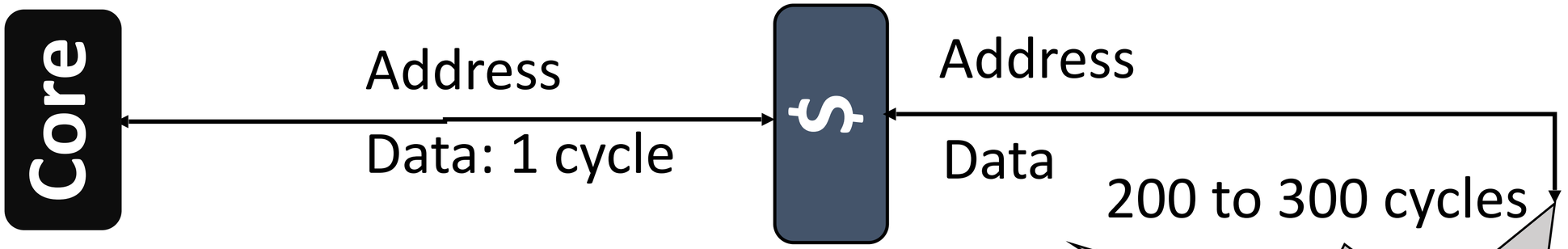
Latency: low
Area: low
Capacity: low



Latency: high
Area: high
Capacity: high

Cache with latency

North pole ☺



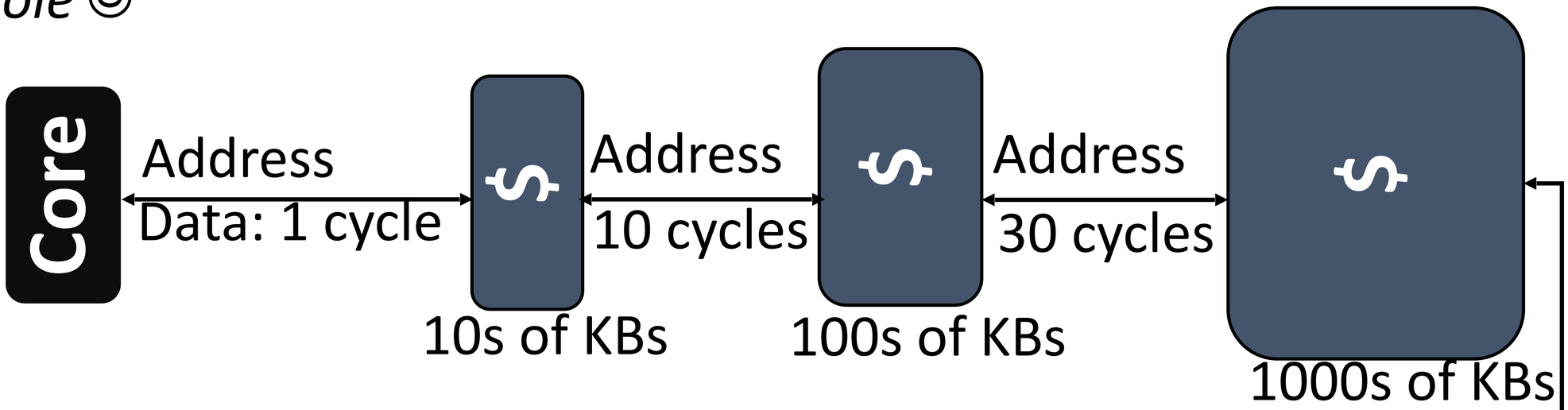
Costly DRAM accesses☹

32 to 64KB $ will be available in one to four cycles ☹

South pole ☺

Cache hierarchy with latency

North pole ☺



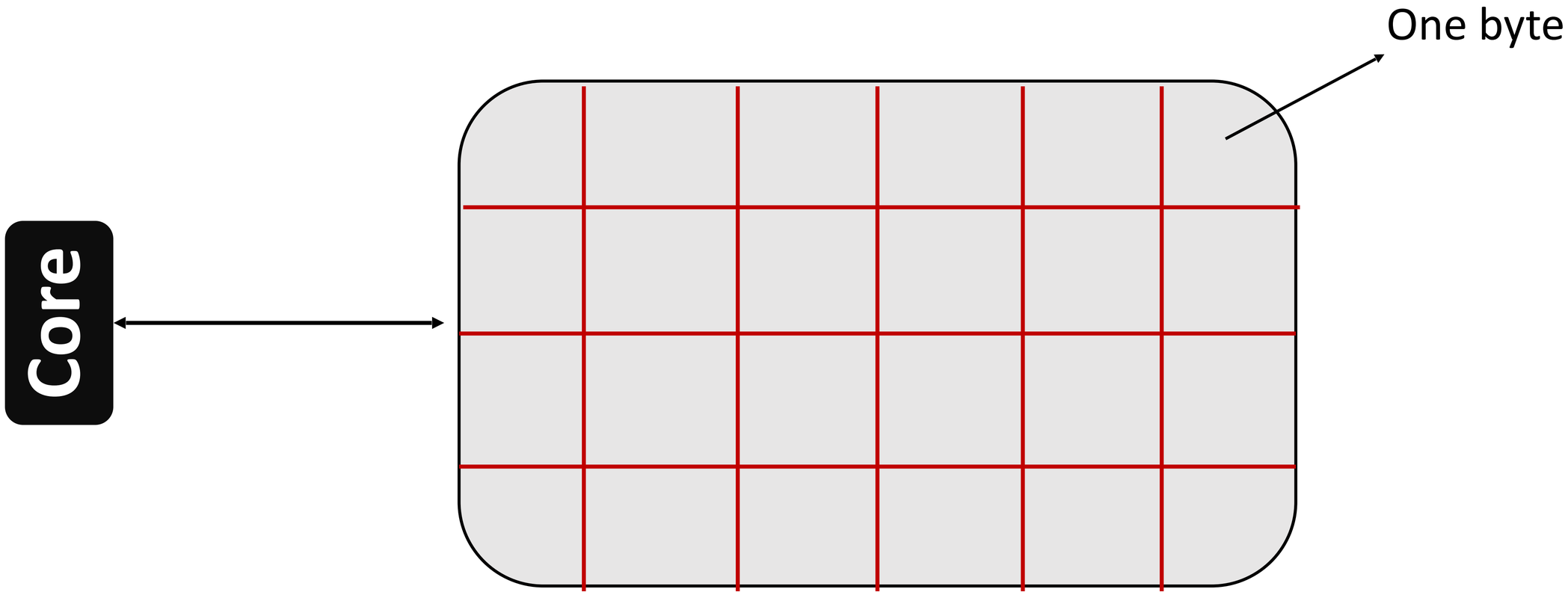
Multi-level cache hierarchy

How many levels ?

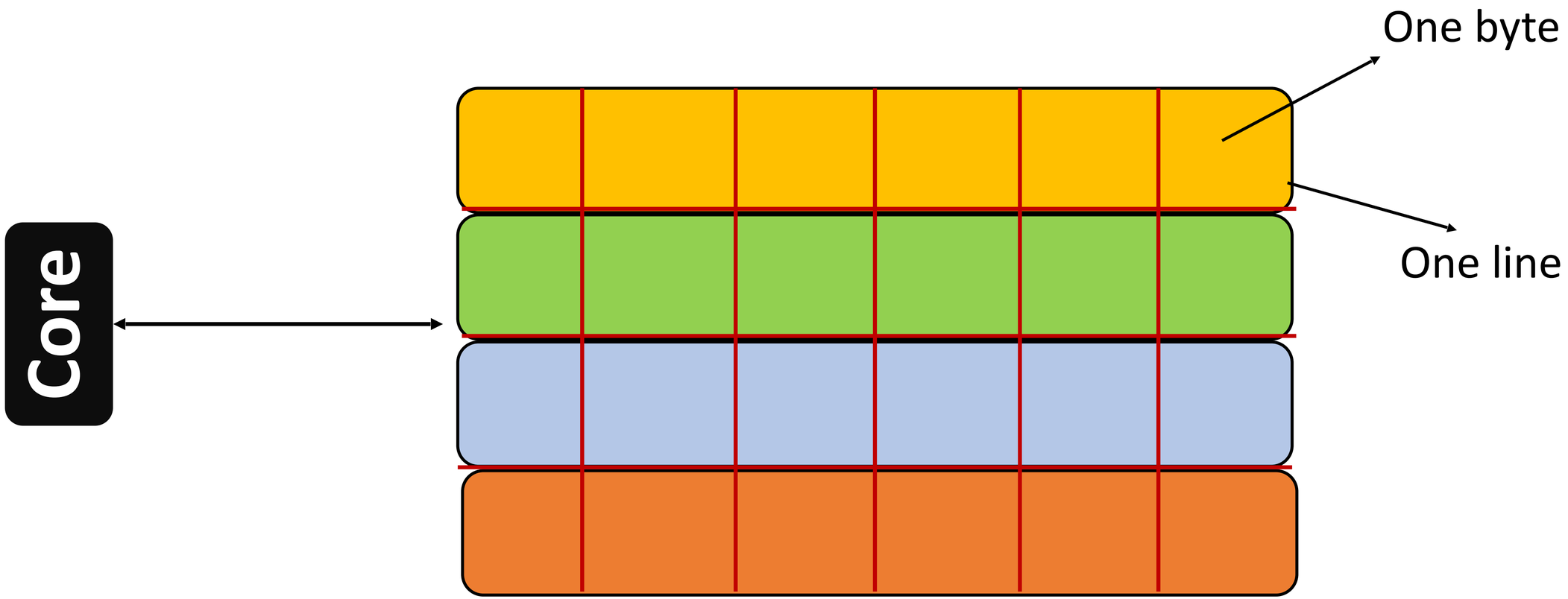
Total latency < DRAM latency

South pole ☺

Accessing a cache

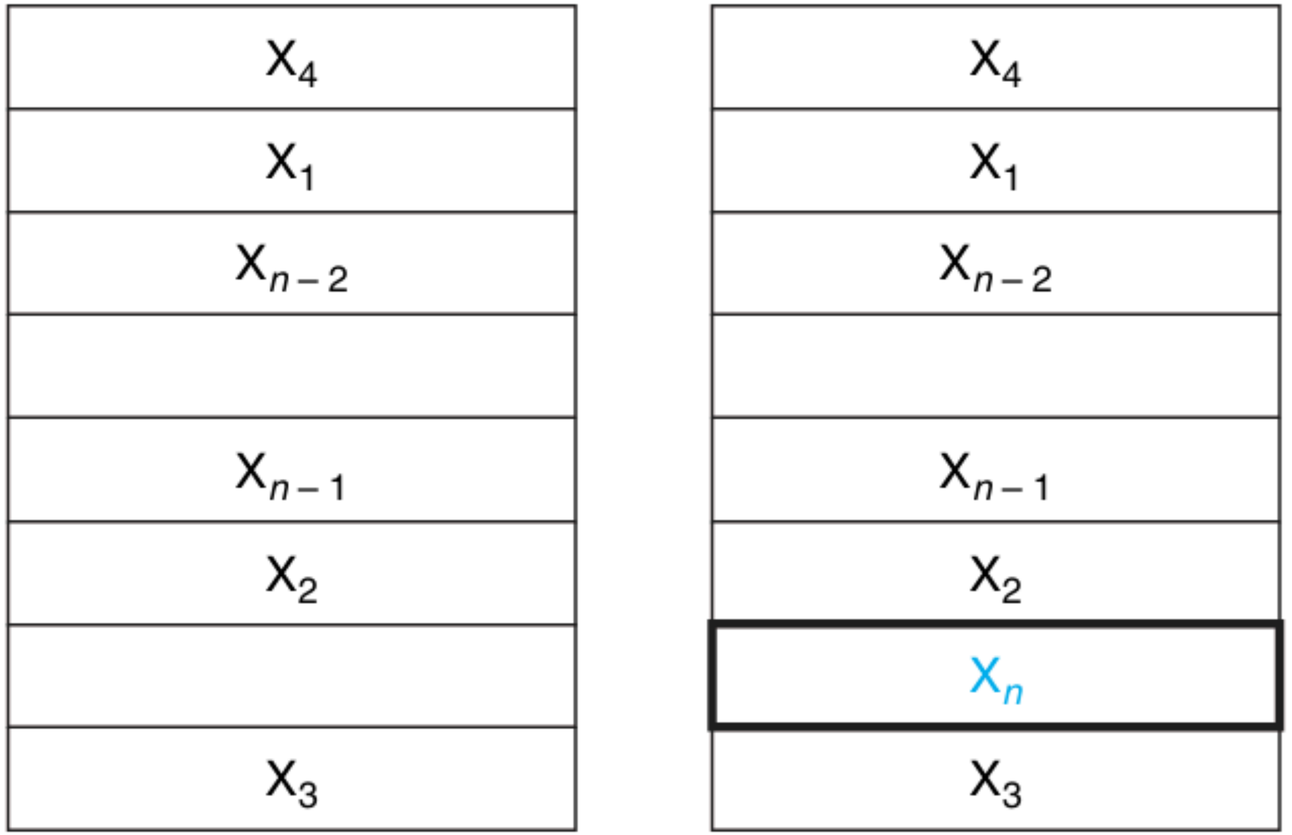


Bytes to blocks (lines)



