

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 ($\text{Performance} = 1/\text{latency}$)
- Throughput (bandwidth): number of tasks/unit time. It is not additive

Performance — In our words

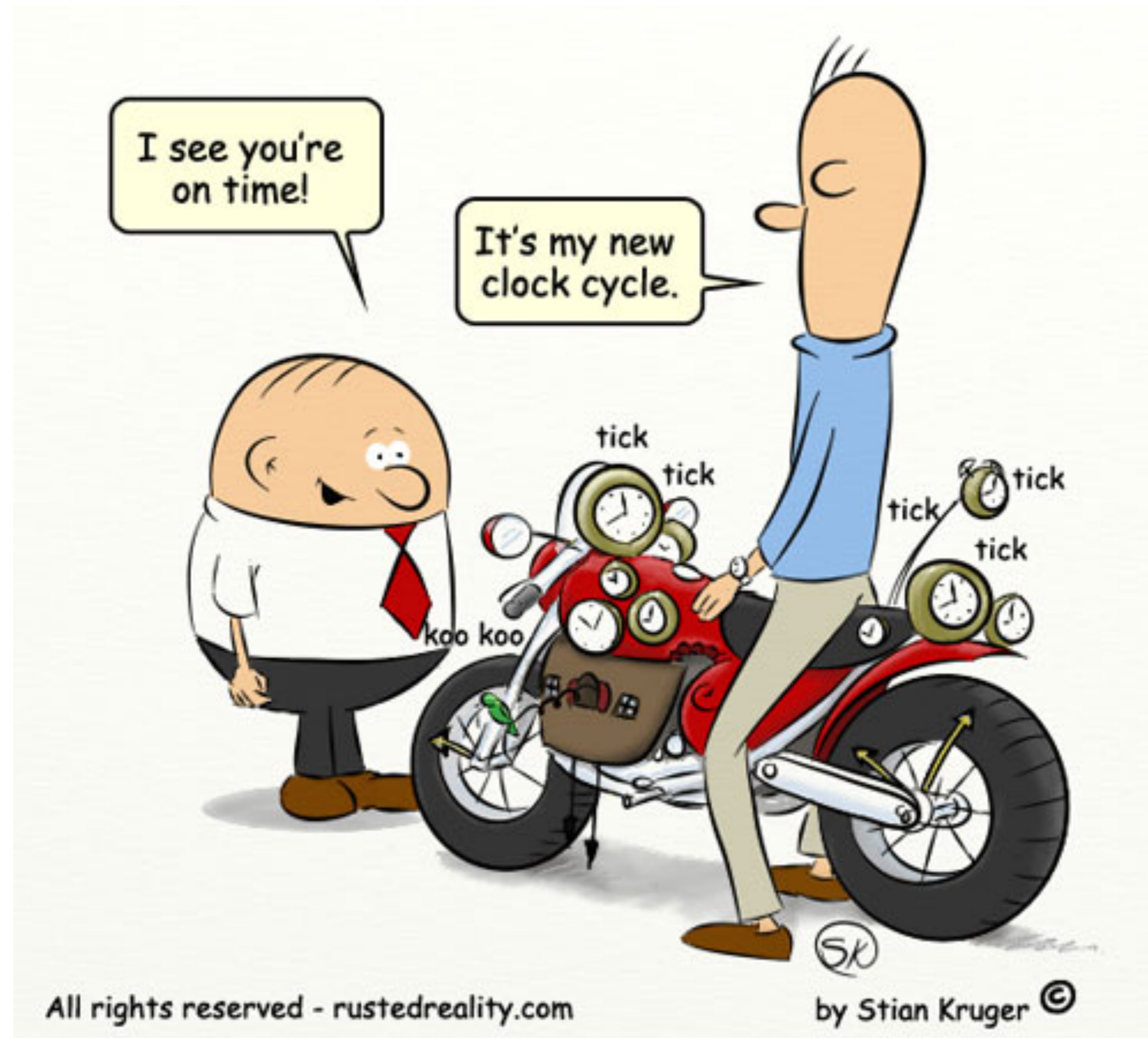
Which computer is faster?

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 $\rightarrow$ SPEC CPU 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

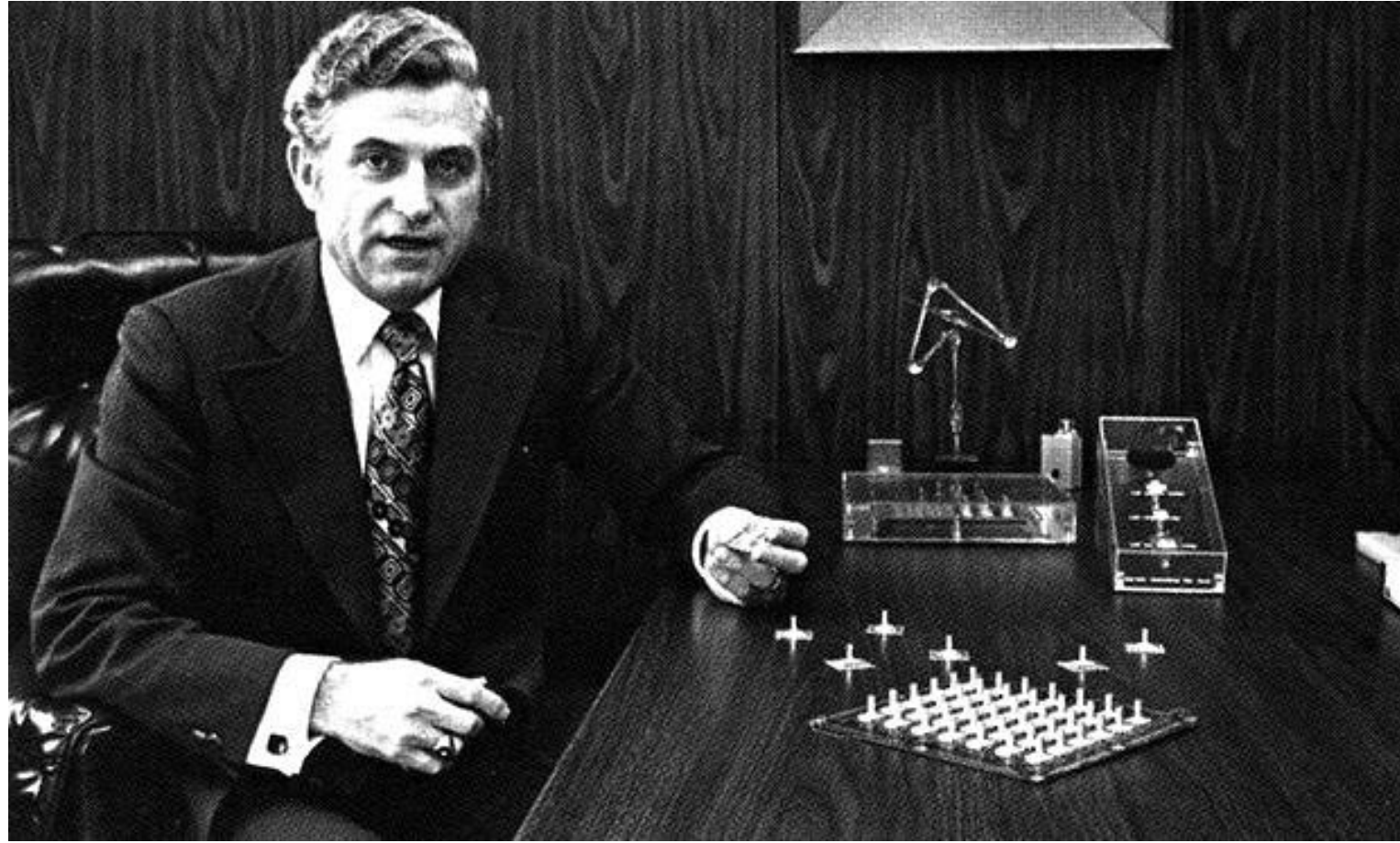


AM vs. GM

- 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)