Typical line size: 64 to 128 Bytes

Before and After You Access…

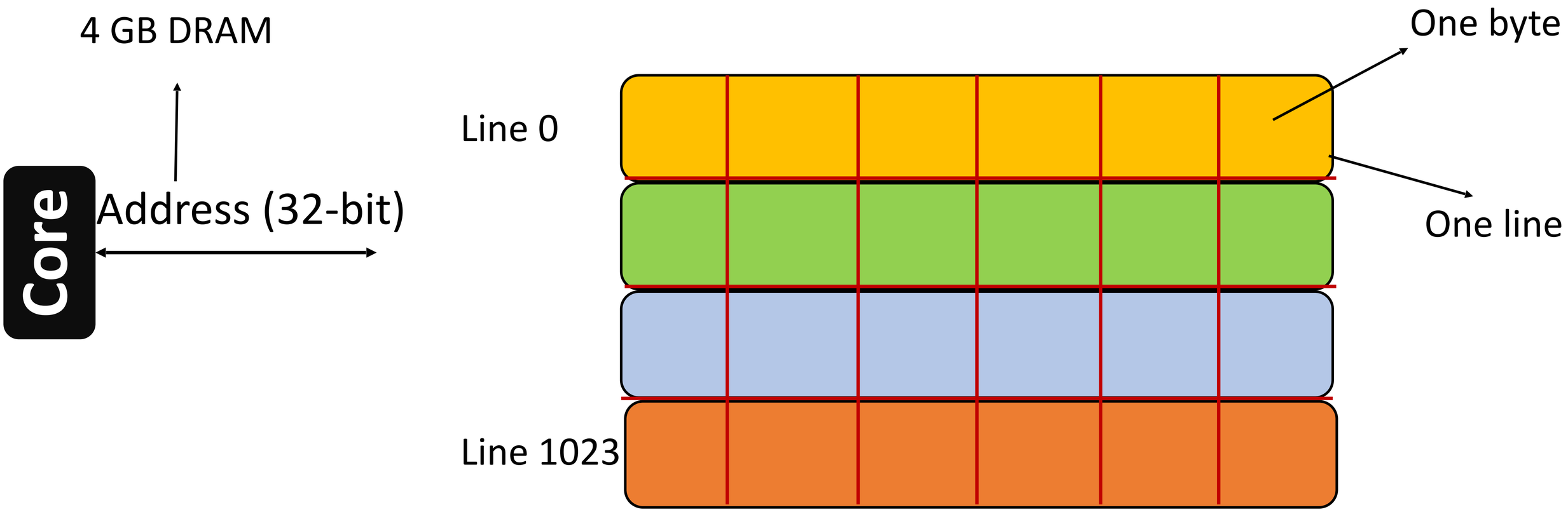


a. Before the reference to X_n b. After the reference to X_n

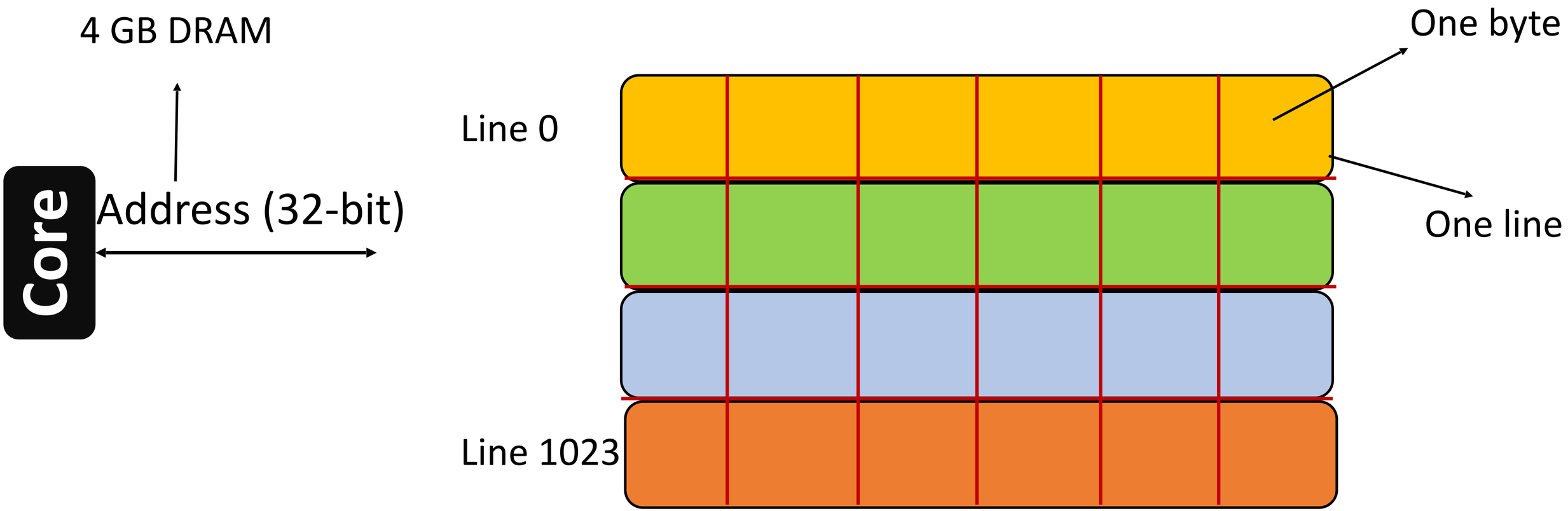
Accessing a cache

- Although cache blocks are of 16/128 bytes processor can still access 1 byte or one word — how?
  - Offset…
- How to efficiently utilize the limited memory? — every program wants to use it no?
  - Cache associativity and replacement and tags

A bit deeper: 1024 lines each of 32B



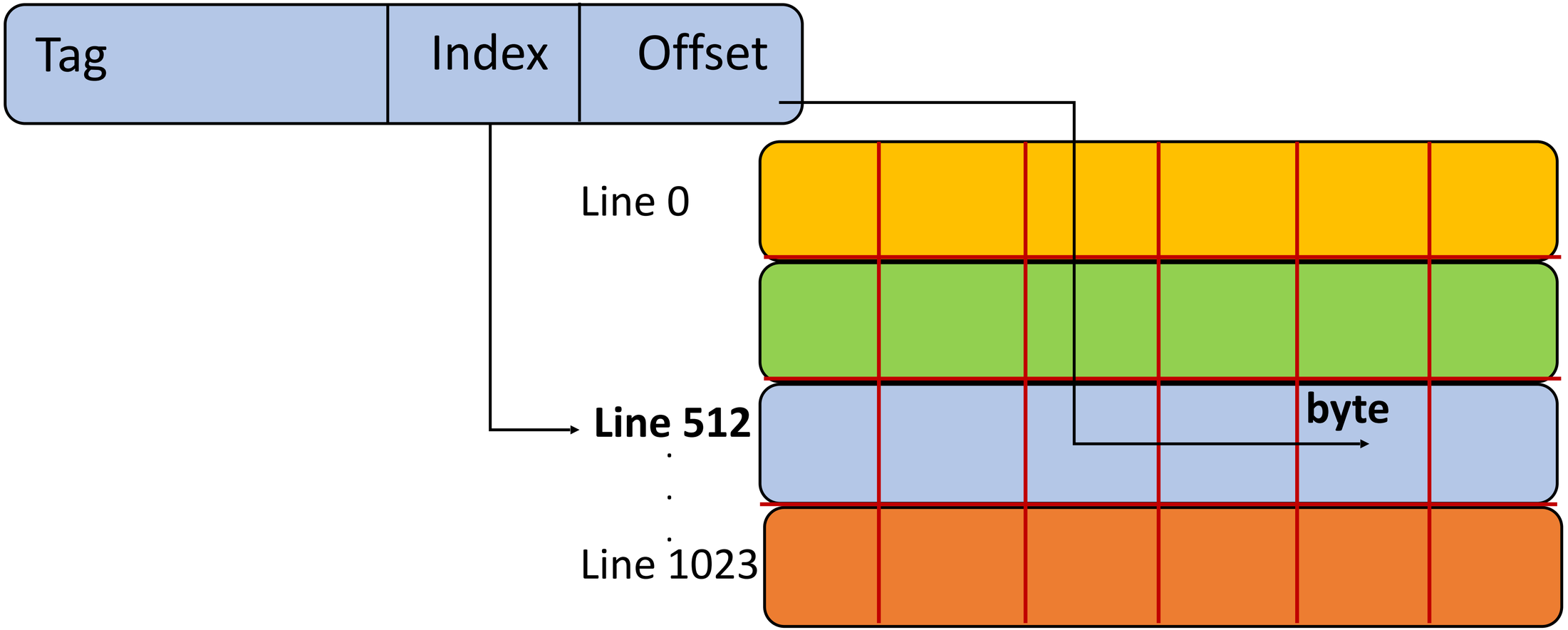
A bit deeper: 1024 lines each of 32B



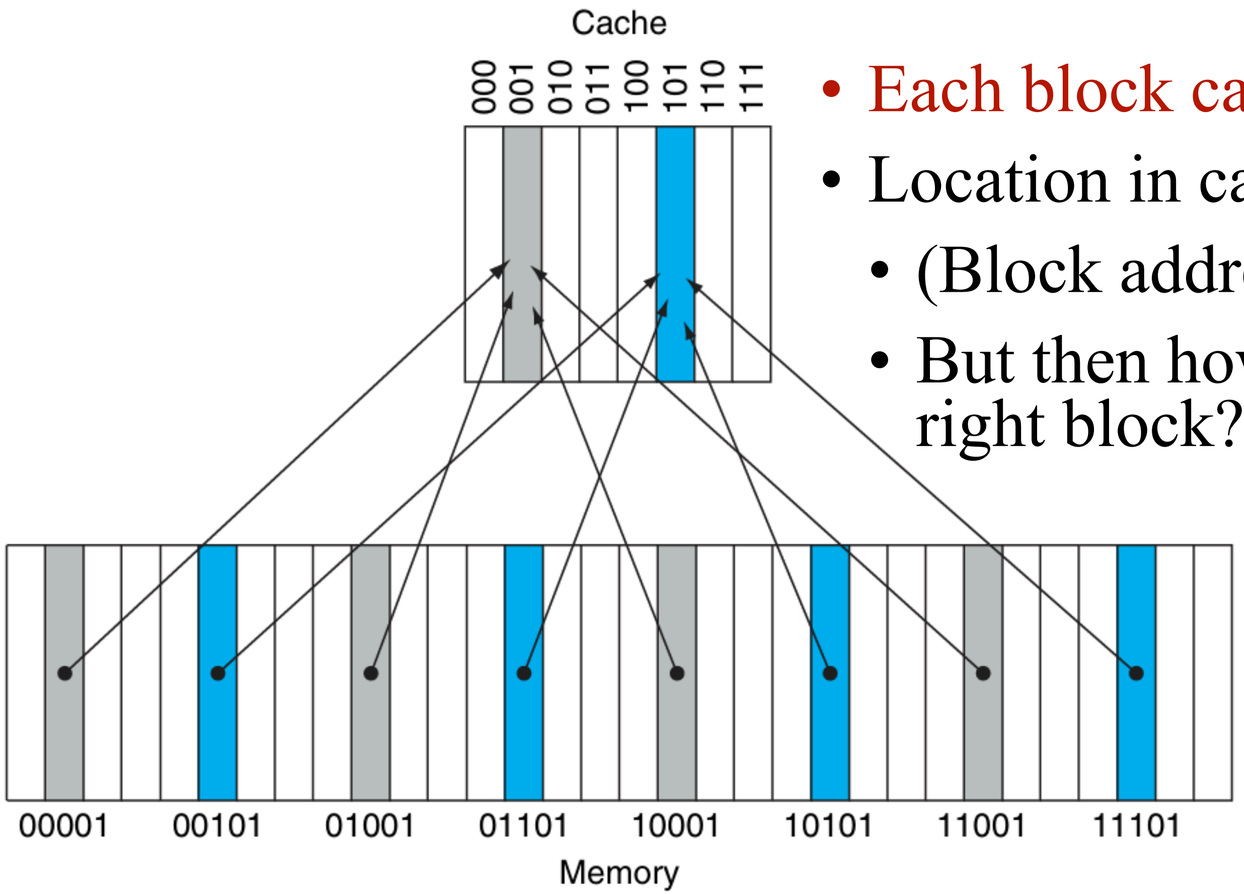
Line number (index): 10 bits

Byte offset (offset): 5 bits

Direct Mapped Cache

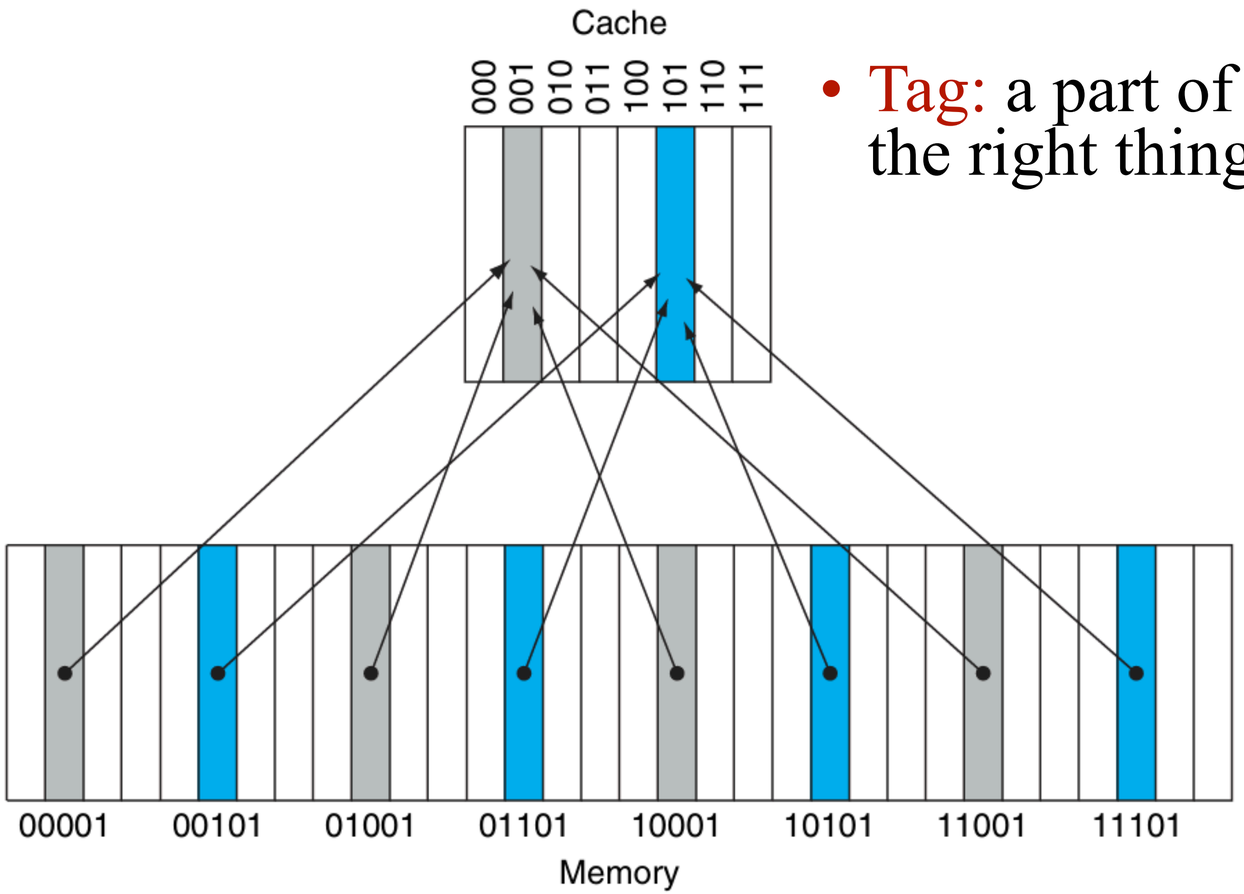


Direct Mapped Cache



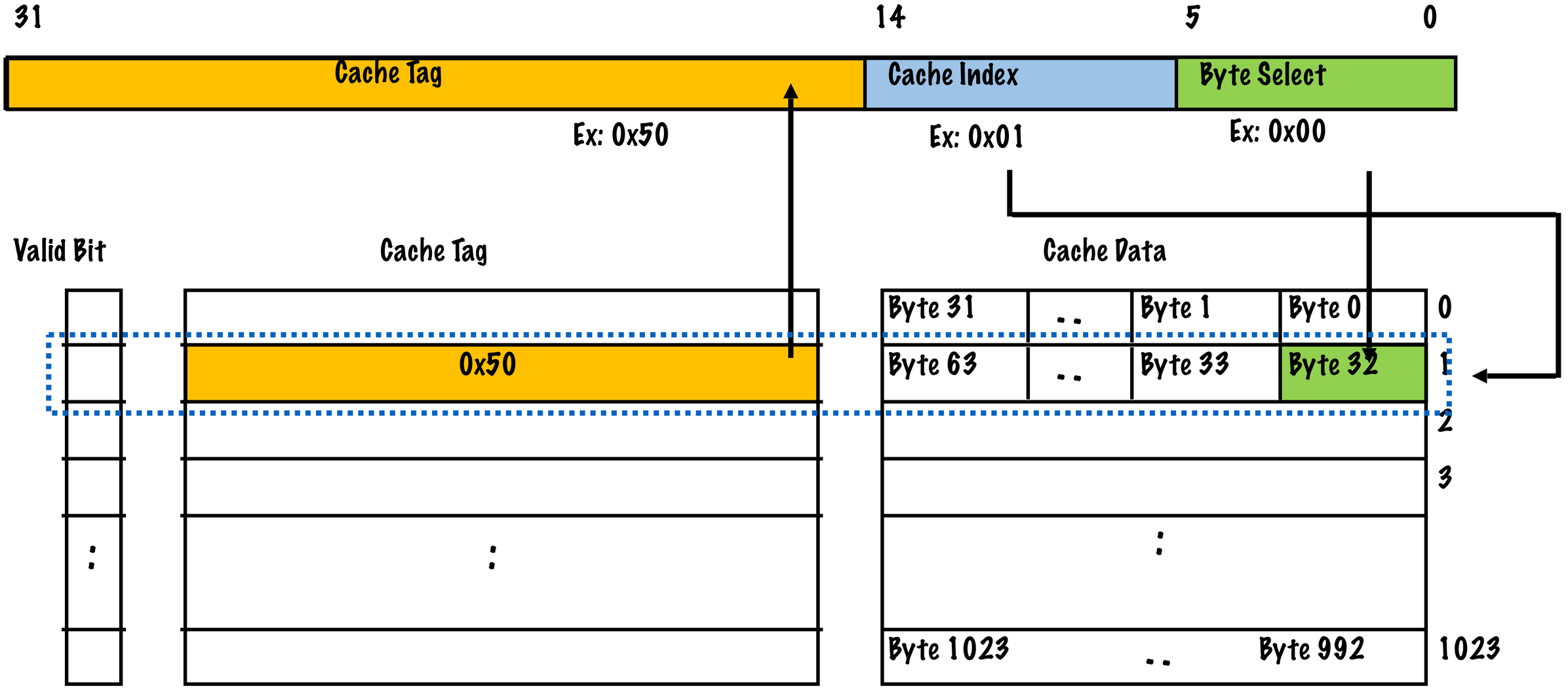
- Each block can go to exactly one place in the cache
- Location in cache:
  - (Block address)mod (number of blocks in the cache)
  - But then how do you know you are accessing the right block?

Direct Mapped Cache



- Tag: a part of your address which helps you identify the right thing…

Direct Mapped in Action



Accessing a Cache

Binary address of reference	Assigned cache block (where found or placed)
10110_{two}	$(10110_{two} \bmod 8) = 110_{two}$
11010_{two}	$(11010_{two} \bmod 8) = 010_{two}$
10110_{two}	$(10110_{two} \bmod 8) = 110_{two}$
11010_{two}	$(11010_{two} \bmod 8) = 010_{two}$
10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$
00011_{two}	$(00011_{two} \bmod 8) = 011_{two}$
10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$
10010_{two}	$(10010_{two} \bmod 8) = 010_{two}$
10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$

Accessing a Cache

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

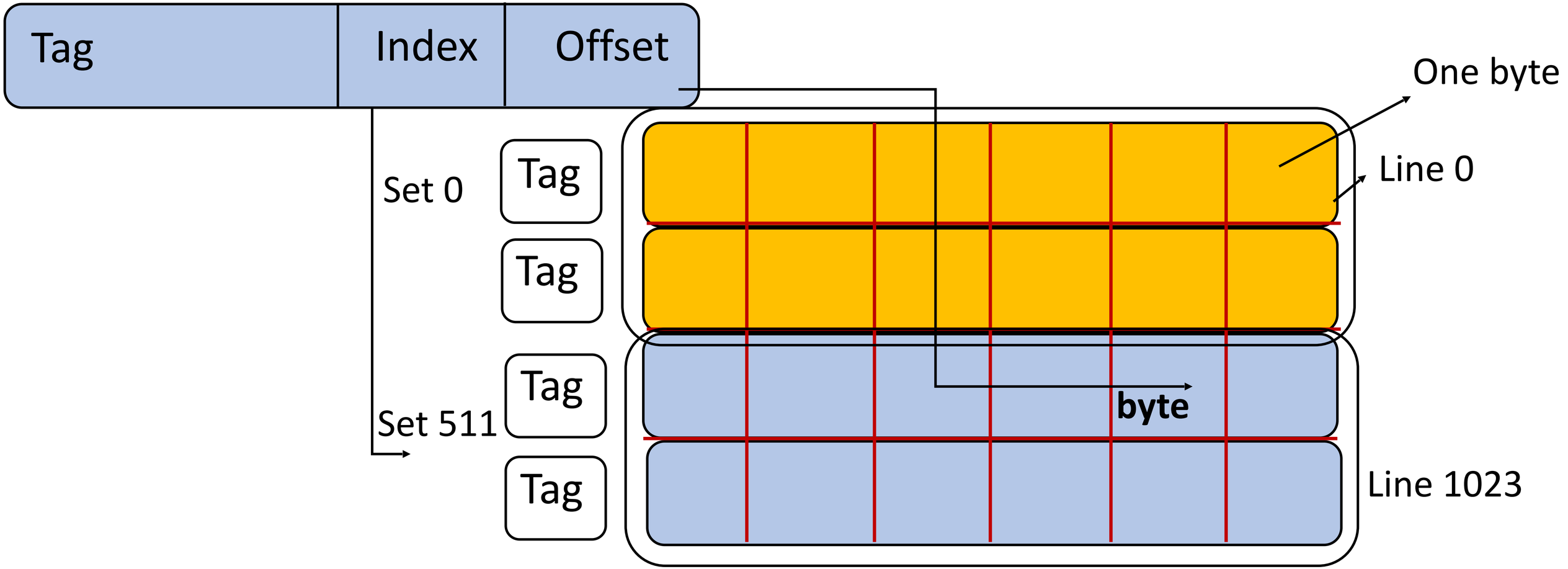
Accessing a cache

- Hit and Miss: you may (hit) or may not (miss) find the data inside the cache…
- When you access for the first time, it is always a miss (compulsory miss)
- When the cache is full, it will be another miss (capacity miss)
- When there is conflict, it can be miss again (conflict miss)

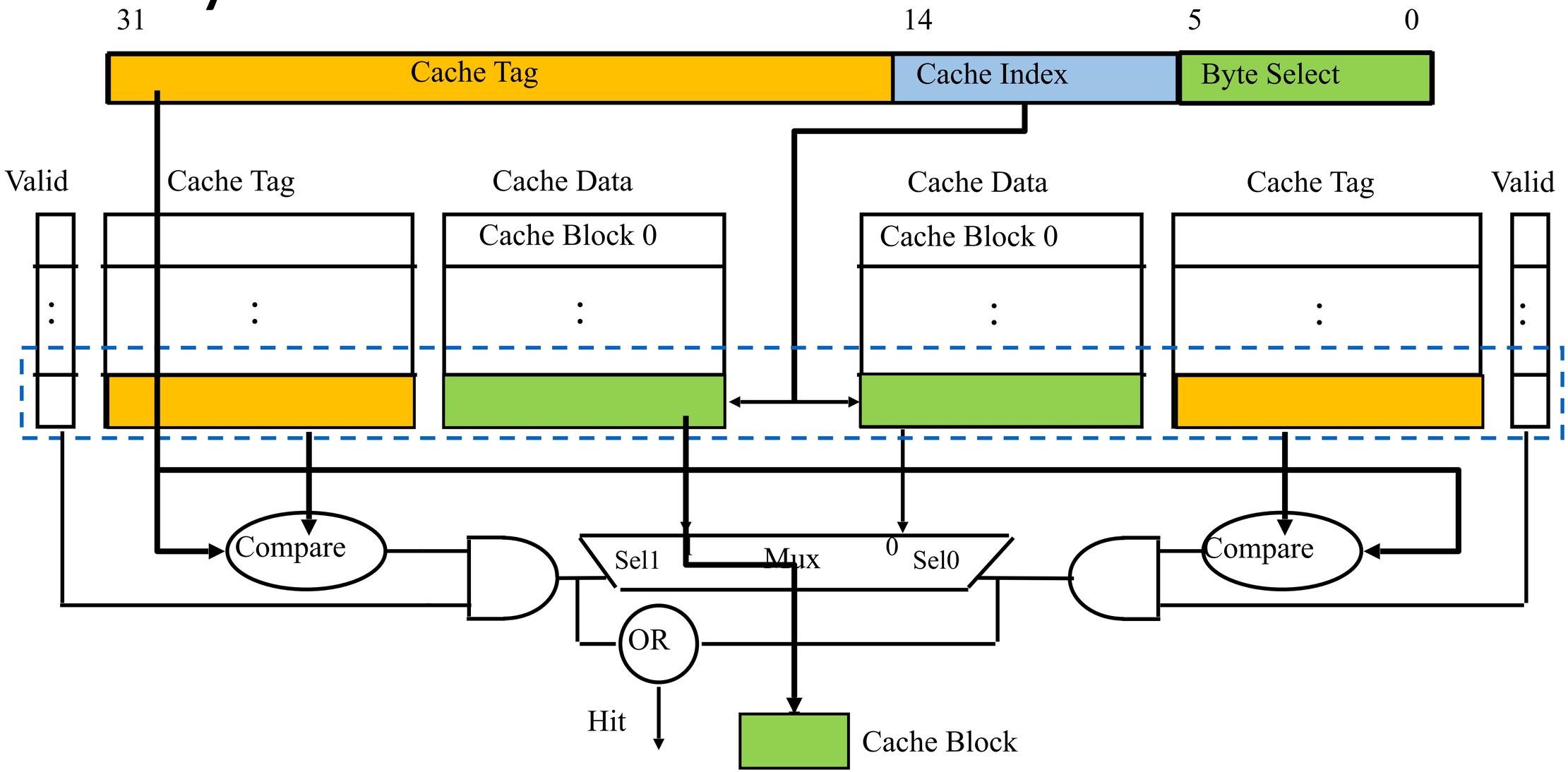
Concept of Valid Bit

- Does the cache word really contains something meaningful?
- Suppose you are just starting to run, at that time even if your tag matches, it might be useless data
- Valid bit: Indicates if a data is stale or useful

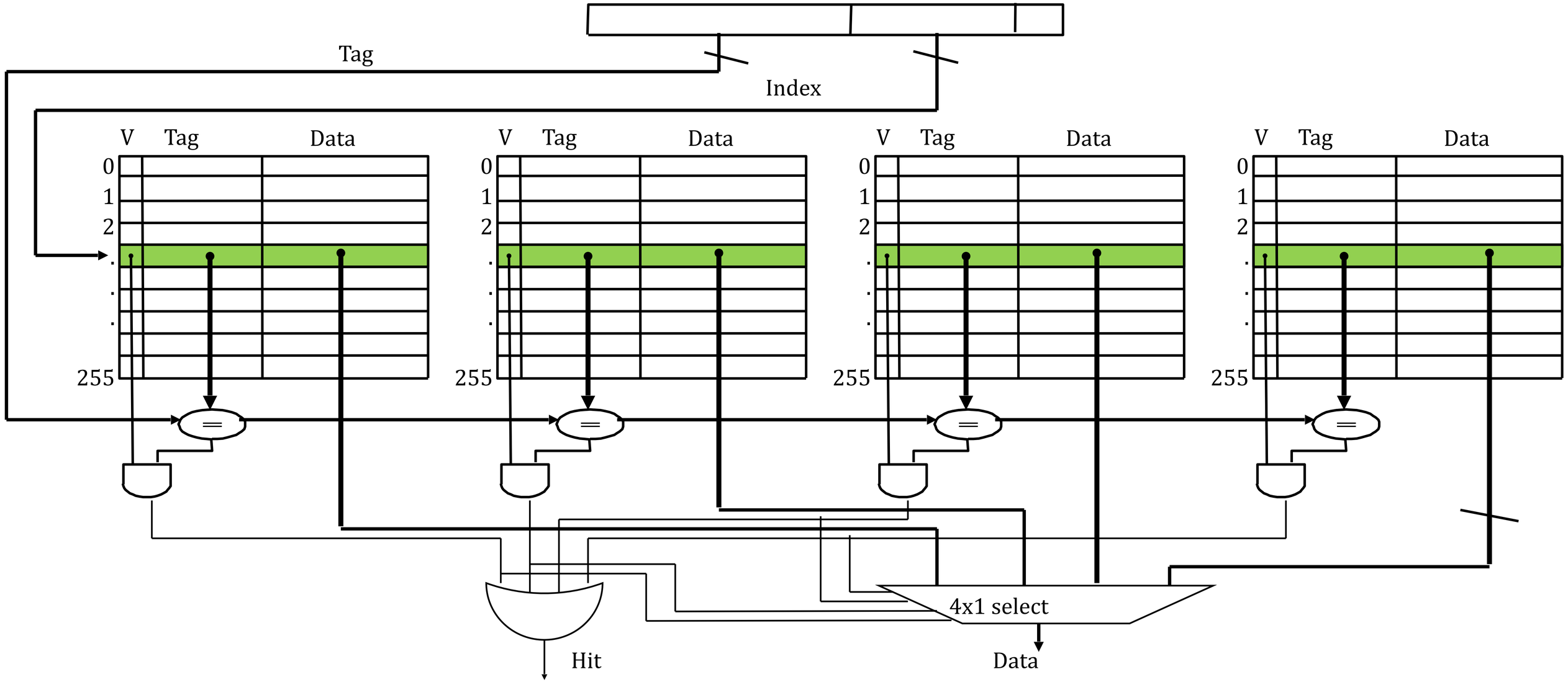
What if we have multiple ways?



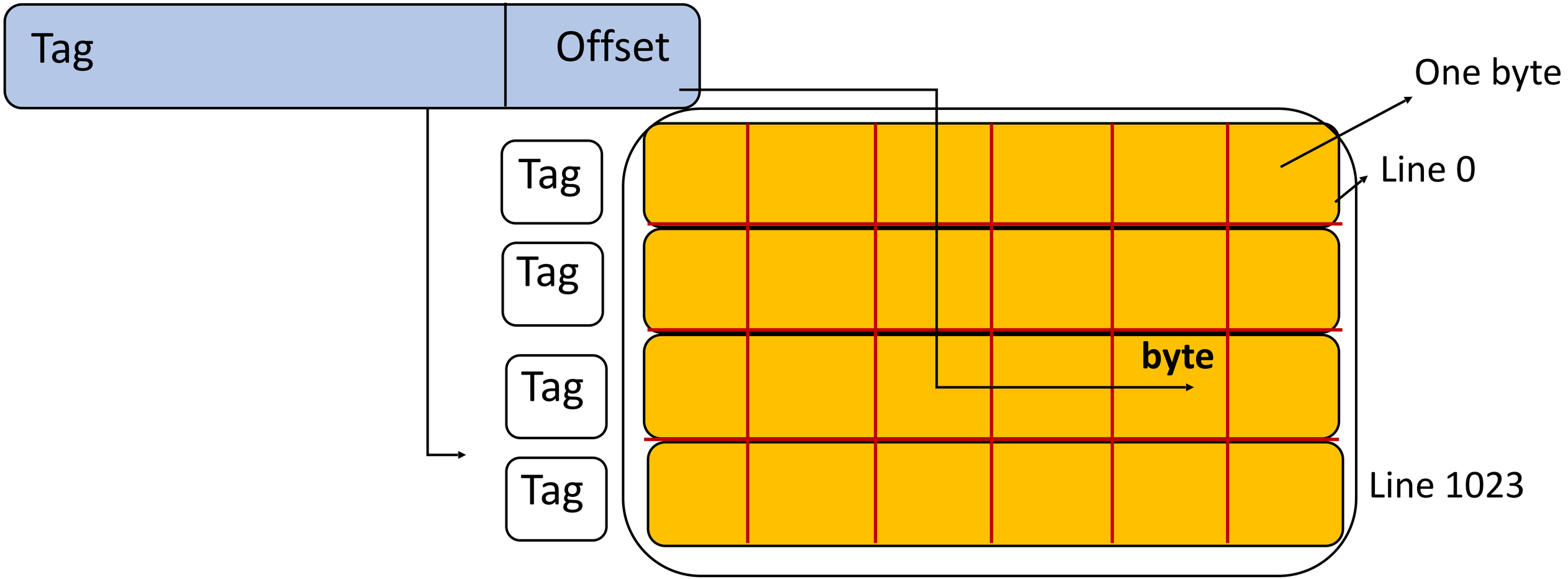
2-way associative in action



4-way associative: Just a better picture



Extreme: One cache, one set, fully associative



Knobs of interest

Line size, associativity, cache size

Tradeoff: latency, complexity, energy/power

Line size = one byte or cache size
Associativity = one or #lines
Cache size = Goal oriented: latency/bandwidth or capacity

Metrics

- Hit time: time taken to handle a hit
- Miss rate: What percentage of cache access results in a miss
- Miss penalty: How much time it takes to serve a miss

On a Miss, Replace a block, which block?

Think of each block in a set having a “priority”
  Indicating how important it is to keep the block in the cache

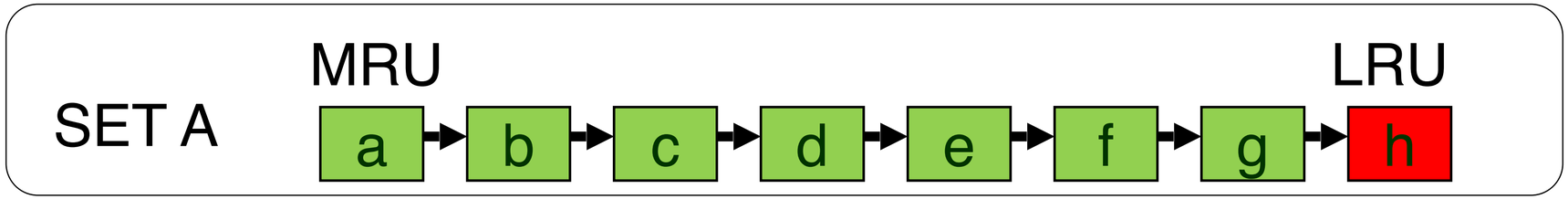
Key issue: How do you determine/adjust block priorities?