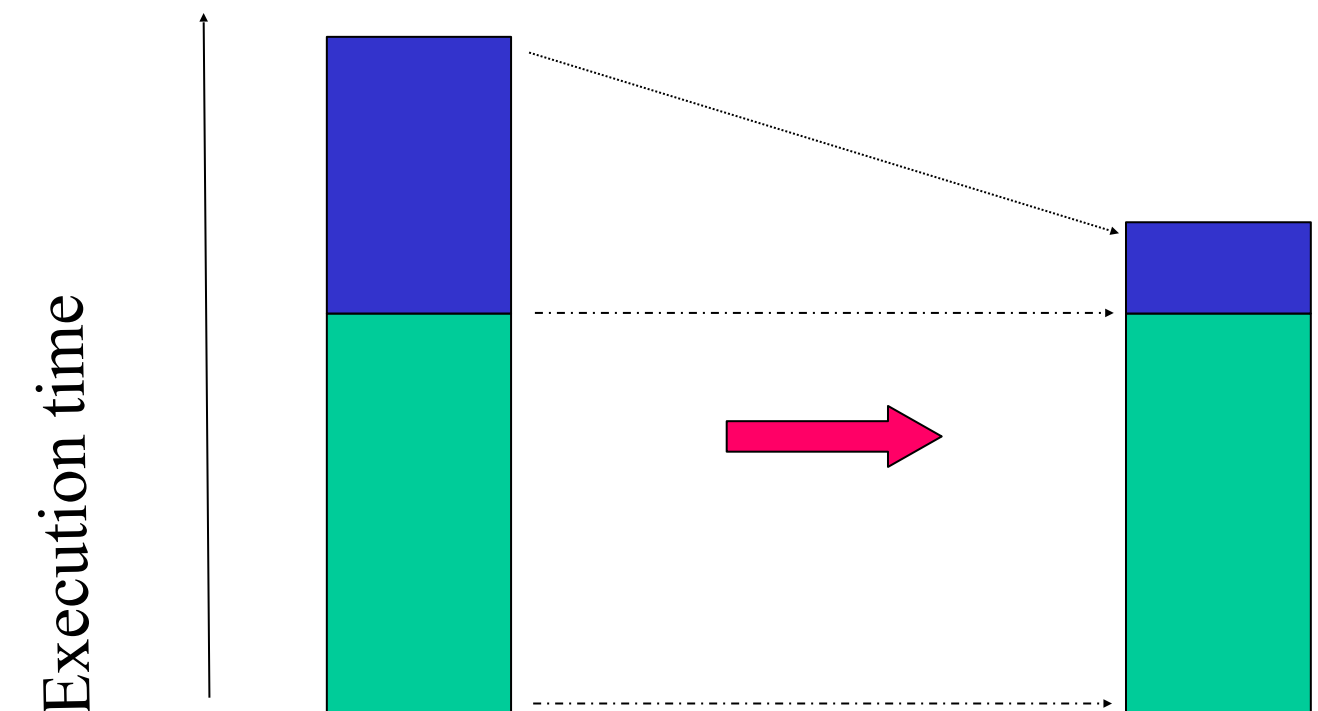


$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}}$$

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

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 = $\frac{1}{2}$
 - 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.

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

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

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?

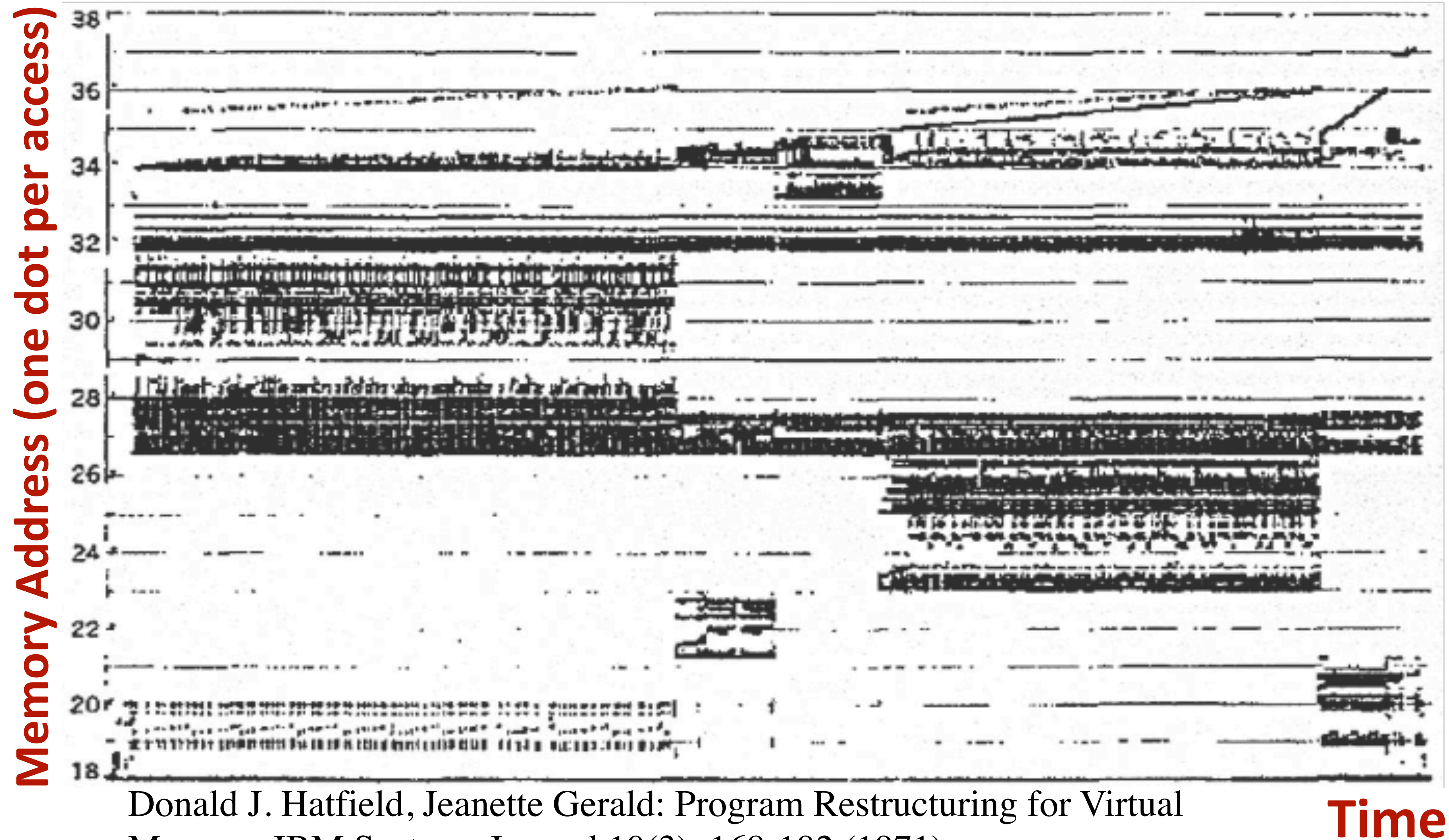
- A. Small speedup on the large fraction of execution time.
- B. Large speedup on the small fraction of execution time.
- C. Does not matter.

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

Principle of Locality

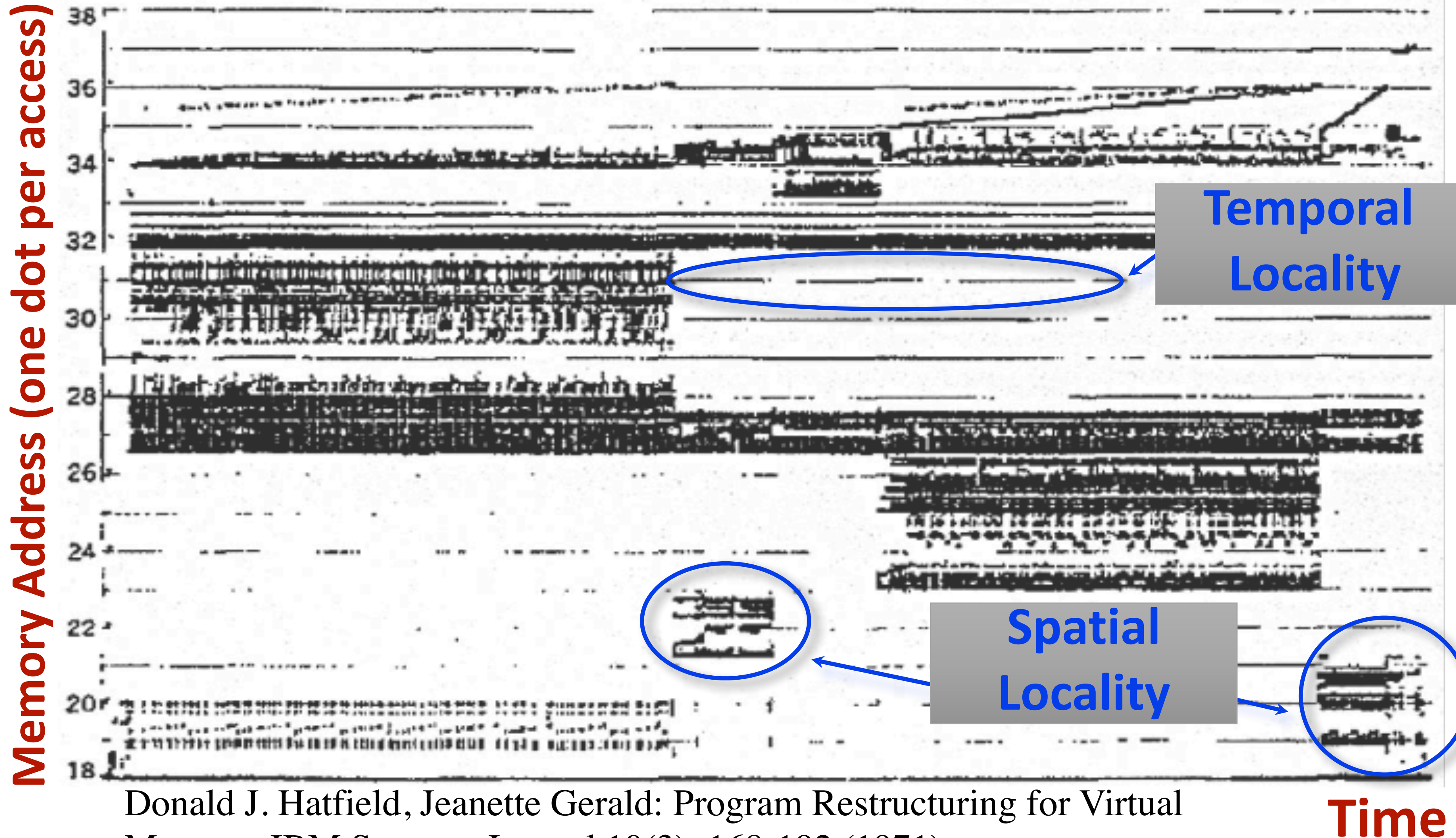
- Programs tend to use the data and instructions they have used recently.
- So from the recent past, we can have a good idea of future
- **Temporal Locality**: Recently accessed items are to be used in near future.
- **Spatial Locality**: Items whose addresses are near to each other tend to be referenced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