Ideally: Belady’s OPT policy, replace the block that will be used furthest in the future. No one knows the future though ☺

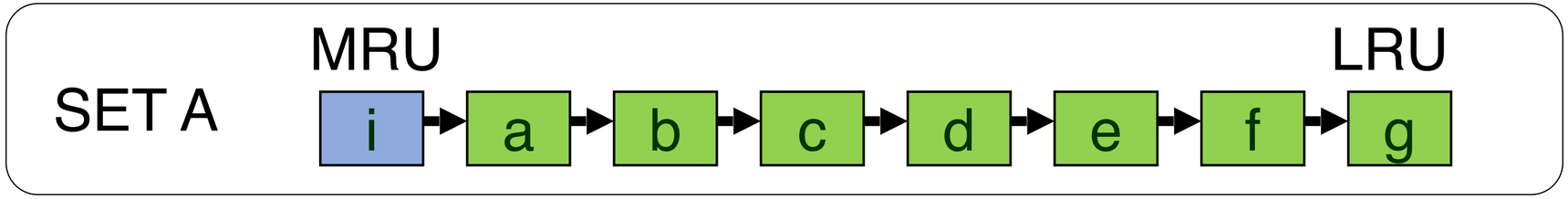
There are three key decisions in a set:
  Insertion, promotion, eviction (replacement)

A simple LRU (Least-Recently-Used) Policy

Cache Eviction Policy: On a miss (block *i*), which block to evict (replace) ?



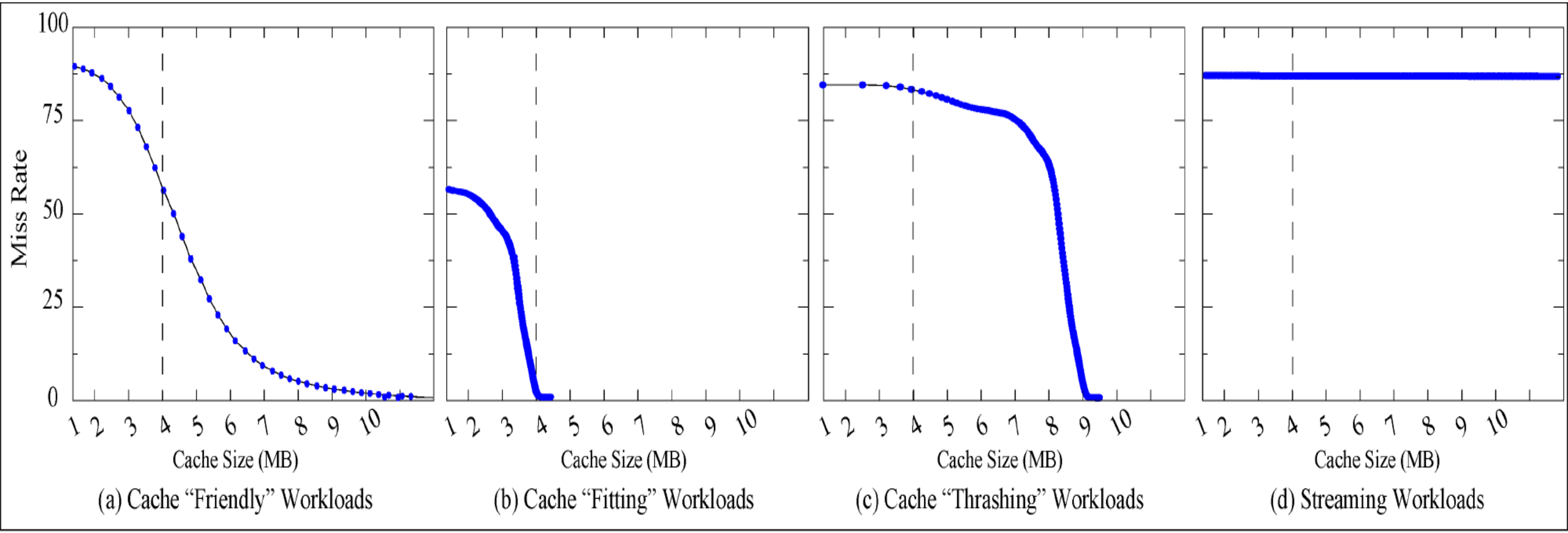
Cache Insertion Policy: New block *i* inserted into MRU.



Cache Promotion Policy: On a future hit (block *i*), promote to MRU

We need priority bits per block. For example, a 16-way cache will need four bit/ block LRU causes thrashing when working set > cache size

Types of Applications



Cache misses once more

Compulsory: first reference to a line (a.k.a. cold start misses)

- *misses that would occur even with infinite cache*

Capacity: cache is too small to hold all data

- *misses that would occur even under perfect (Belady's) replacement policy*

Conflict: misses that occur because of collisions due to line-placement strategy

- *misses that would not occur with ideal full associativity*

Cache knobs and Misses

- Larger cache size
  +reduces capacity and conflict misses?
  - hit time will increase
- Higher associativity
  +reduces conflict misses
  - increase hit time
- Larger line size
  +reduces compulsory misses
  - increases conflict misses and miss penalty

Line size

Too small blocks:

don’t exploit spatial locality well

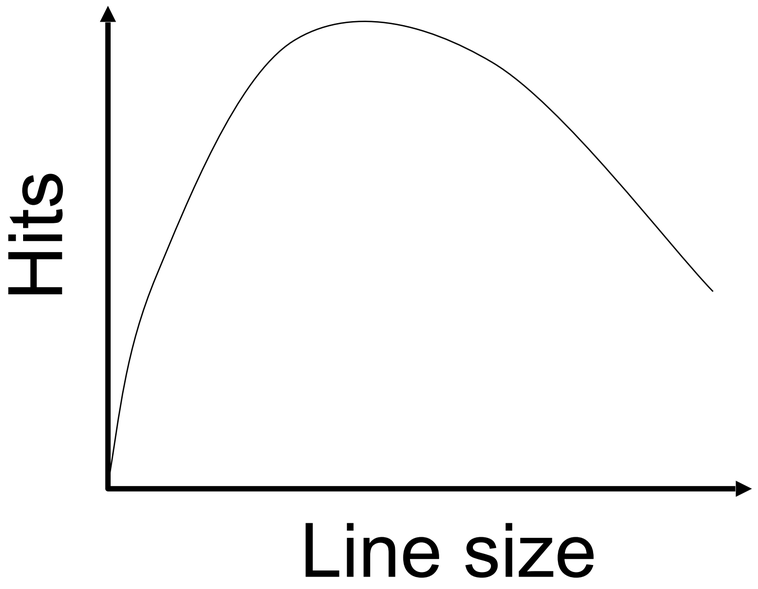
have larger tag overhead

Too large blocks:

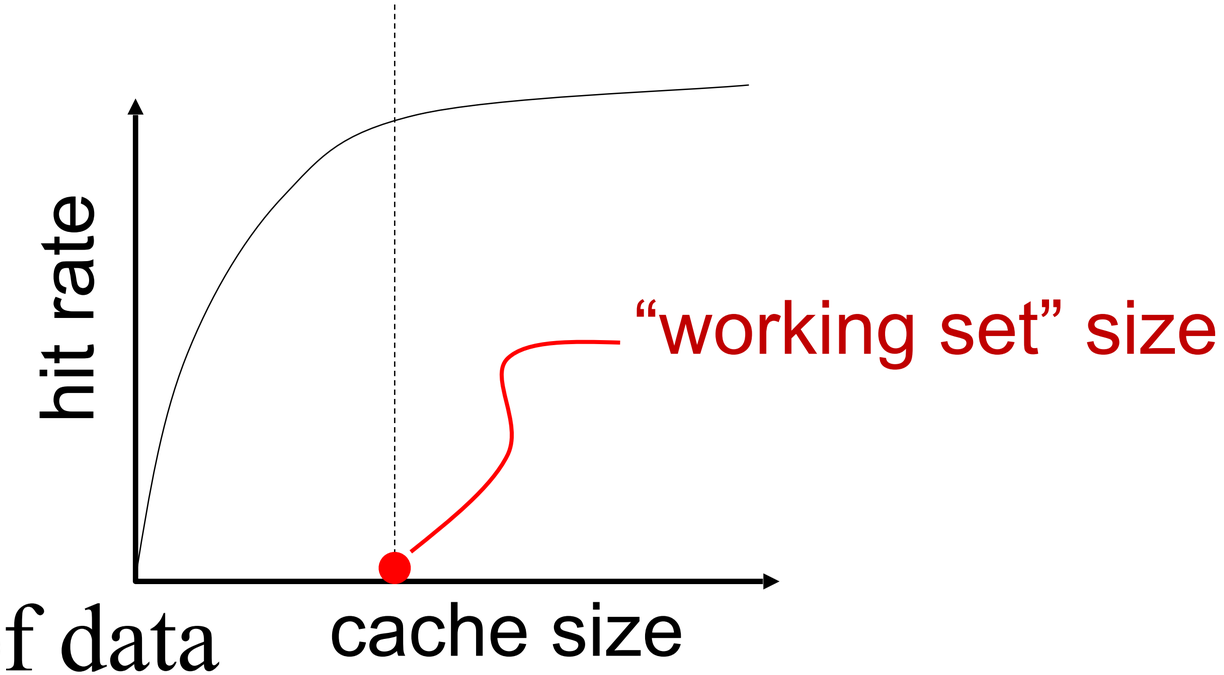
too few total # of blocks

likely-useless data transferred

Extra bandwidth/energy consumed

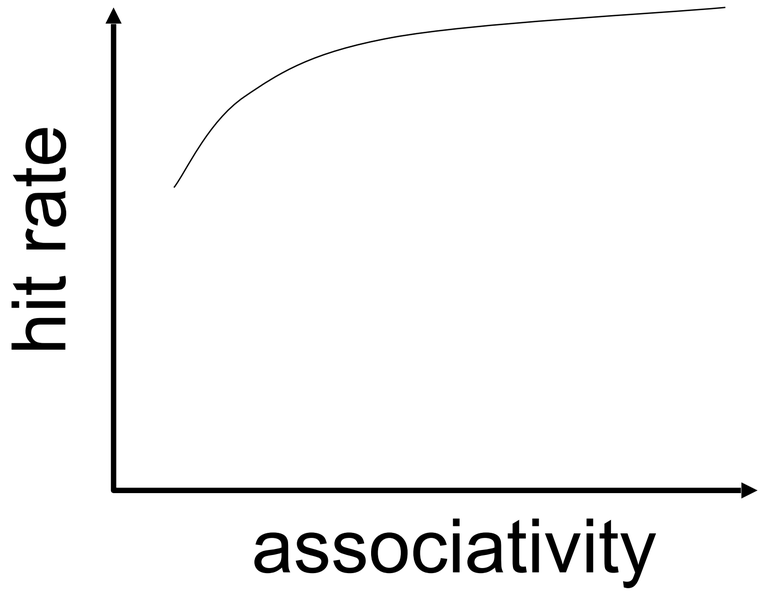


Cache size



Working set: the whole set of data the executing application references within a time interval

Associativity



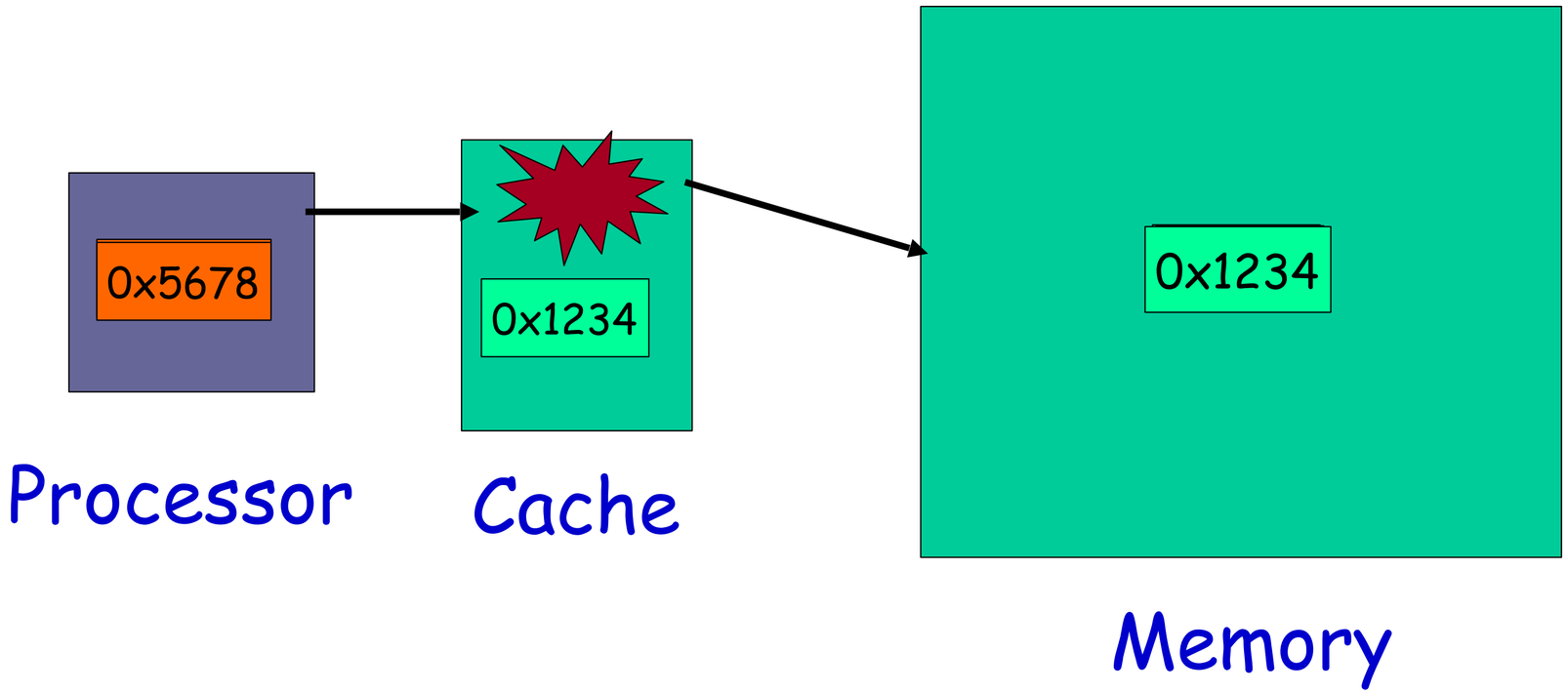
L1 cache: lower associativity, hit time

L3 cache: higher associativity

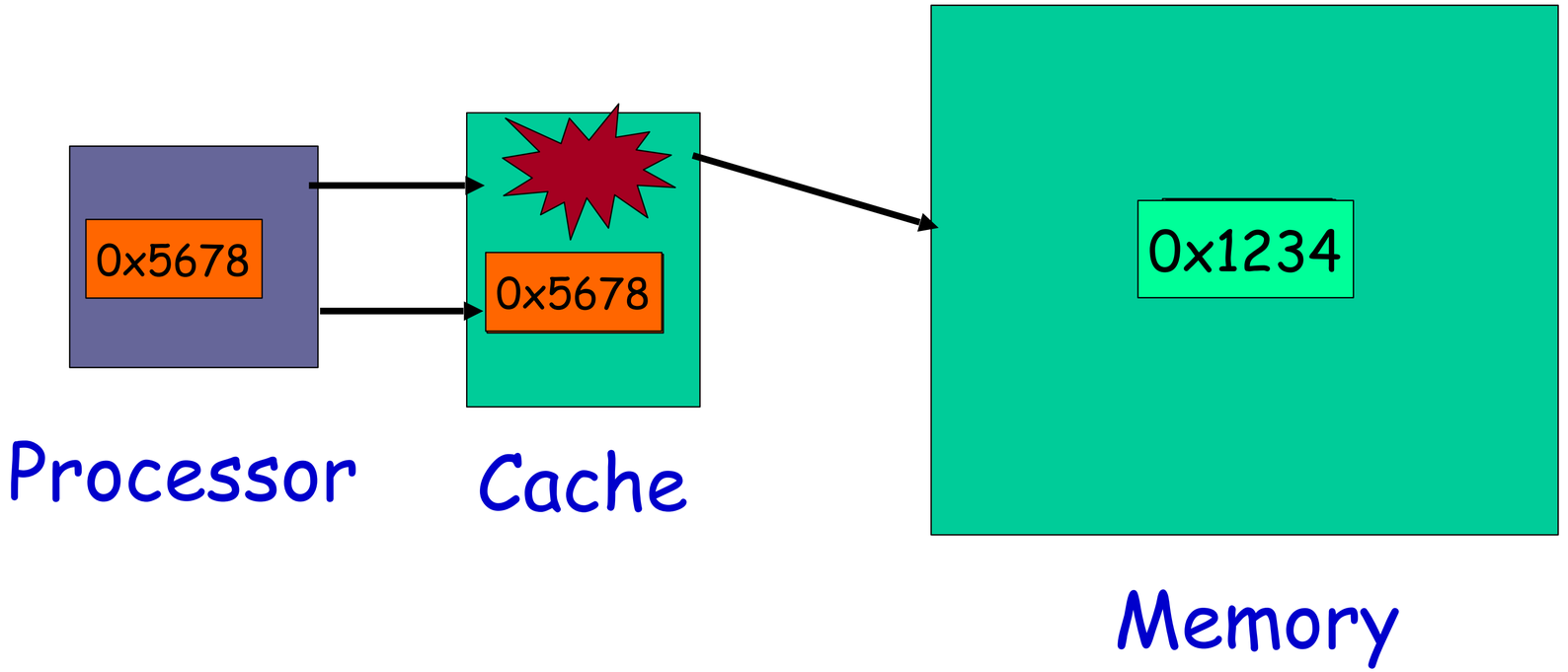
Cache Write Policies

- **Write-through:** Information is written to both the block in the cache and that in memory
- **Write-back:** Information is written back to memory only when a block frame is replaced:
  - **Uses a "dirty" bit to indicate whether a block was actually written to,**
  - Saves unnecessary writes to memory when a block is "clean"

Write-Through Policy



Write back Policy



Trade-offs

- **Write back:**
  - Faster because writes occur at the speed of the cache, not the memory.
  - Faster because multiple writes to the same block is written back to memory only once, uses less memory bandwidth.
- **Write through:**
  - Easier to implement

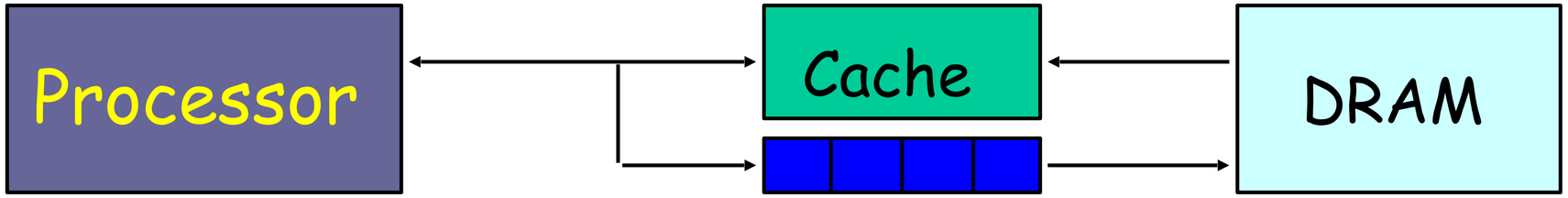
Write Allocate, No-write Allocate

- What happens on a write miss?
  - On a read miss, a block has to be brought in from a lower level memory
- **Two options:**
  - **Write allocate**: A block allocated in cache.
  - **No-write allocate**: No block allocation, but just written to main memory.

Write Allocate, No-write Allocate

- In no-write allocate,
  - Only blocks that are read from can be in cache.
  - Write-only blocks are never in cache.
- **Can either be used with write through and write back?**
  - Write-allocate used with write-back
  - No-write allocate used with write-through
- **Why does this make sense?**

Write Buffer



Write Buffer

- **Processor:**
  - writes data into cache and write buffer
- **Memory controller:**
  - writes contents of the buffer to memory
- **Write buffer is a FIFO structure:**
  - Typically 4 to 8 entries
  - Desirable: Occurrence of Writes << DRAM write cycles
- **Memory system designer's nightmare:**
  - Write buffer saturation (i.e., Writes ~ DRAM write cycles)

Cache Performance Parameters

- AMAT is largely determined by:
  - **Cache miss rate:** number of cache misses divided by number of accesses.
  - **Cache hit time:** the time between sending address and data returning from cache.
  - **Cache miss penalty:** the extra processor stall cycles caused by access to the next-level cache.

Average Memory Access Time

- **AMAT:** The average time it takes for the processor to get a data item it requests.
  - Can vary considerably for different memory configurations due to various attributes of the memory hierarchy.
- AMAT can be expressed as:

$$\text{AMAT} = \text{Cache hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Impact of Memory System on Processor Performance

CPU Performance $_{\text{with Memory Stall}}$ = CPI $_{\text{without stall}}$ + Memory Stall CPI

Memory Stall CPI

= Miss per inst × miss penalty

= % Memory Access/Instr × Miss rate × Miss Penalty

Example: Assume 20% memory acc/instruction, 2% miss rate, 400-cycle miss penalty. How much is memory stall CPI?

Memory Stall CPI= 0.2*0.02*400=1.6 cycles

CPU Performance with Memory Stall

$$\text{CPU Performance}_{\text{with Memory Stall}} = \text{CPI}_{\text{without stall}} + \text{Memory Stall CPI}$$

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{CPI}_{\text{mem\_stall}}\right) \times \text{Cycle Time}$$

$$\text{CPI}_{\text{mem\_stall}} = \text{Miss per inst} \times \text{miss penalty}$$

$$\text{CPI}_{\text{mem\_stall}} = \text{Memory Inst Frequency} \times \text{Miss Rate} \times \text{Miss Penalty}$$

Performance Exercise 1

- Suppose:
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1 (including control stalls)
  - 50% arith/logic, 30% load/store, 20% control
  - 10% of data memory operations incur 50 cycles miss penalty
  - 1% of instruction memory operations also incur 50 cycles miss penalty
- Compute CPI with cache miss and AMAT.

Solution

- CPI = ideal CPI + average stalls per instruction = 1.1(cycles/ns) + [0.30 (DataMops/ins) x 0.10 (miss/DataMop) x 50 (cycle/miss)] +[1 (InstMop/ins) x 0.01 (miss/InstMop) x 50 (cycle/miss)]
  = (1.1 + 1.5 + .5) cycle/ins = 3.1
- Combined AMAT (normalised)=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54

Exercise 2

- Assume 20% Load/Store instructions
- Assume CPI without memory stalls is 1
- Cache hit time = 1 cycle
- Cache miss penalty = 100 cycles
- Miss rate = 1%
- What is:
  - Stall cycles per instruction?

Exercise 2:Solution

- Average memory accesses per instruction = 1.2
- Stall cycles = 1.2 cycles
  - Instruction misses per instruction: 0.01×100=1.0 cycles / instruction.
  - Data misses per instruction: 0.20×0.01×100=0.2 cycles/instr

Miss Rate Vs. MPKI

Miss rate

Misses per kilo instructions (MPKI)

But Why?

Miss Rate Vs. MPKI

Miss rate

MPKI

$$\text{Miss Rate} = \frac{\text{\# of cache misses}}{\text{\# of cache accesses}}$$

$$\text{MPKI} = \frac{\text{\# of cache misses}}{\text{\# of instructions executed}} \times 1000$$

What if you have only one memory access?

MPKI is often used for reporting performance evaluation results on benchmarks, as it takes care of the frequency of memory instructions real benchmarks

Memory Hierarchy Optimizations

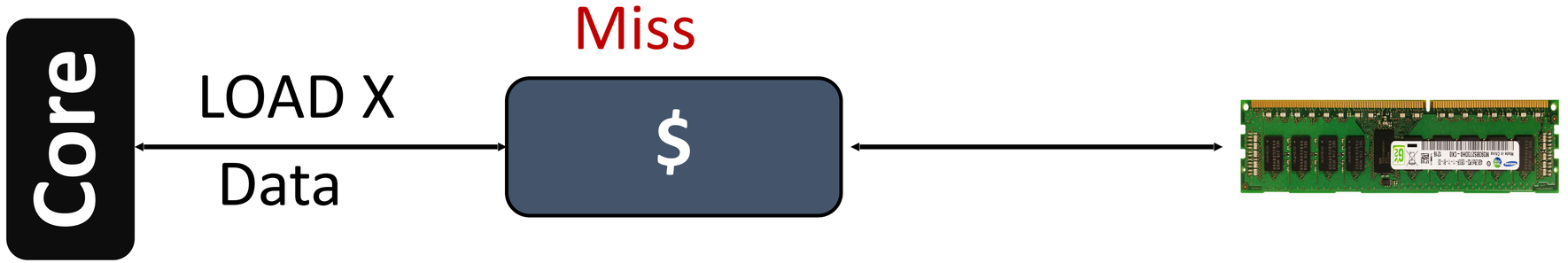
Reducing Miss Rates

- Techniques:
  - Larger block size
  - Larger cache size
  - Higher associativity

Reducing Miss Penalty

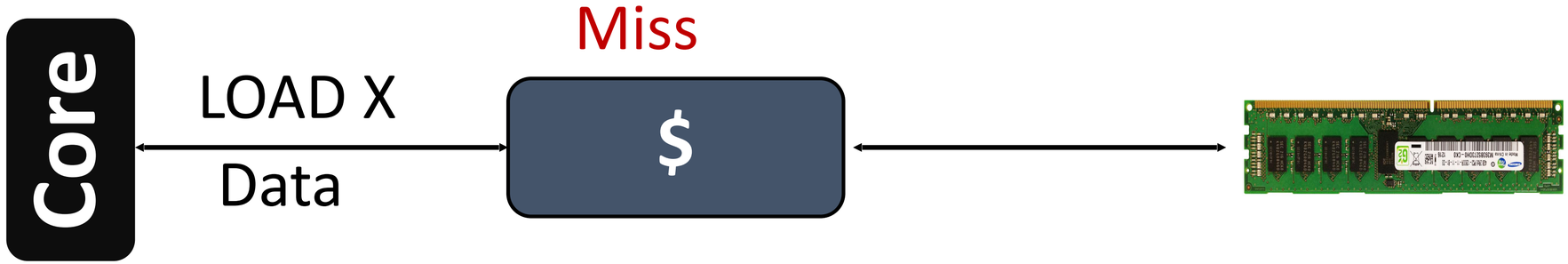
- Techniques:
  - Critical word first
  - Multilevel caches

On a miss: Critical Word first



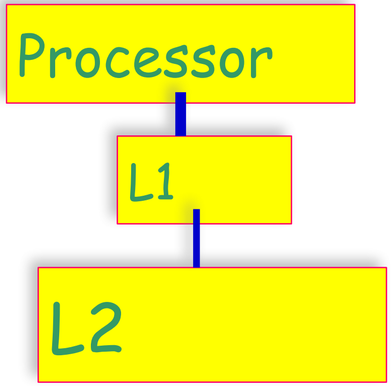
On a miss, respond with the word/byte requested to the core so that core can continue while fetching the rest of the block

On a miss: Early Restart



On a miss, fetch the words/bytes in normal order, but as soon as the requested word/byte of the block arrives, send it to the core.

Multi-Level Cache



- Add a second-level cache.
- L2 Equations:

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

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

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

Multi-Level Cache: Some Definitions

- **Local miss rate:**
  - **M**isses in this cache divided by the total number of memory accesses to this cache (e.g. Miss rate_{L2})
- **Global miss rate:**
  - Misses in this cache divided by the total number of memory accesses generated by the CPU
- L1 Global miss rate = L1 Local miss rate

Global vs. Local Miss Rates

- At lower level caches (L2 or L3), global miss rates provide more useful information:
  - **Indicate how effective is the cache in reducing AMAT.**
  - **Who cares if the miss rate of L3 is 50% as long as only 1% of processor memory accesses ever benefit from it?**

Performance Improvement Due to L2 Cache: Exercise

Assume:

- For 1000 memory instructions:
  - 40 misses in L1,
  - 20 misses in L2
- L1 hit time: 1 cycle,
- L2 hit time: 10 cycles,
- L2 miss penalty=100 cycles
- 1.5 memory references per instruction
- Assume ideal CPI=1.0

Find: Local miss rate of L2, AMAT, stall cycles per instruction, and those for case without L2 cache.

Solution

- **With L2 cache:**
  - Local miss rate of L2 = 50%
  - AMAT=1+(40/1500)X(10+50%X100)=1.92
  - Average Memory Stalls per Instruction = (1.92-1.0)x1.5=2.84
- **Without L2 cache:**
  - AMAT=1+40/1500X100=3.7
  - Average Memory Stalls per Inst=(3.7-1.0)x1.5=4.1
- **Perf. Improv. with L2** =(4.1+1)/(2.84+1)=32%

Note: We have not distinguished reads and writes. Access L2 only on L1 miss, i.e. write back cache…

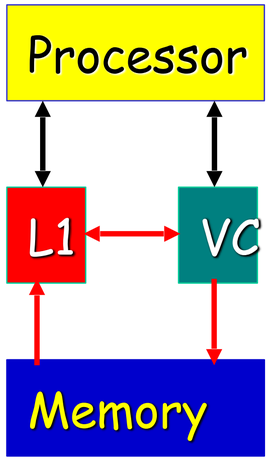
Multilevel Cache: Some Issues

- The speed (hit time) of L1 cache affects the clock rate of CPU:
  - Speed of L2 cache only affects miss penalty of L1.
- **Inclusion Policy:**
  - Many designers keep L1 and L2 block sizes the same.
  - Otherwise on a L2 miss, several L1 blocks may have to be invalidated.
- **Multilevel Exclusion:**
  - L1 data never found in L2 ---Saves some L2 space
  - AMD Athlon follows exclusion policy .

Reducing Miss Penalty: Victim Cache

- How to combine fast hit time of direct mapped cache:
  - yet avoid conflict misses?
- **Add a fully associative buffer (victim cache) to keep data discarded from cache.**
- 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 machines.
- AMD uses 8-entry victim buffer.