Let's look at the Applications (benchmarks)



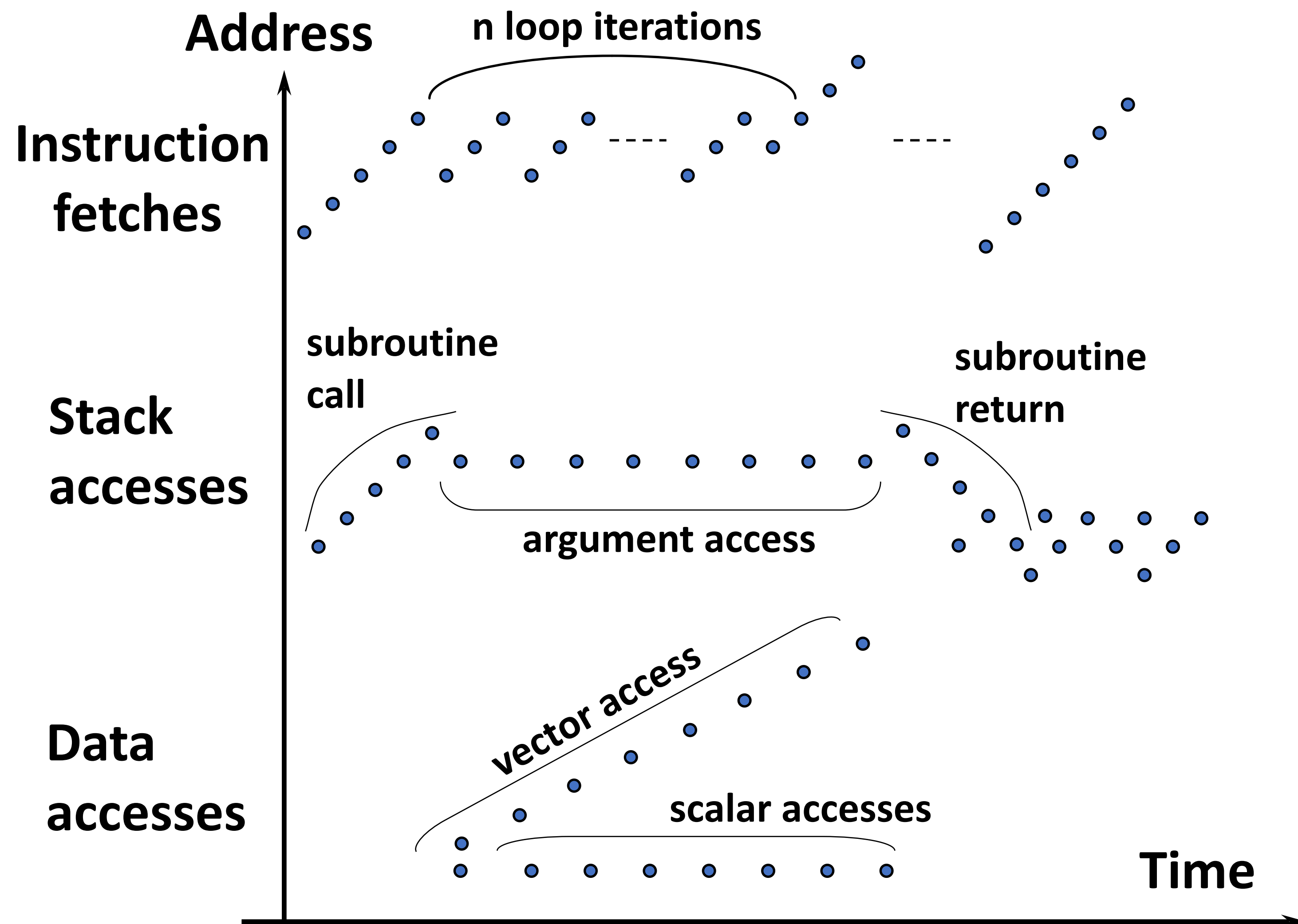
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

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

Minimizing costly DRAM accesses
is critical for performance

Costly DRAM
accesses ☹️

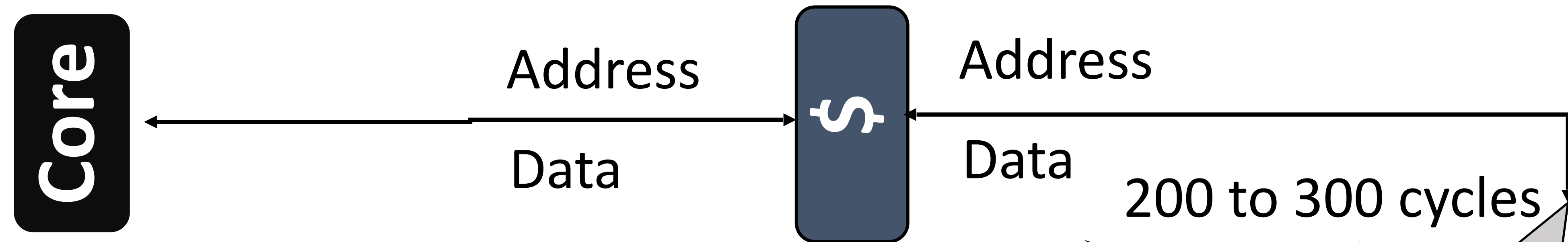


4 GB DRAM

South pole ☹️

Caching: Why does it work...?

North pole 😊



Caching is a **speculation** technique 😊
Works – if locality

**Costly DRAM
accesses** 😞



Do not ignore the common case

Reduction in DRAM accesses ~ Improvement in execution time



WRONG!

What if your program is not memory intensive

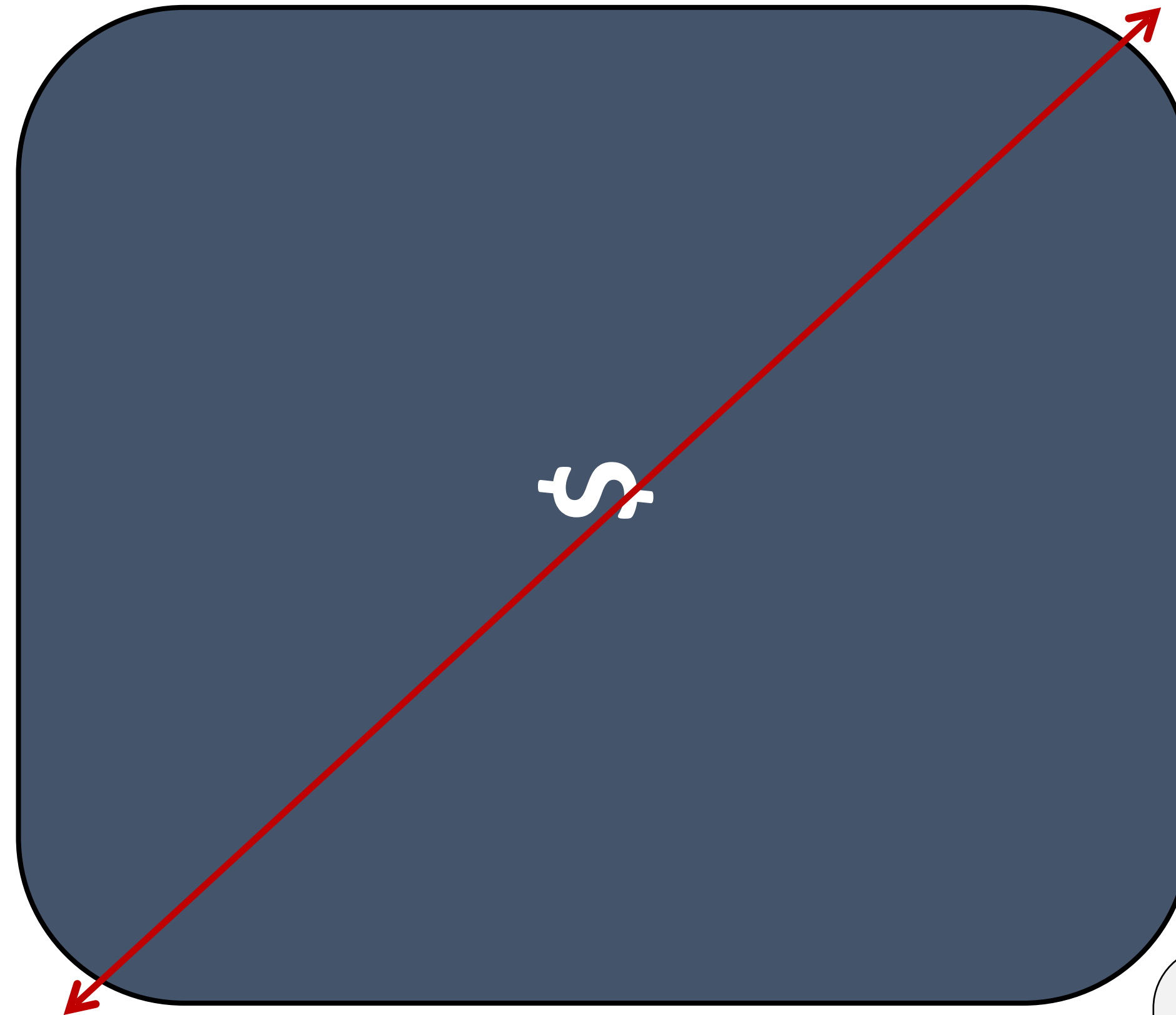


How big/small?