Victim Cache [Jouppi'90]



Victim Cache Organization

- A small, fully associative structure
- Effective in direct-mapped caches
- Whenever a line is displaced from L1 cache, it is loaded into VC
- Processor checks both L1 and VC simultaneously
- **Swap** data between VC and L1 if L1 misses and VC hits
- When data has to be evicted from VC, it is written back to memory

Reducing Miss Penalty or Miss Rates via Parallelism

- Techniques:
  - Non-blocking caches
  - Prefetching

Non-blocking Caches

- Non-blocking cache:
  - –Allow data cache to continue to serve other requests during a miss.
  - –Meaningful only with out-of-order execution processor.
  - **–Requires multi-bank memories.**
  - **–Pentium Pro allows 4 outstanding memory misses.**

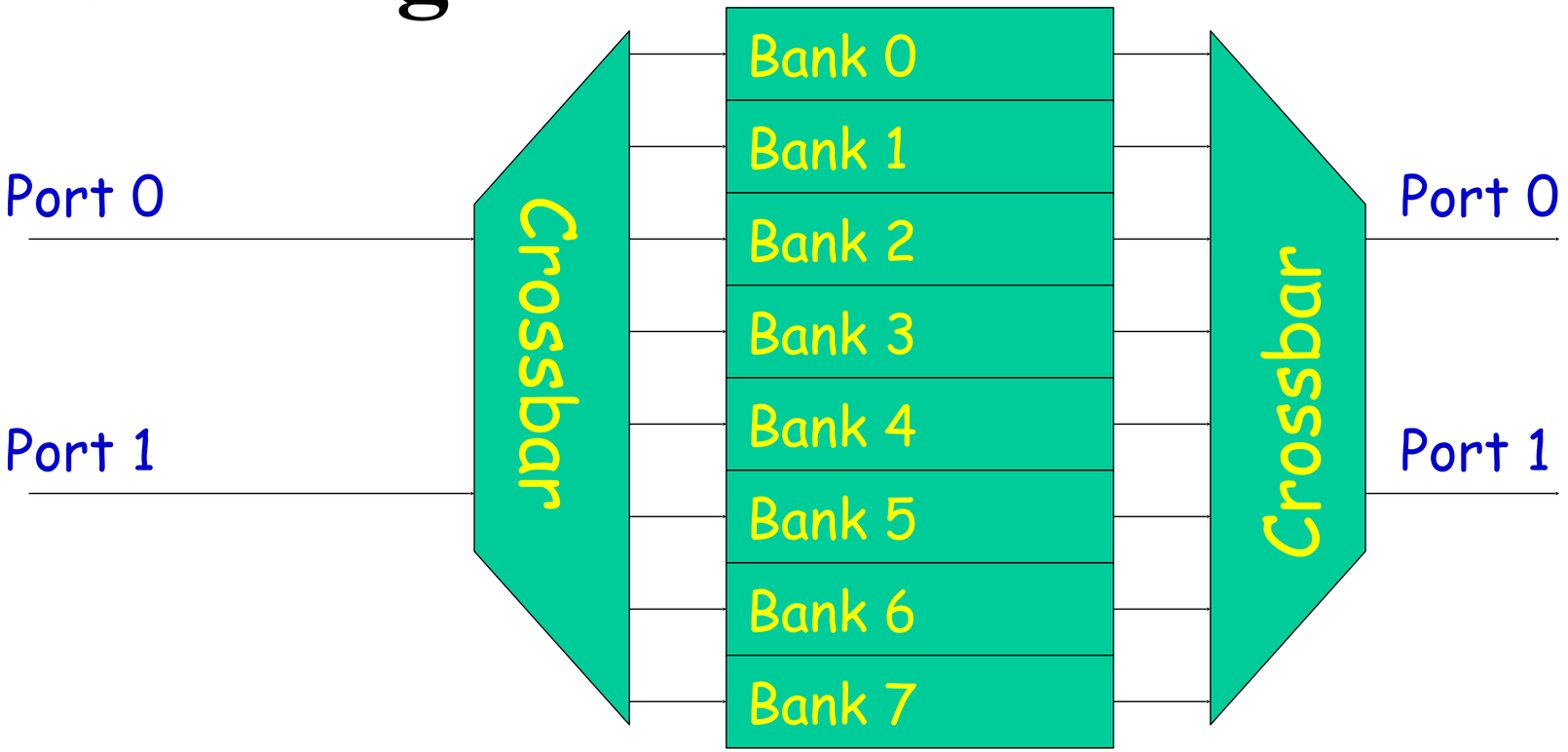
Non-blocking Caches

- Hit under miss reduces the effective miss penalty by working during miss.
- "Hit under multiple 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.

Non-Blocking Caches

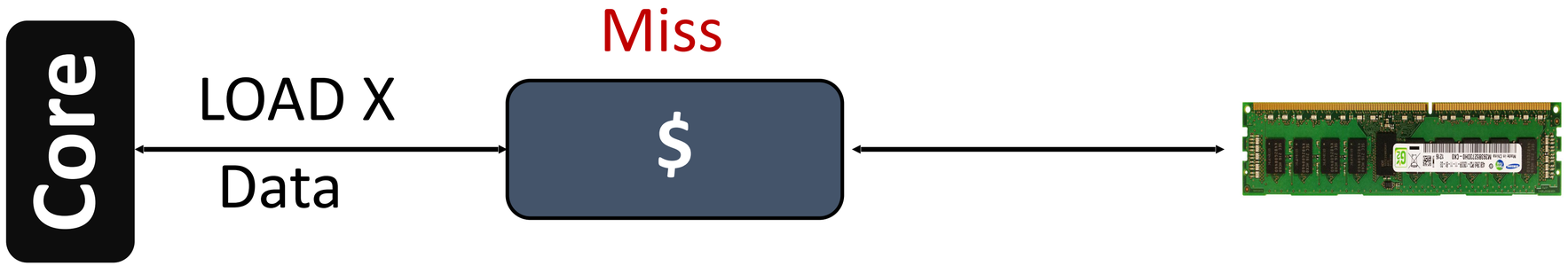
- Multiple memory controllers:
  - Allow memory banks to operate almost independently.
  - Each bank needs separate address lines and data lines.

Multi-banking



- Used in Intel Pentium (8 banks)
- Need routing network
- Must deal with bank conflicts

Core, cache, DRAM interaction

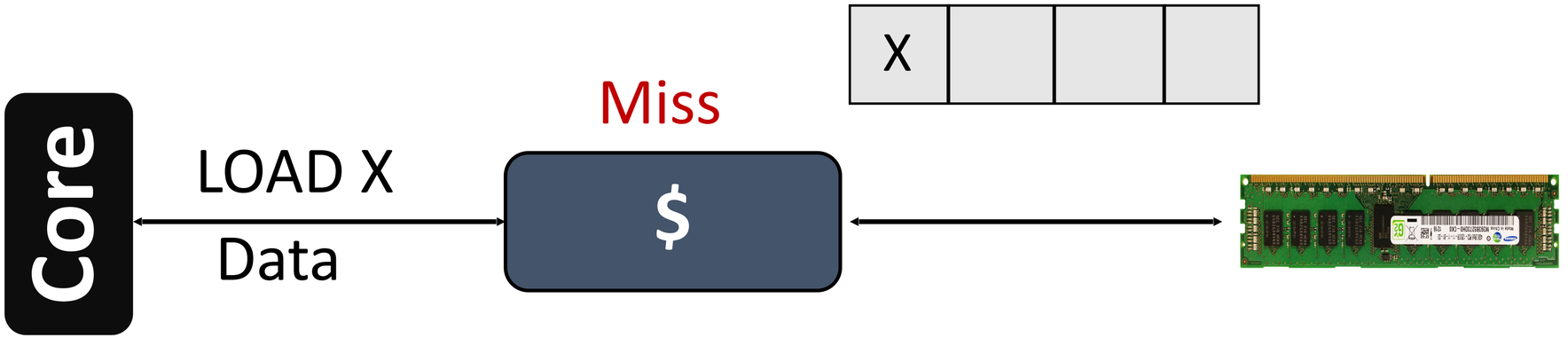


100s of cycles

I am an out-of-order core

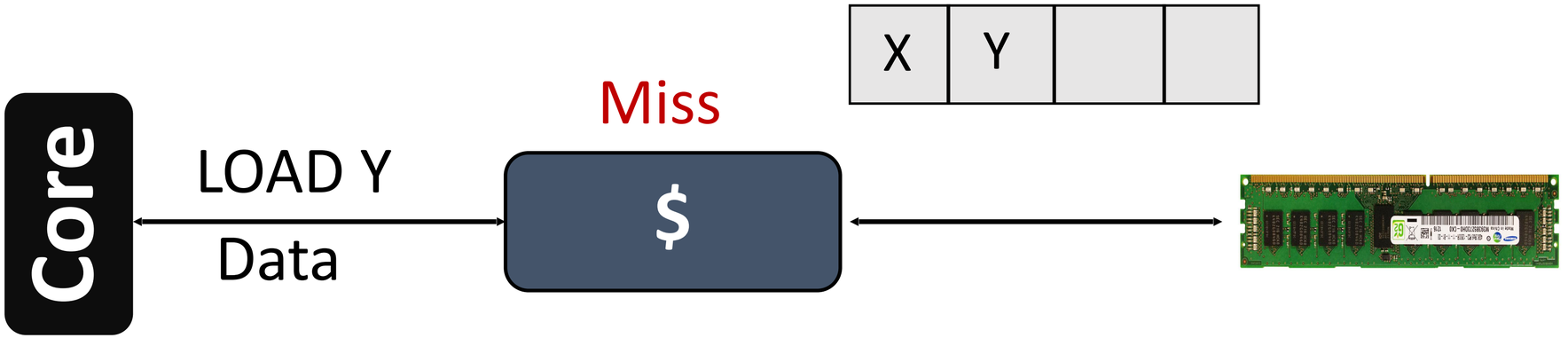
One cache miss and can’t handle anymore misses

MSHRS (Miss-status holding registers)

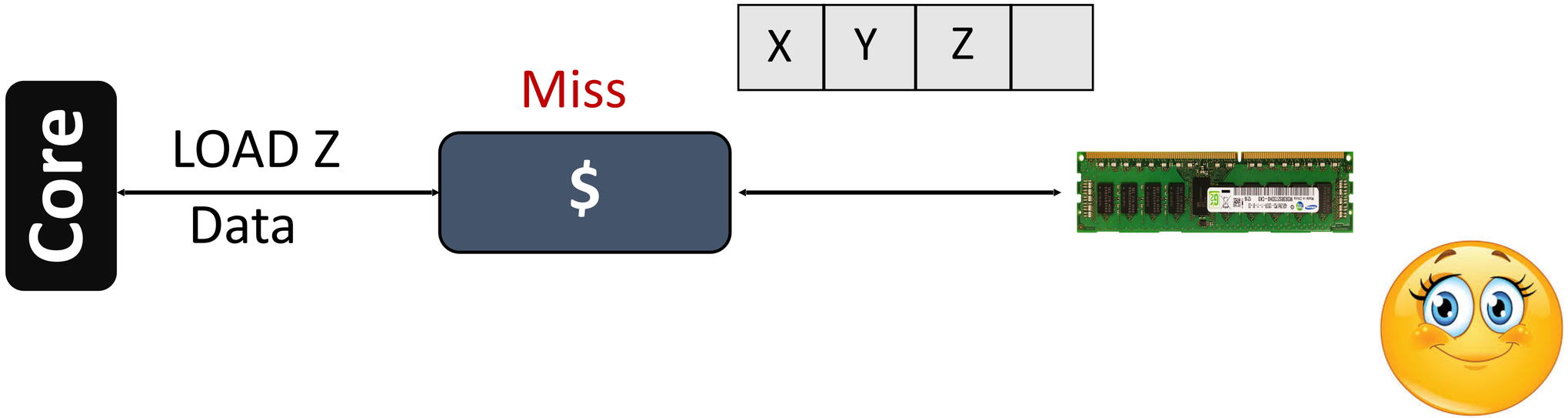


Non-blocking cache

MSHRS (Miss-status holding registers)

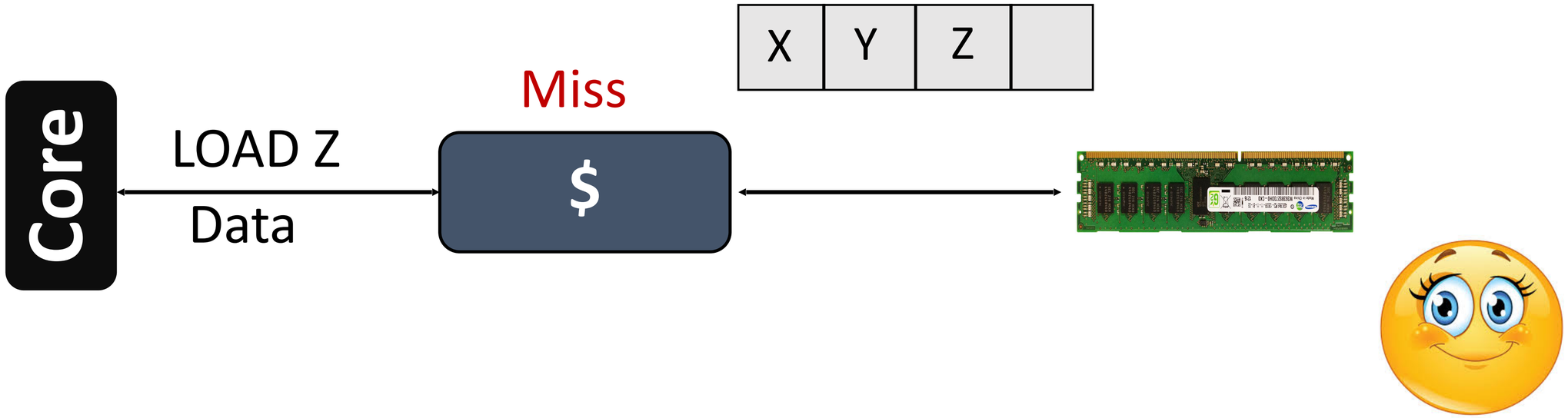


MSHRS (Miss-status holding registers)



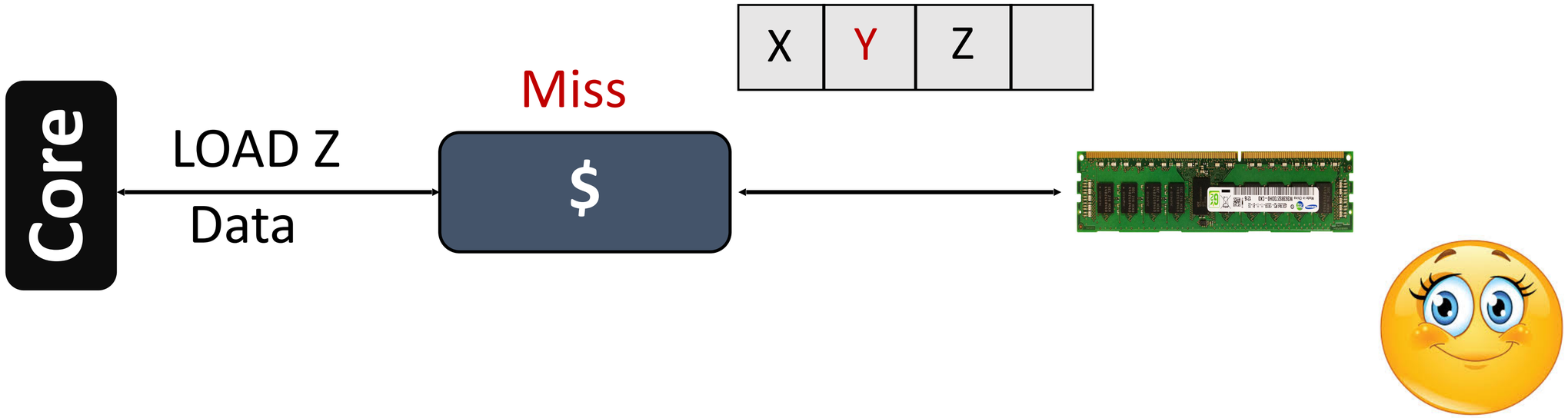
K-entry MSHR allows K outstanding misses: provides memory-level parallelism

MSHRS (Miss-status holding registers)



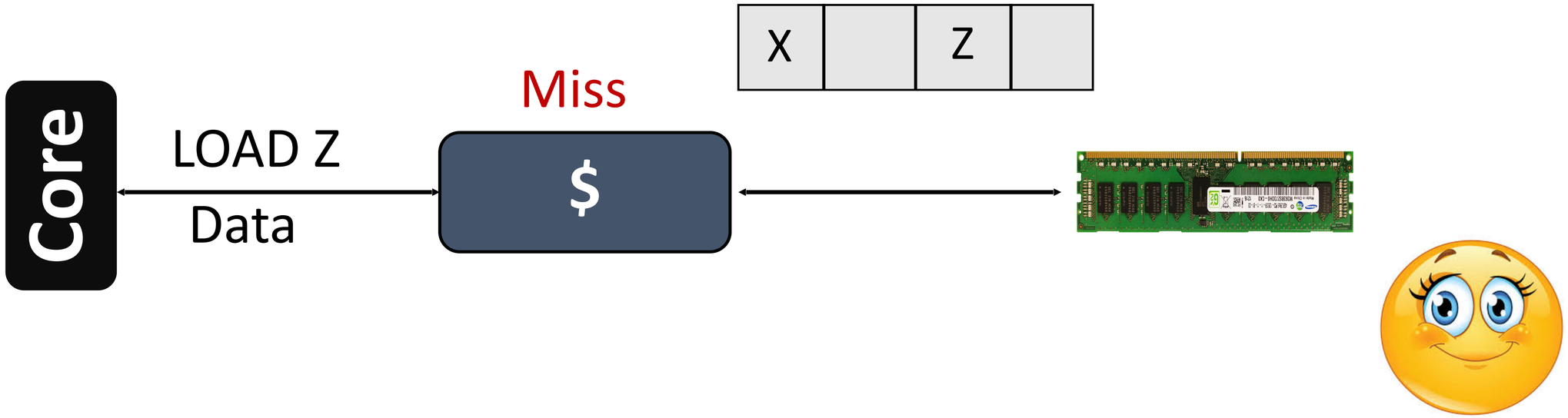
DRAM response time is not constant: can take from 60 cycles to 1000s of cycles (on a multi-core system).