Core



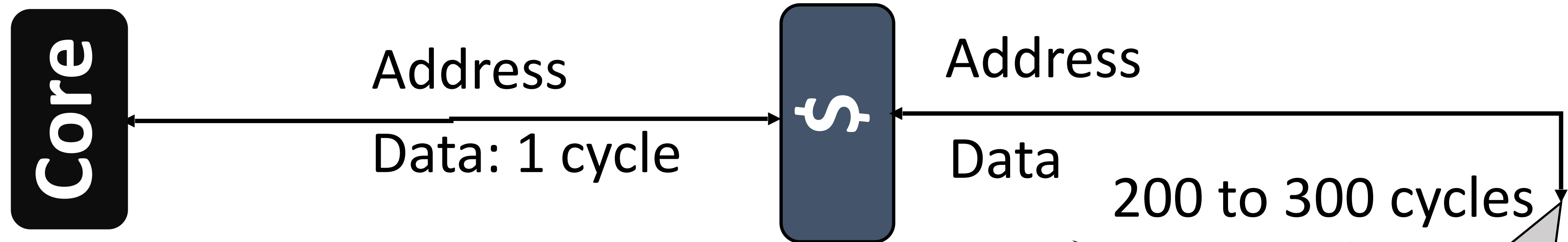
Latency: low
Area: low
Capacity: low



Latency: high
Area: high
Capacity: high

Cache with latency

North pole 😊



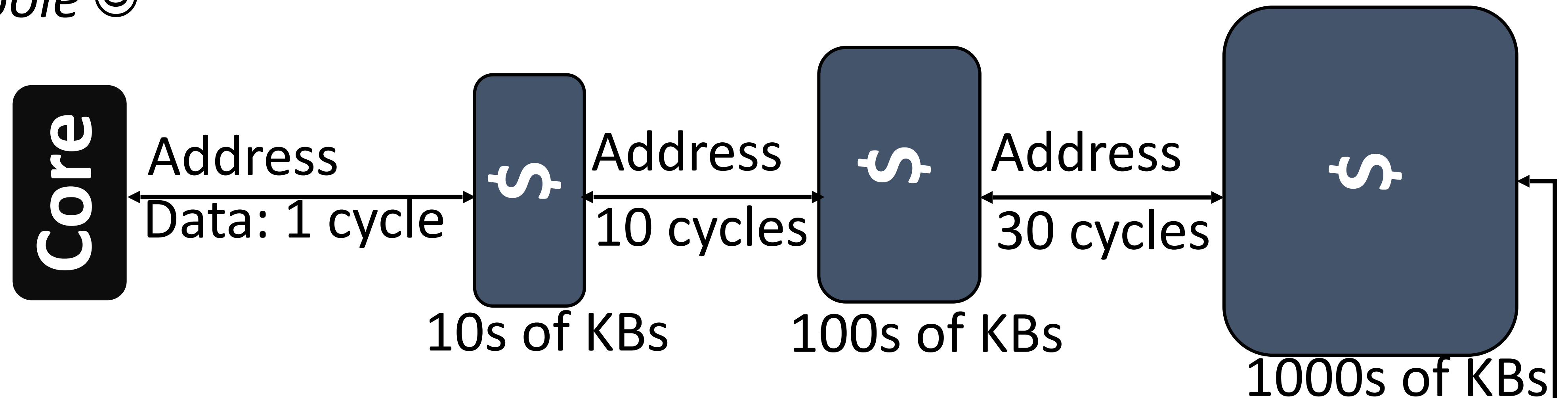
32 to 64KB \$ will be available in one to four cycles ☹️



South pole 😊

Cache hierarchy with latency

North pole 😊



Multi-level cache hierarchy

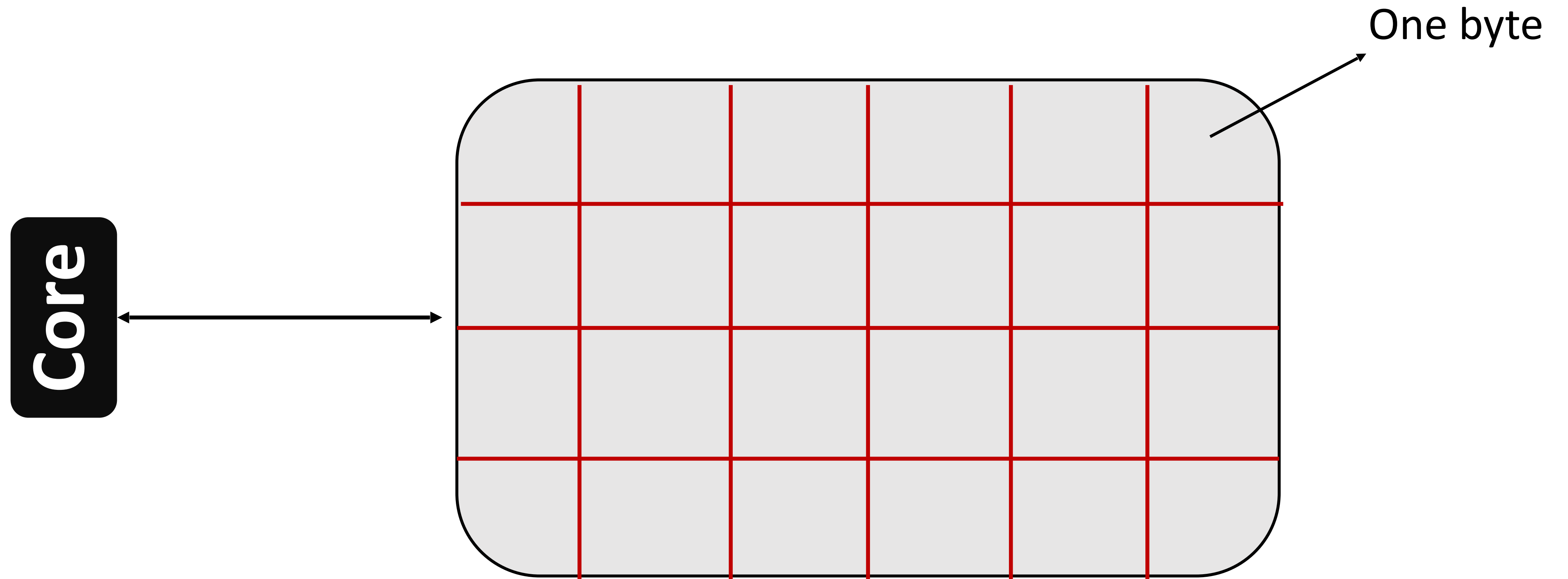
How many levels ?

Total latency < DRAM latency

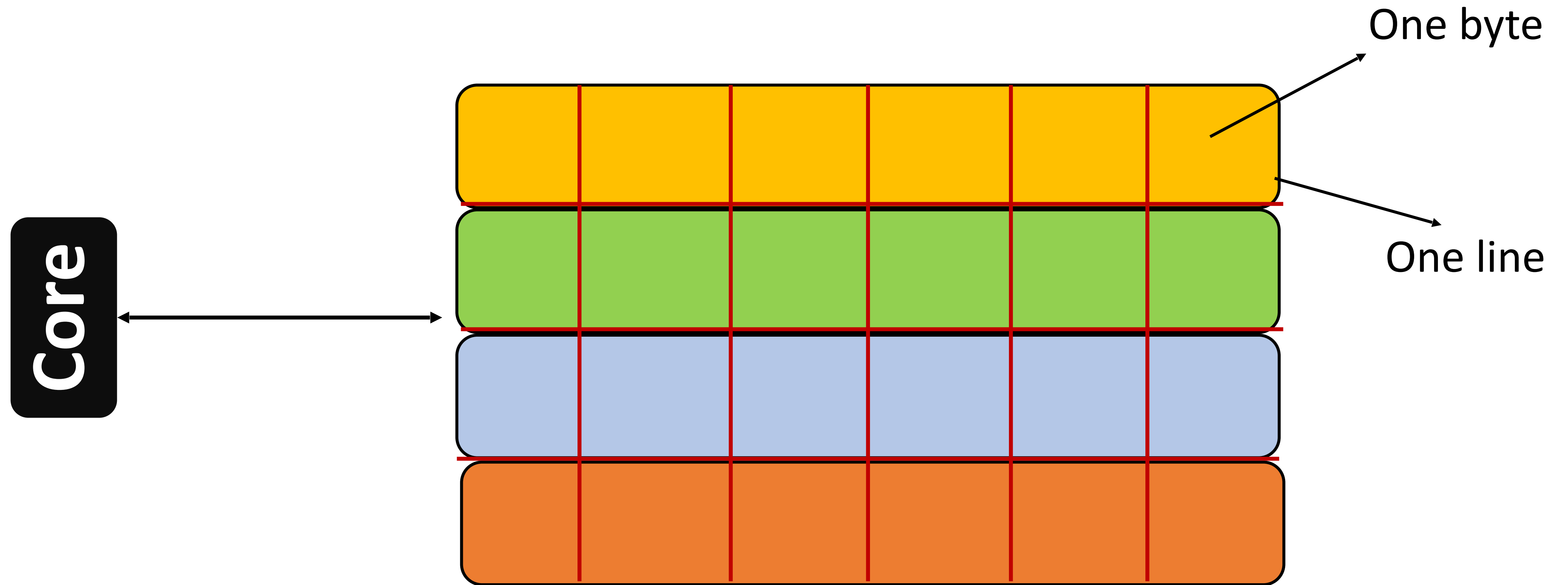


South pole 😊

Accessing a cache



Bytes to blocks (lines)



Typical line size: 64 to 128 Bytes

Before and After You Access...