MSHRS (Miss-status holding registers)



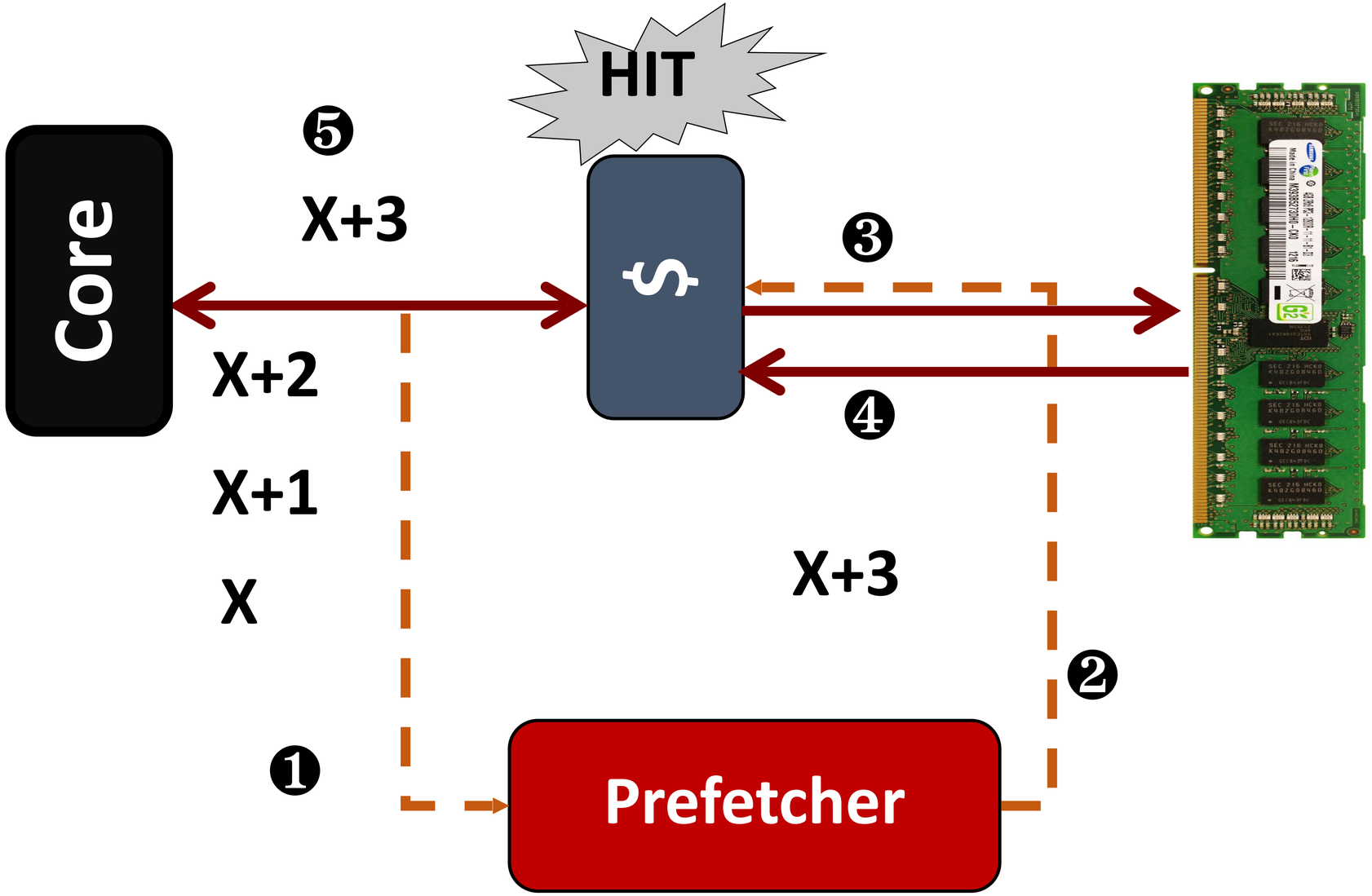
DRAM response time is not constant: can take from 60 cycles to 1000s of cycles (on a multi-core system).

MSHRS (Miss-status holding registers)



DRAM response time is not constant: can take from 60 cycles to 1000s of cycles (on a multi-core system).

Hardware Prefetching



10K Feet View

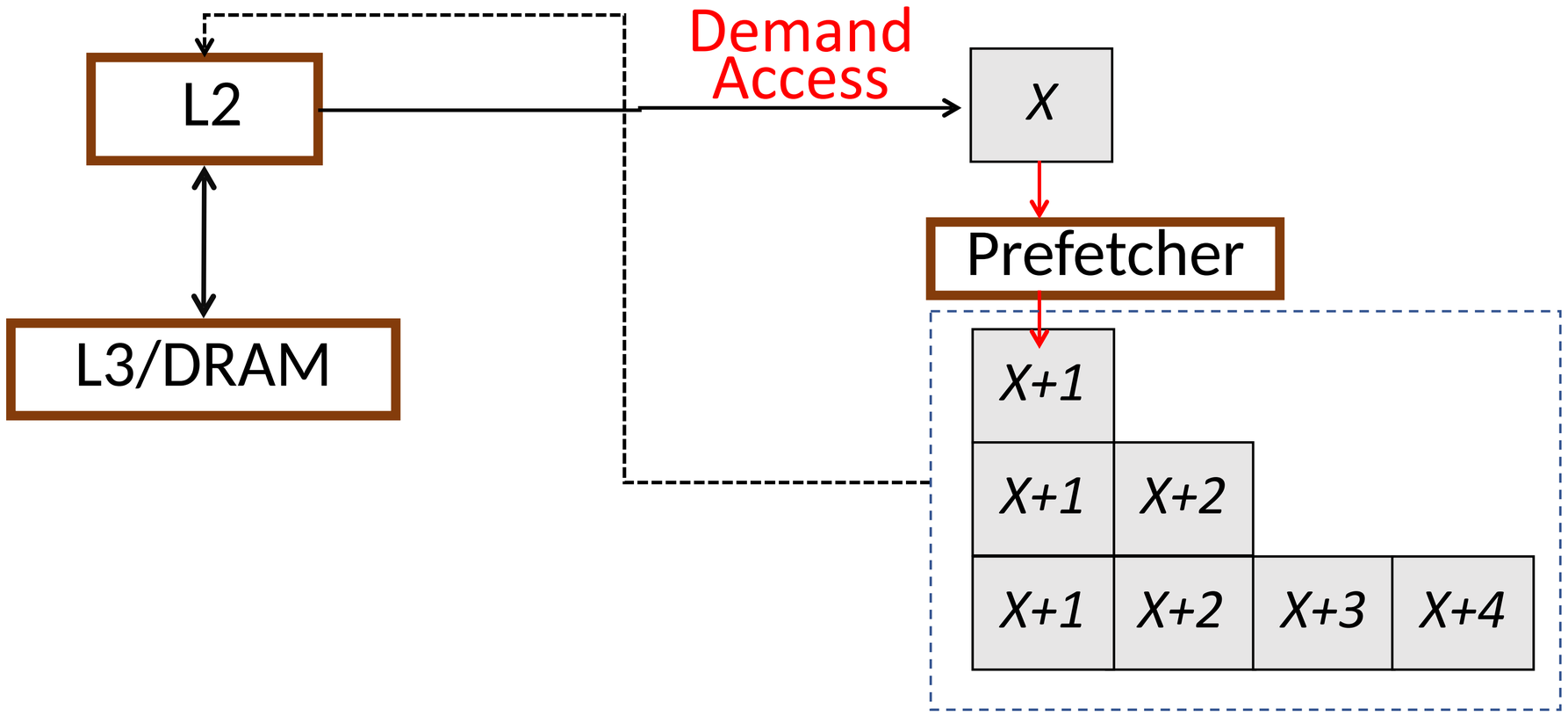
What?
Latency-hiding technique - Fetches data before the core demands.

Why?
Off-chip DRAM latency has grown up to 400 to 800 cycles.

How?
By observing/predicting the demand access (LOAD/STORE) patterns.

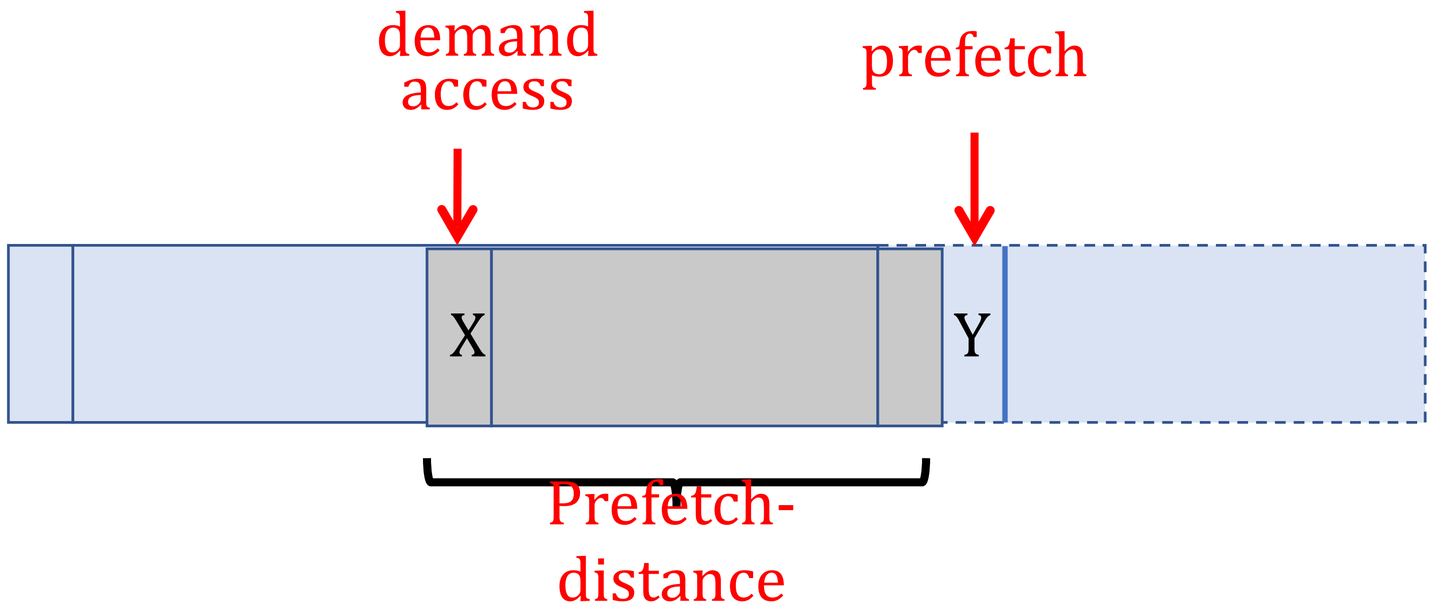
Prefetch Degree

Prefetch Degree: Number of prefetch requests to issue at a given time.



Prefetch Distance

Prefetch Distance: How far ahead of the demand access stream are the prefetch requests issued?



Y = X + 4

Y = X + 8

Y = X + 16

Next-line prefetcher

Next Line: Miss to cache block X , prefetch X+1. Degree=1, Distance=1

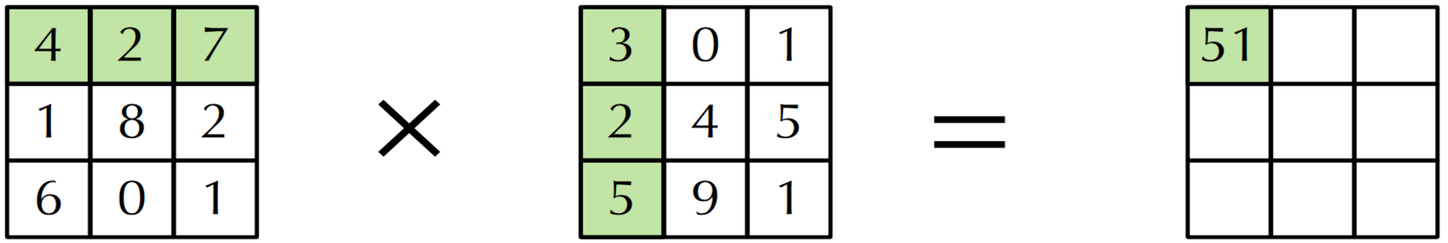
Works well for L1 Icache and L1 Dcache.

Compiler Level Optimization for Speedup

Matrix Multiplication: 101

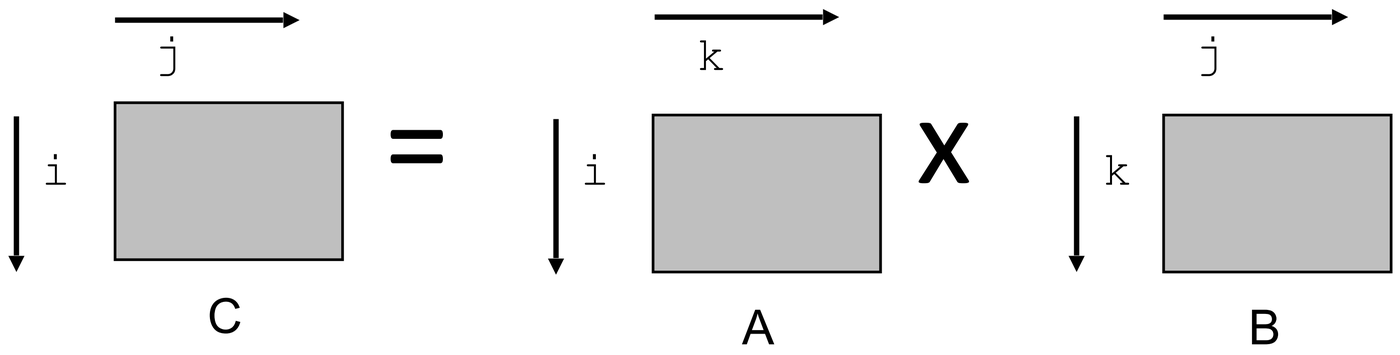
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

$$4\times3 + 2\times2 + 7\times5 = 51$$



Miss Rate analysis

- Assume:
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



Effect of Cache Layout

C arrays allocated in row-major order

- each row in contiguous memory locations

Stepping through columns in one row:

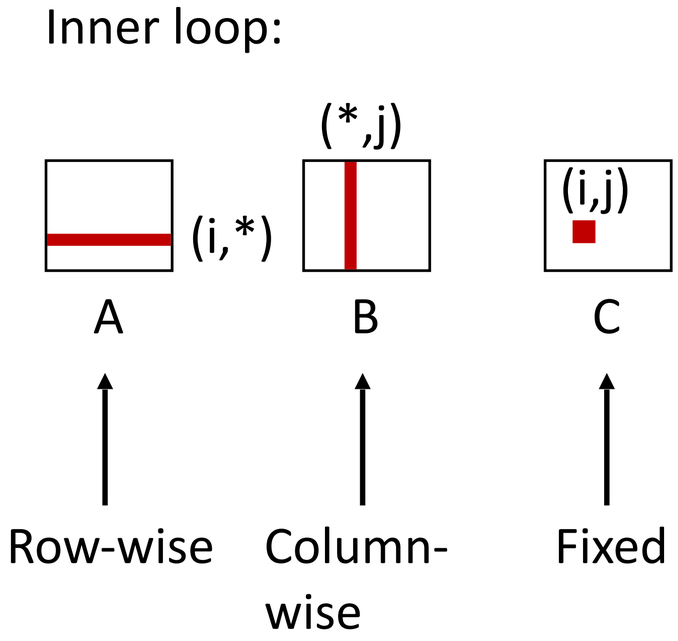
- **for (i = 0; i < N; i++)**
  sum += a[0][i];
- accesses successive elements
- if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
  - miss rate = sizeof(a_{ij}) / B

Stepping through rows in one column:

- **for (i = 0; i < N; i++)**
  sum += a[i][0];
- accesses distant elements
- no spatial locality!
  - miss rate = 1 (i.e. 100%)

Effect of loop order (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

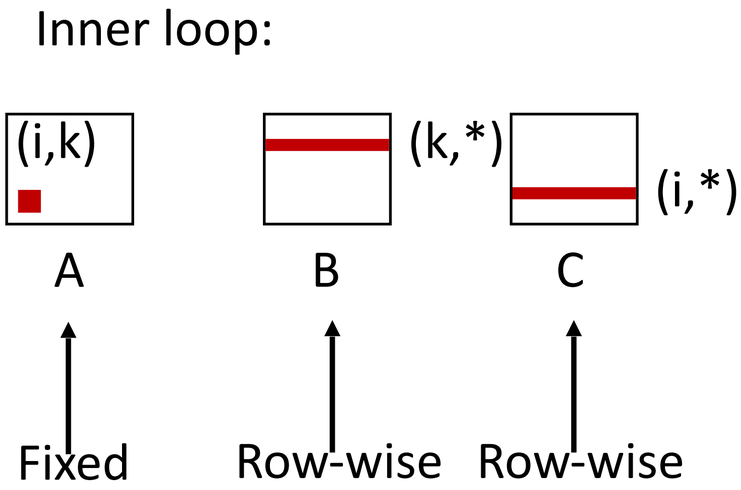


Miss rate for inner loop iterations:

A	B	C
0.25	1.0	0.0

Effect of loops (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

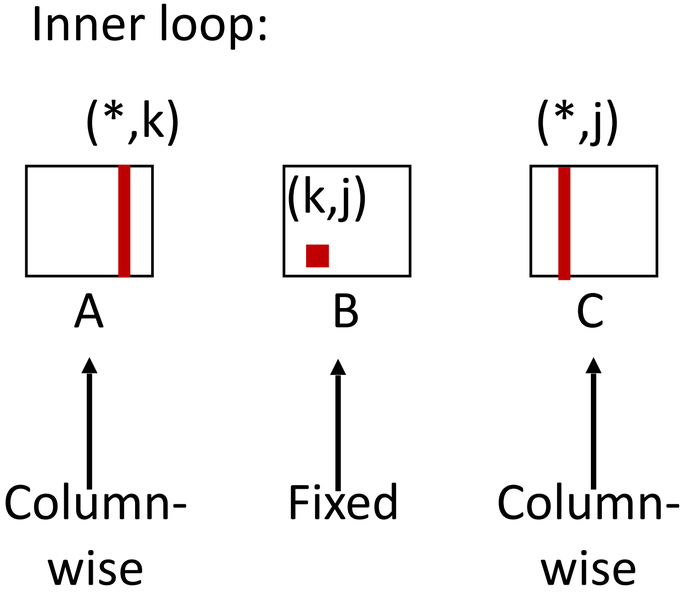


Miss rate for inner loop iterations:

A	B	C
0.0	0.25	0.25

Effect of loops (jki)

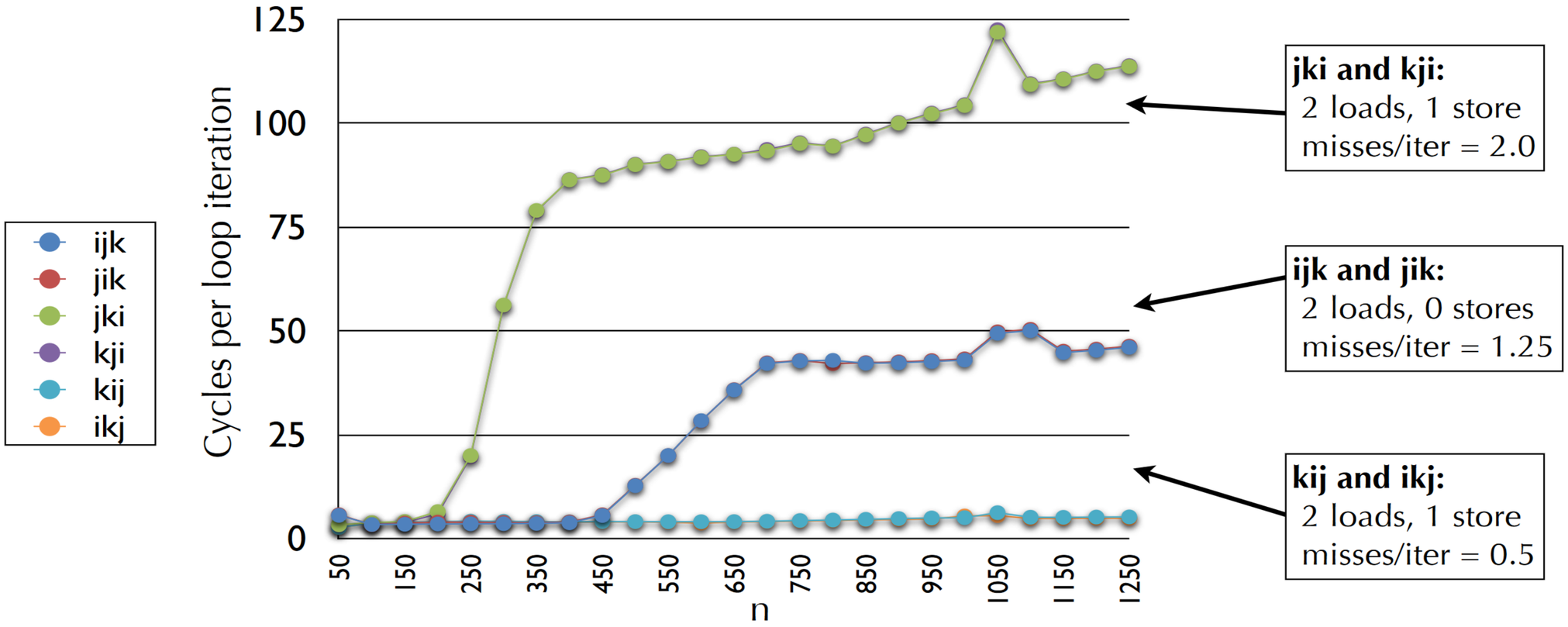
```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Miss rate for inner loop iterations:

A	B	C
1.0	0.0	1.0

Effect of loops



- Miss rate better predictor or performance than number of mem. accesses!
- For large N, kij and ikj performance almost constant.
  Due to **hardware prefetching**, able to recognize stride-1 patterns.

Book

P & H, Chapter 4

Exercise
From H & P Sixth Edition Appendix B

Correction on AMAT Calculation

- Do not calculate the normalized AMAT in case of separate instruction and data caches unless explicitly specified. Not everybody does this normalisation

Exercise 1

Assume a fully associative write-back cache with many cache entries that starts empty. Following is a sequence of five memory operations (the address is in square brackets):

```
Write Mem[100];
Write Mem[100];
Read  Mem[200];
Write Mem[200];
Write Mem[100].
```

What are the number of hits and misses when using no-write allocate versus write allocate?

Exercise 1

For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no-write allocate is four misses and one hit.

For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits because 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

Exercise 2

Which has the lower miss rate: a 16 KiB instruction cache with a 16 KiB data cache or a 32 KiB unified cache? Use the miss rates in Figure B.6 to help calculate the correct answer, assuming 36% of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 200 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. Using the pipelining terminology the unified cache leads to a structural hazard. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

Size (KiB)	Instruction cache	Data cache	Unified cache
8	8.16	44.0	63.0
16	3.82	40.9	51.0
32	1.36	38.4	43.3
64	0.61	36.9	39.4
128	0.30	35.3	36.2
256	0.02	32.6	32.9

Figure B.6 Miss per 1000 instructions for instruction, data, and unified caches of different sizes. The percentage of instruction references is about 74%. The data are for two-way associative caches with 64-byte blocks for the same computer

Exercise 2

First let's convert misses per 1000 instructions into miss rates. Solving the preceding general formula, the miss rate is

$$\text{Miss rate} = \frac{\dfrac{\text{Misses}}{\dfrac{1000\ \text{Instructions}}{}}/1000}{\dfrac{\text{Memory accesses}}{\text{Instruction}}}$$

Because every instruction access has exactly one memory access to fetch the instruction, the instruction miss rate is

$$\text{Miss rate}_{16\,\text{KB instruction}} = \frac{3.82/1000}{1.00} = 0.004$$

Because 36% of the instructions are data transfers, the data miss rate is

$$\text{Miss rate}_{16\,\text{KB data}} = \frac{40.9/1000}{0.36} = 0.114$$

The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss rate}_{32\,\text{KB unified}} = \frac{43.3/1000}{1.00 + 0.36} = 0.0318$$

Exercise 2

As stated herein, about 74% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(74\% \times 0.004) + (26\% \times 0.114) = 0.0326$$

This is Normalised

Thus, a 32 KiB unified cache has a slightly lower effective miss rate than two 16 KiB caches.

The average memory access time formula can be divided into instruction and data accesses:

$$\begin{aligned}&\text{Average memory access time}\\&= \%\,\text{instructions} \times (\text{Hit time} + \text{Instruction miss rate} \times \text{Miss penalty})\\&\quad + \%\,\text{data} \times (\text{Hit time} + \text{Data miss rate} \times \text{Miss penalty})\end{aligned}$$

Therefore, the time for each organization is

This is Normalised AMAT

$$\begin{aligned}&\text{Average memory access time}_{\text{split}}\\&= 74\% \times (1 + 0.004 \times 200) + 26\% \times (1 + 0.114 \times 200)\\&= (74\% \times 1.80) + (26\% \times 23.80) = 1.332 + 6.188 = 7.52\\&\text{Average memory access time}_{\text{unified}}\\&= 74\% \times (1 + 0.0318 \times 200) + 26\% \times (1 + 1 + 0.0318 \times 200)\\&= (74\% \times 7.36) + (26\% \times 8.36) = 5.446 + 2.174 = 7.62\end{aligned}$$

Hence, the split caches in this example—which offer two memory ports per clock cycle, thereby avoiding the structural hazard—have a better average memory access time than the single-ported unified cache despite having a worse effective miss rate.

Exercise 3

Let's use an in-order execution computer . Assume that the cache miss penalty is 200 clock cycles, and all instructions usually take 1.0 clock cycles (ignoring memory stalls). Assume that the average miss rate is 2%, there is an average of 1.5 memory references per instruction, and the average number of cache misses per 1000 instructions is 30. What is the impact on performance when behavior of the cache is included? Calculate the impact using both misses per instruction and miss rate.

Exercise 3

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

The performance, including cache misses, is

$$\begin{aligned} \text{CPU time}_{\text{with cache}} &= \text{IC} \times [1.0 + (30/1000 \times 200)] \times \text{Clock cycle time} \\ &= \text{IC} \times 7.00 \times \text{Clock cycle time} \end{aligned}$$

Now calculating performance using miss rate:

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\begin{aligned} \text{CPU time}_{\text{with cache}} &= \text{IC} \times [1.0 + (1.5 \times 2\% \times 200)] \times \text{Clock cycle time} \\ &= \text{IC} \times 7.00 \times \text{Clock cycle time} \end{aligned}$$

The clock cycle time and instruction count are the same, with or without a cache. Thus, CPU time increases sevenfold, with CPI from 1.00 for a "perfect cache" to 7.00 with a cache that can miss. Without any memory hierarchy at all the CPI would increase again to $1.0 + 200 \times 1.5$ or 301—a factor of more than 40 times longer than a system with a cache!

Exercise 4

What is the impact of 1-way vs 2-way cache organizations on the performance of a processor? Assume that the CPI with a perfect cache is 1.0, the clock cycle time is 0.35 ns, there are 1.4 memory references per instruction, the size of both caches is 128 KiB, and both have a block size of 64 bytes. One cache is direct mapped and the other is two-way set associative. for set associative caches we must add a multiplexor to select between the blocks in the set depending on the tag match. Because the speed of the processor can be tied directly to the speed of a cache hit, assume the processor clock cycle time must be stretched 1.35 times to accommodate the selection multiplexor of the set associative cache. the cache miss penalty is 65 ns for either cache organization. (In practice, it is normally rounded up or down to an integer number of clock cycles.) First, calculate the average memory access time and then processor performance. Assume the hit time is 1 clock cycle, the miss rate of a direct-mapped 128 KiB cache is 2.1%, and the miss rate for a two-way set associative cache of the same size is 1.9%.

Exercise 4

Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Thus, the time for each organization is

$$\text{Average memory access time}_{\text{1-way}} = 0.35 + (.021 \times 65) = 1.72\,\text{ns}$$

$$\text{Average memory access time}_{\text{2-way}} = 0.35 \times 1.35 + (.019 \times 65) = 1.71\,\text{ns}$$

The average memory access time is better for the two-way set-associative cache.

The processor performance is

$$\begin{aligned}\text{CPU time} &= \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time} \\ &= \text{IC} \times \left[(\text{CPI}_{\text{execution}} \times \text{Clock cycle time}) \right. \\ &\quad \left. + \left(\text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \times \text{Clock cycle time}\right)\right]\end{aligned}$$

Exercise 4

Substituting 65 ns for (Miss penalty × Clock cycle time), the performance of each cache organization is

$$\text{CPU time}_{\text{1-way}} = \text{IC} \times [1.0 \times 0.35 + (0.021 \times 1.4 \times 65)] = 2.26 \times \text{IC}$$

$$\text{CPU time}_{\text{2-way}} = \text{IC} \times [1.0 \times 0.35 \times 1.35 + (0.019 \times 1.4 \times 65)] = 2.20 \times \text{IC}$$

and relative performance is

$$\frac{\text{CPU time}_{\text{2-way}}}{\text{CPU time}_{\text{1-way}}} = \frac{2.26 \times \text{Instruction count}}{2.20 \times \text{Instruction count}} = 1.03$$

In contrast to the results of average memory access time comparison, the direct-mapped cache leads to slightly better average performance because the clock cycle is stretched for *all* instructions for the two-way set associative case, even if there are fewer misses. Because CPU time is our bottom-line evaluation and because direct mapped is simpler to build, the preferred cache is direct mapped in this example.

Exercise 5

Let's redo the preceding example, but this time we assume the processor with the longer clock cycle time supports out-of-order execution yet still has a direct-mapped cache. Assume 30% of the 65 ns miss penalty can be overlapped; that is, the average Miss penalty is now 45.5 ns.

Exercise 5

Average memory access time for the out-of-order (OOO) computer is

$$\text{Average memory access time}_{\text{1-way, OOO}} = 0.35 \times 1.35 + (0.021 \times 45.5) = 1.43\,\text{ns}$$

The performance of the OOO cache is

$$\text{CUP time}_{\text{1-way, OOO}} = \text{IC} \times \left| 1.0 \times 0.35 \times 1.35 + (0.021 \times 1.4 \times 45.5)\right] = 1.81 \times \text{IC}$$

Hence, despite a much slower clock cycle time and the higher miss rate of a direct-mapped cache, the out-of-order computer can be slightly faster if it can hide 30% of the miss penalty.

Exercise 6

Given the following data, what is the impact of second-level cache associativity on its miss penalty?

- Hit time$_{L2}$ for direct mapped = 10 clock cycles.
- Two-way set associativity increases hit time by 0.1 clock cycle to 10.1 clock cycles.
- Local miss rate$_{L2}$ for direct mapped = 25%.
- Local miss rate$_{L2}$ for two-way set associative = 20%.
- Miss penalty$_{L2}$ = 200 clock cycles.

Exercise 6

For a direct-mapped second-level cache, the first-level cache miss penalty is

$$\text{Miss penalty}_{\text{1-way L2}} = 10 + 25\% \times 200 = 60.0 \text{ clock cycles}$$

Adding the cost of associativity increases the hit cost only 0.1 clock cycle, making the new first-level cache miss penalty:

$$\text{Miss penalty}_{\text{2-way L2}} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and processor. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we shave the second-level hit time to 10 cycles; if not, we round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{\text{2-way L2}} = 10 + 20\% \times 200 = 50.0 \text{ clock cycles}$$
$$\text{Miss penalty}_{\text{2-way L2}} = 11 + 20\% \times 200 = 51.0 \text{ clock cycles}$$