X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_3

a. Before the reference to X_n

X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_n
X_3

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

4 GB DRAM

Address (32-bit)

Core

One byte

One line

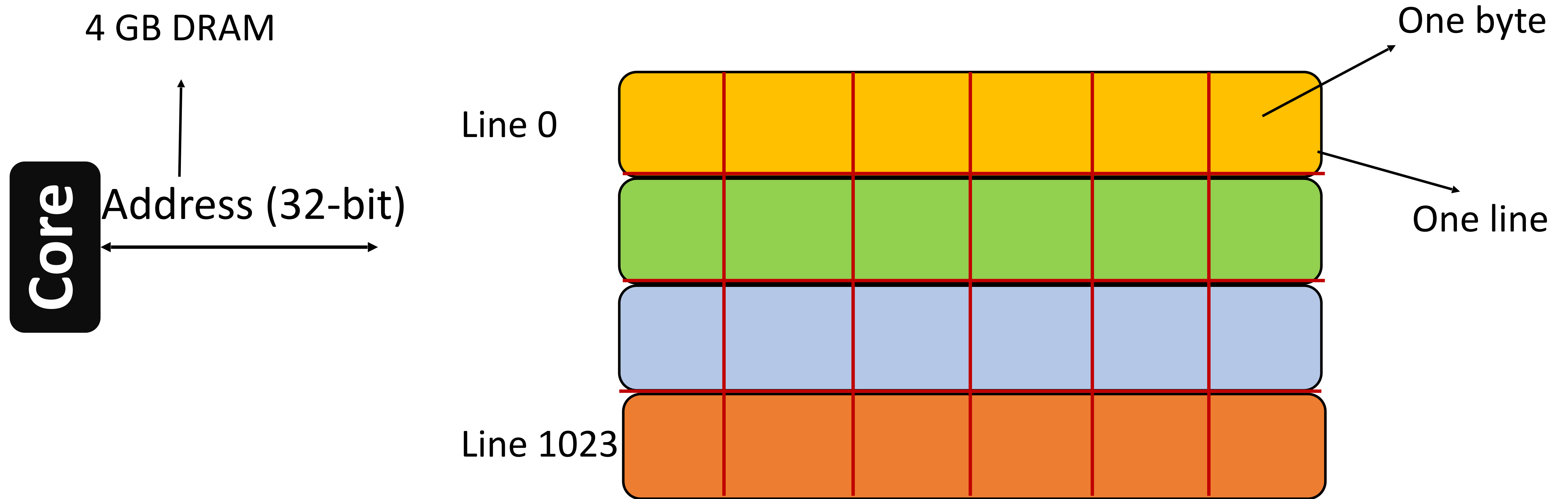
Line 0

Line 1023



A bit deeper: 1024 lines each of 32B

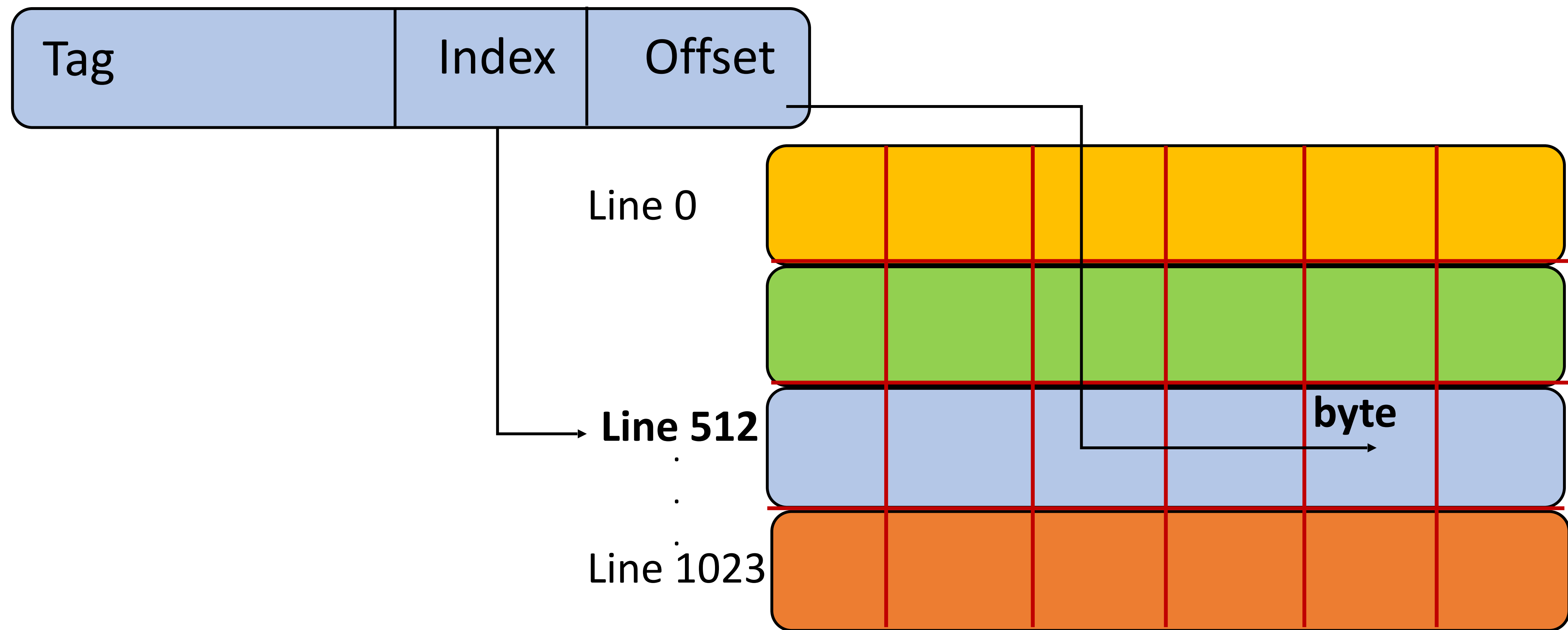
4 GB DRAM



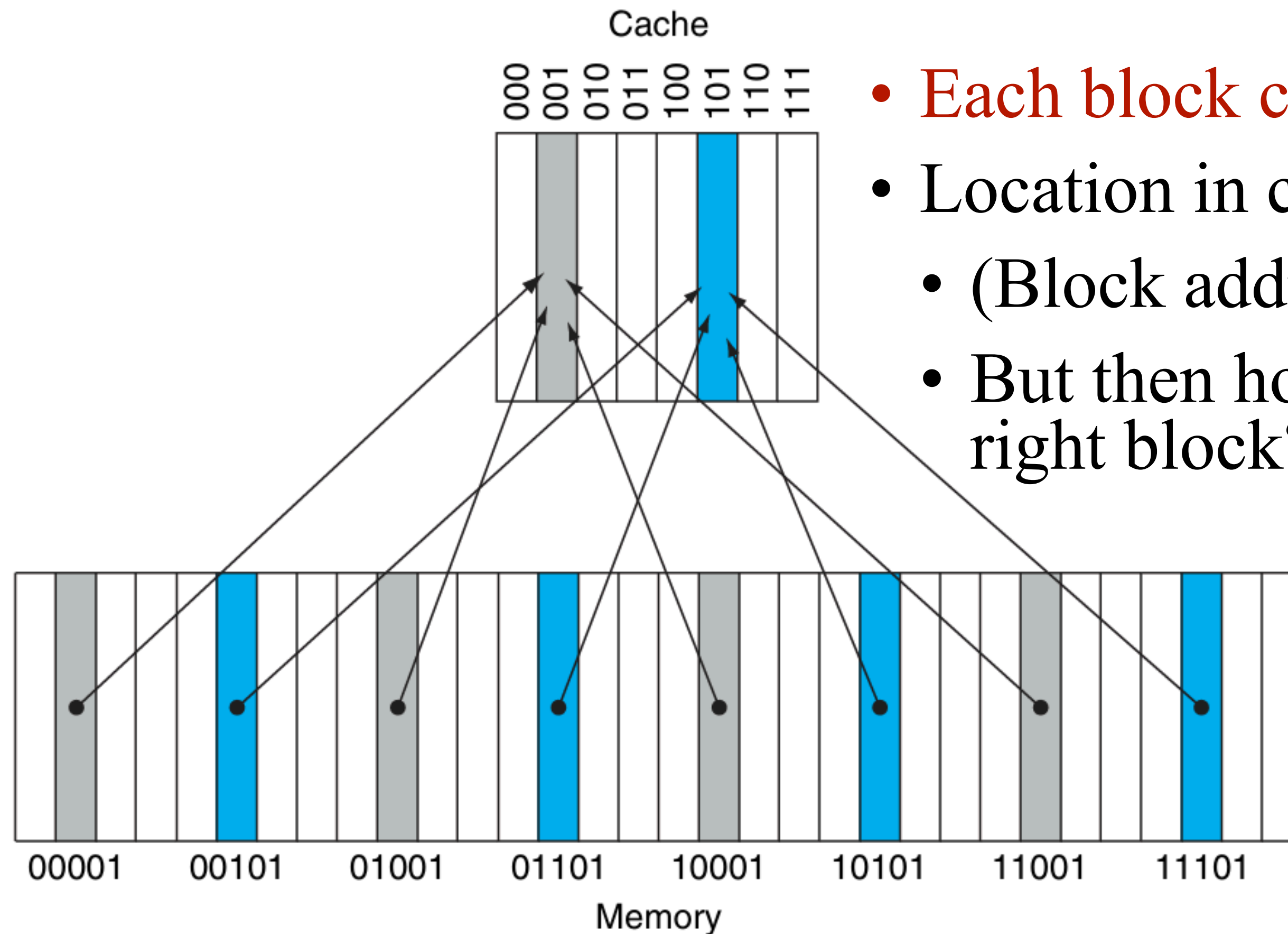
Line number (index): 10 bits

Byte offset (offset): 5 bits

Direct Mapped Cache

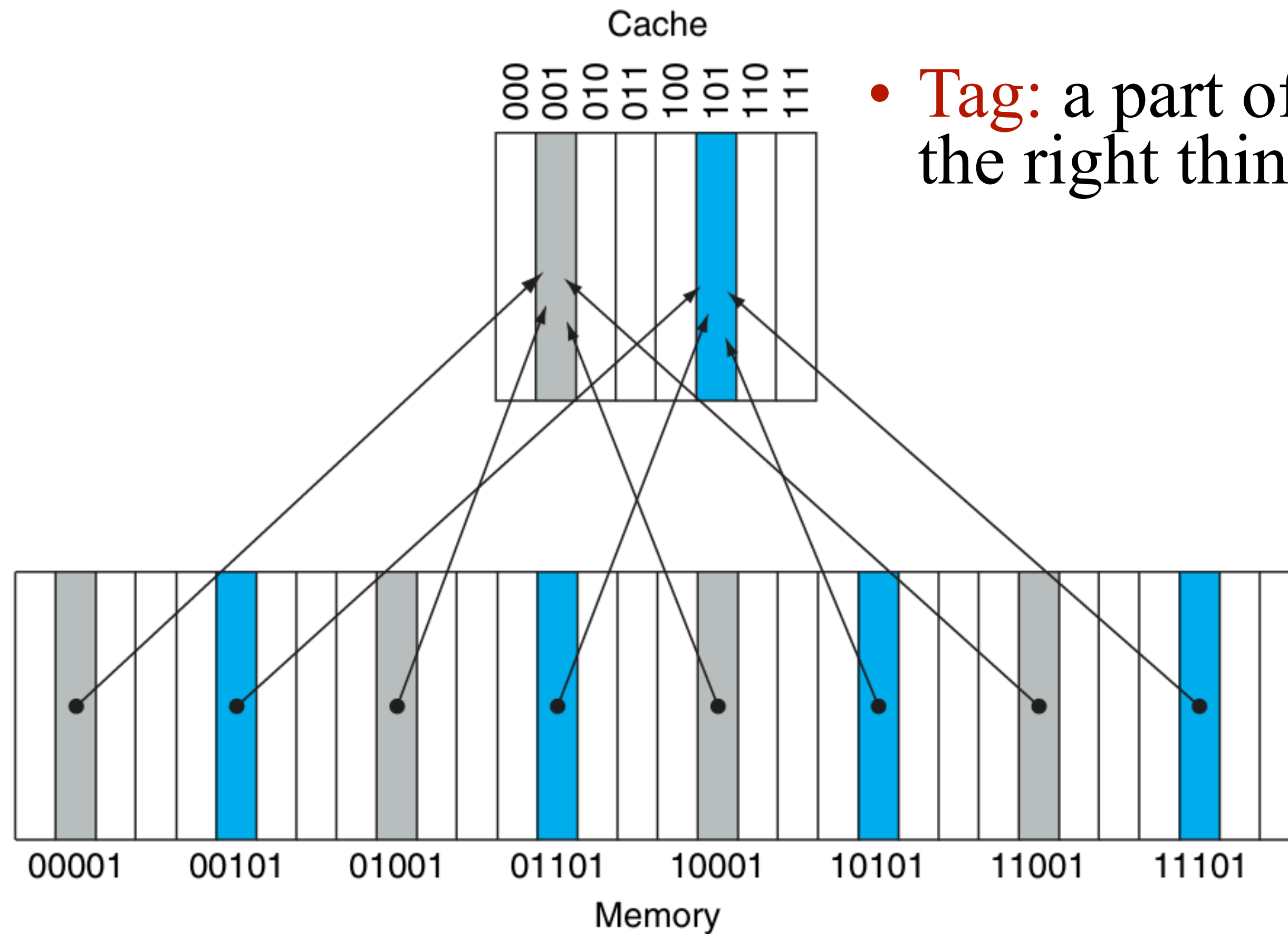


Direct Mapped Cache



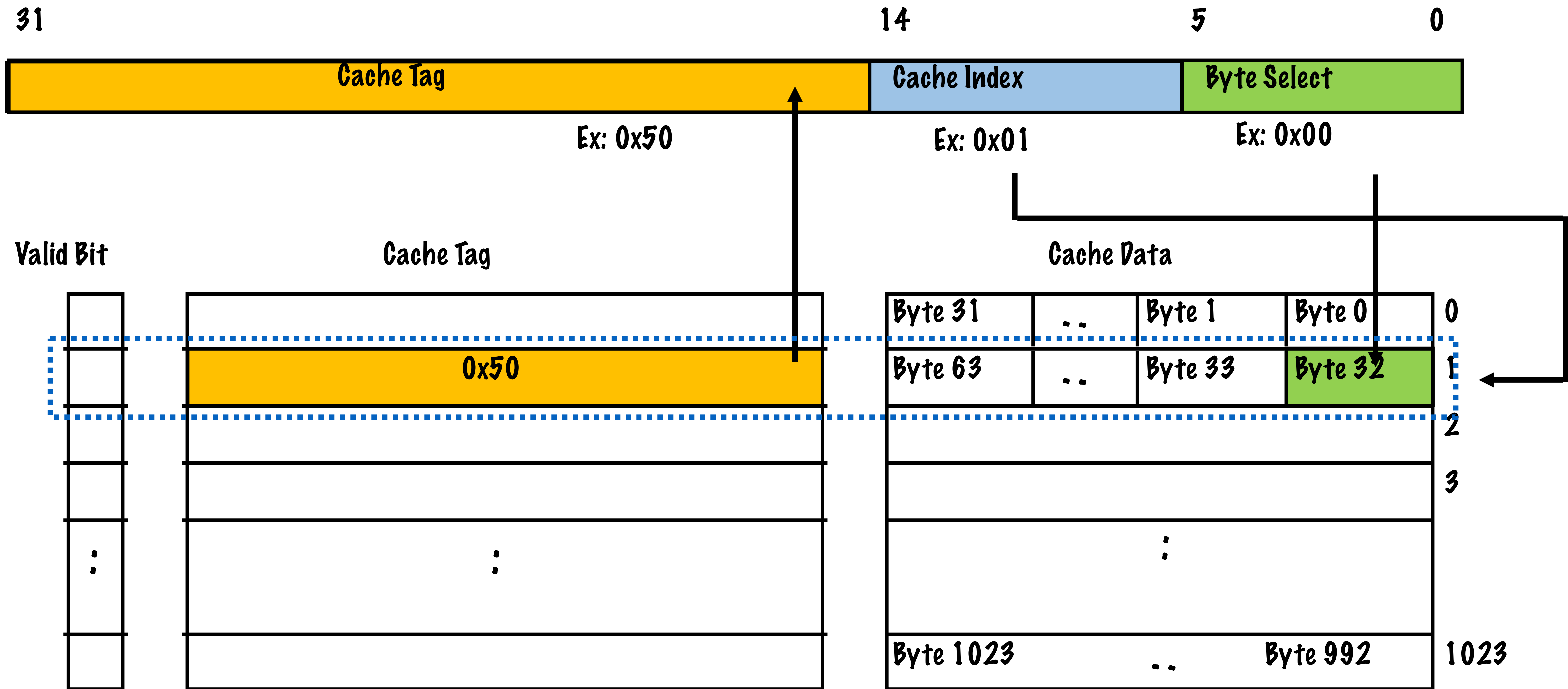
- Each block can go to exactly one place in the cache
- Location in cache:
 - $(\text{Block address}) \bmod (\text{number of blocks in the cache})$
 - But then how do you know you are accessing the right block?

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}	$(10\textcolor{teal}{110}_{\text{two}} \bmod 8) = \textcolor{teal}{110}_{\text{two}}$
11010_{two}	$(11\textcolor{teal}{010}_{\text{two}} \bmod 8) = \textcolor{teal}{010}_{\text{two}}$
10110_{two}	$(10\textcolor{teal}{110}_{\text{two}} \bmod 8) = \textcolor{teal}{110}_{\text{two}}$
11010_{two}	$(11\textcolor{teal}{010}_{\text{two}} \bmod 8) = \textcolor{teal}{010}_{\text{two}}$
10000_{two}	$(10\textcolor{teal}{000}_{\text{two}} \bmod 8) = \textcolor{teal}{000}_{\text{two}}$
00011_{two}	$(00\textcolor{teal}{011}_{\text{two}} \bmod 8) = \textcolor{teal}{011}_{\text{two}}$
10000_{two}	$(10\textcolor{teal}{000}_{\text{two}} \bmod 8) = \textcolor{teal}{000}_{\text{two}}$
10010_{two}	$(10\textcolor{teal}{010}_{\text{two}} \bmod 8) = \textcolor{teal}{010}_{\text{two}}$
10000_{two}	$(10\textcolor{teal}{000}_{\text{two}} \bmod 8) = \textcolor{teal}{000}_{\text{two}}$

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	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	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	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		

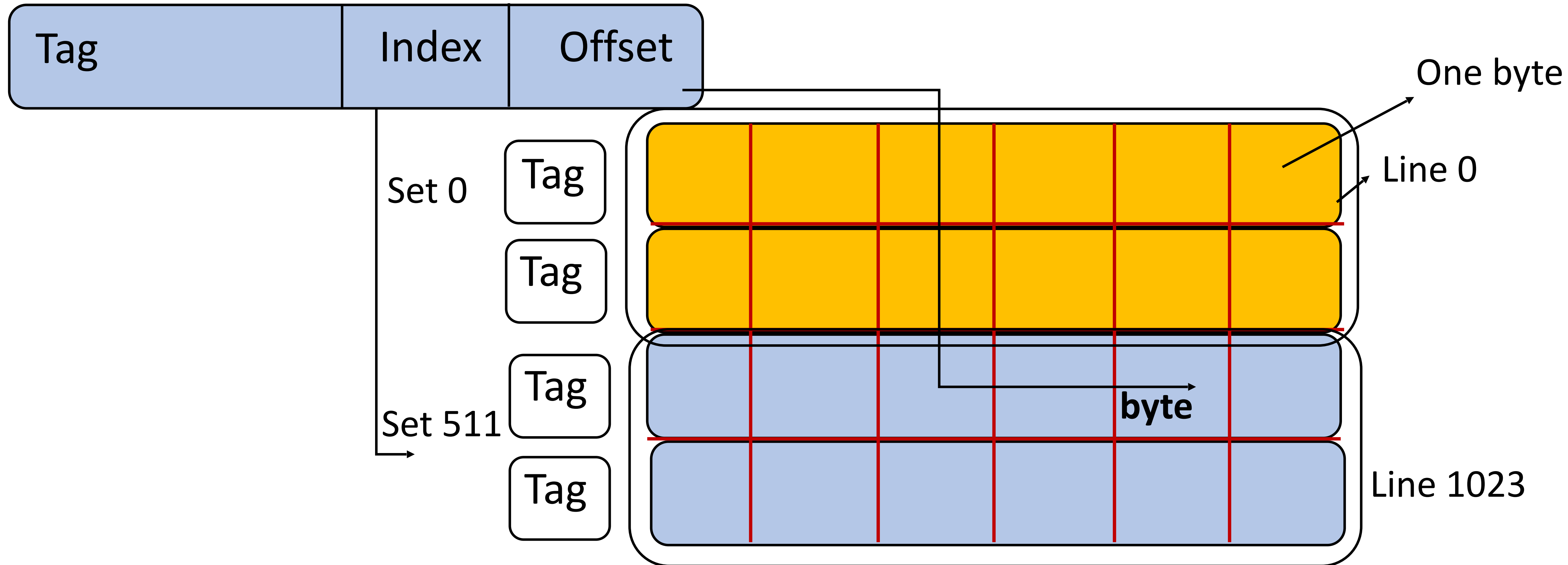
Accessing a cache

- **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**)

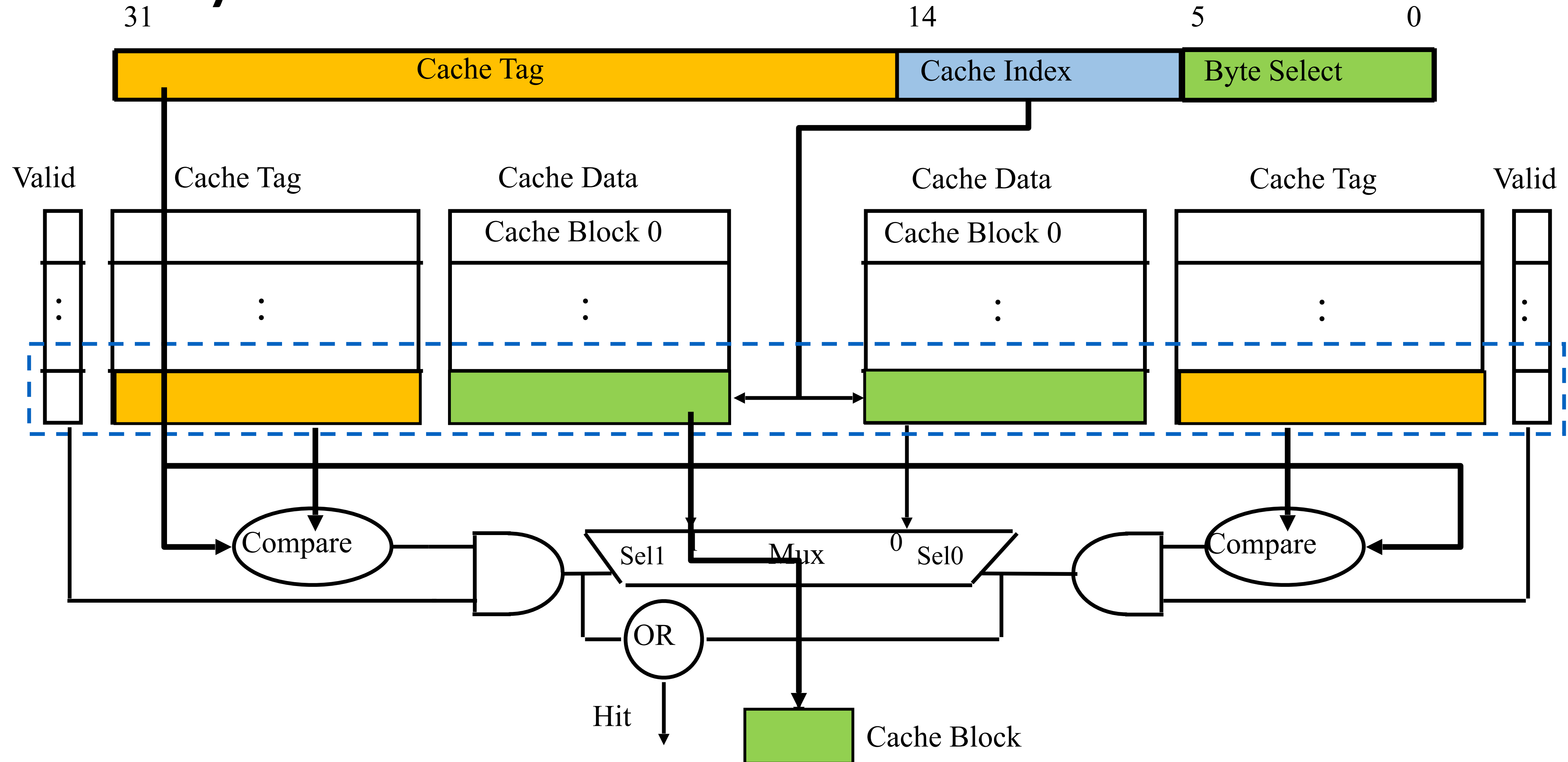
Concept of Valid Bit

- 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

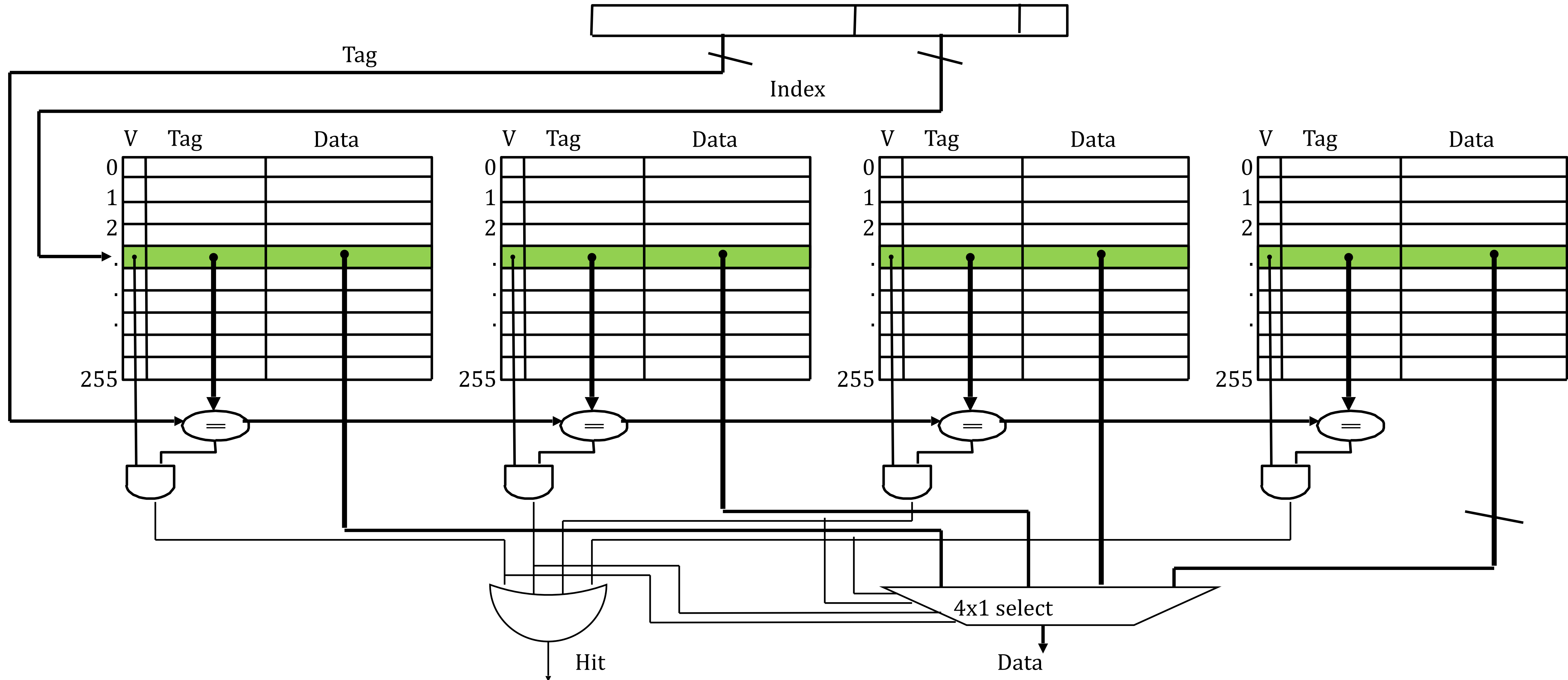
What if we have multiple ways?



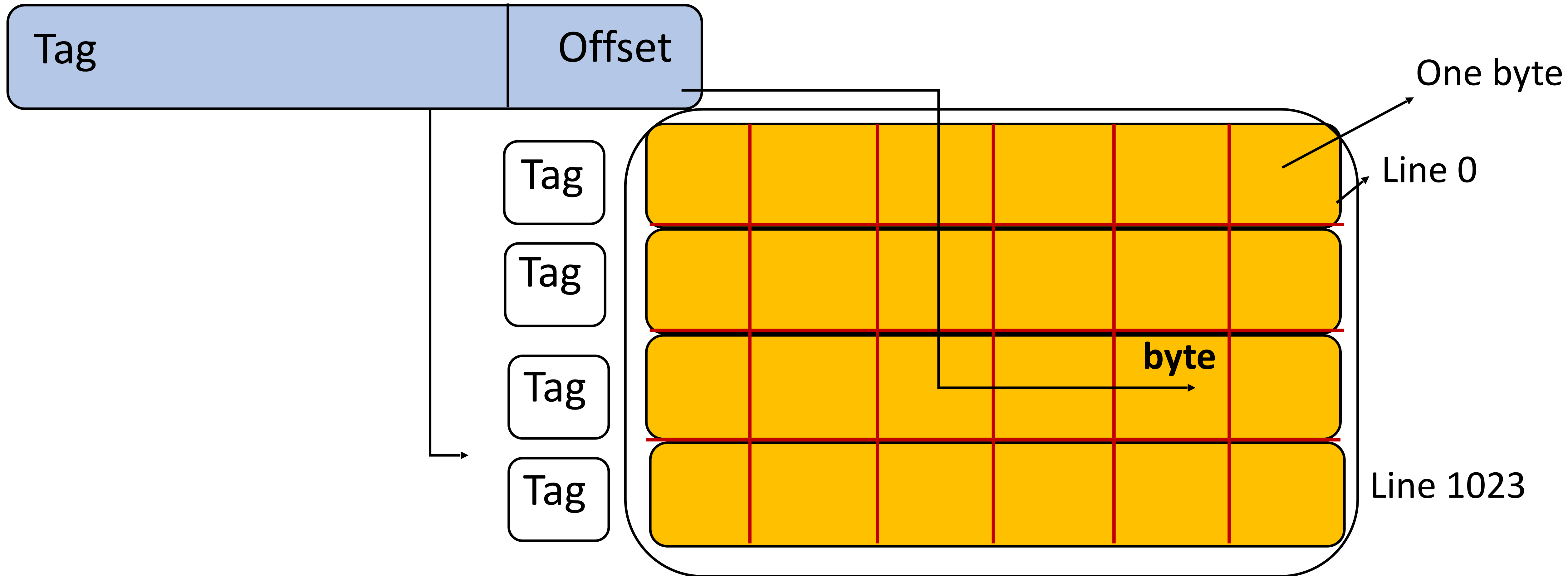
2-way associative in action



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?

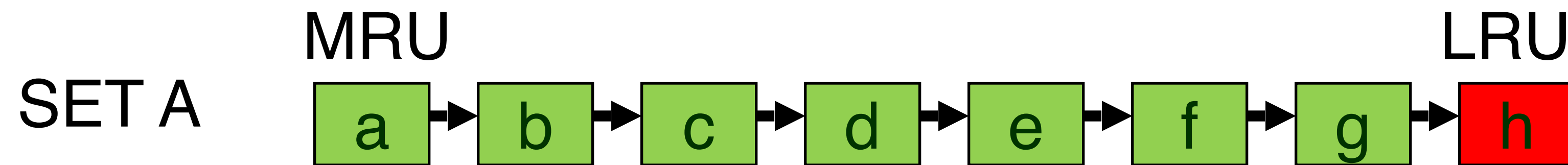
Ideally: Belady’s OPT policy, replace the block that will be used furthest in the future. No one knows the future though 😊

There are three key decisions in a set:

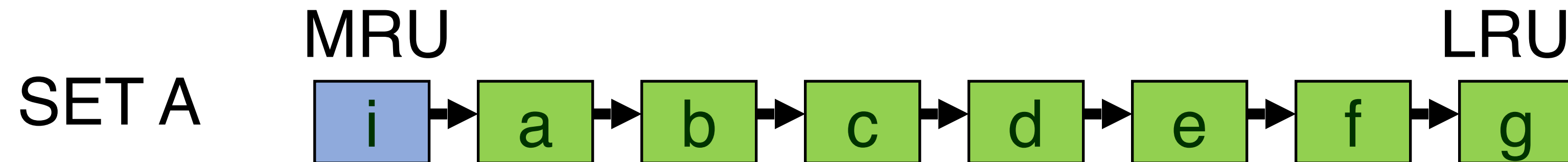
Insertion, promotion, eviction (replacement)

A simple LRU (Least-Recently-Used) Policy

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



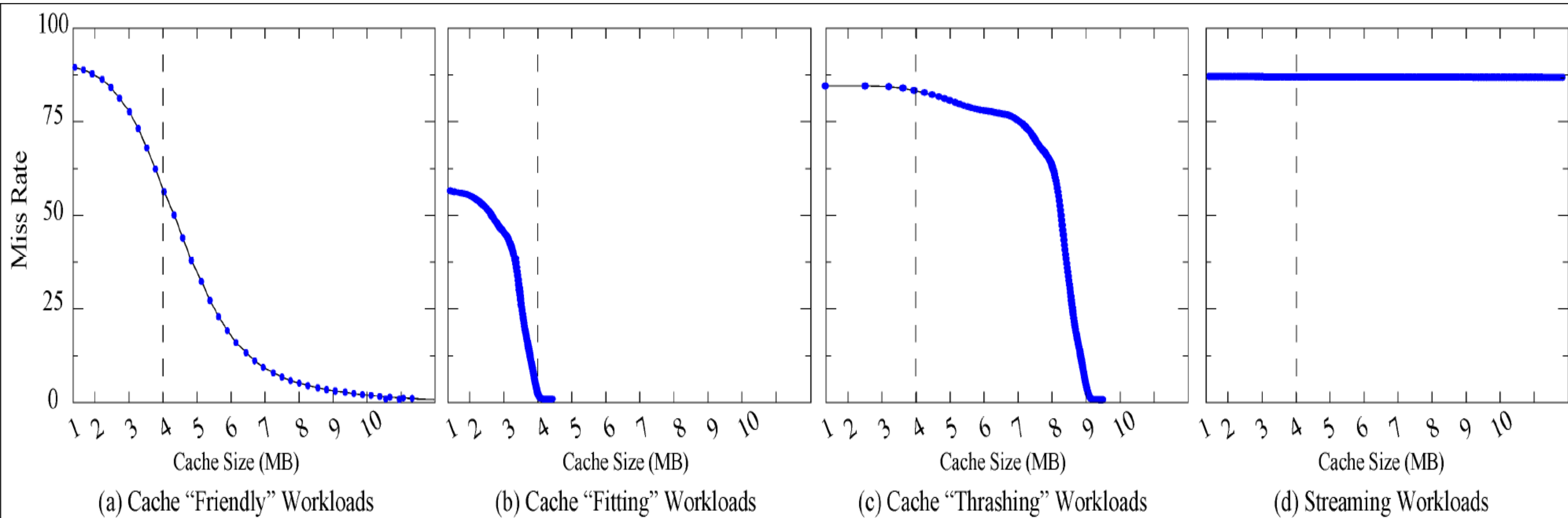
Cache Insertion Policy: New block i inserted into MRU.



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

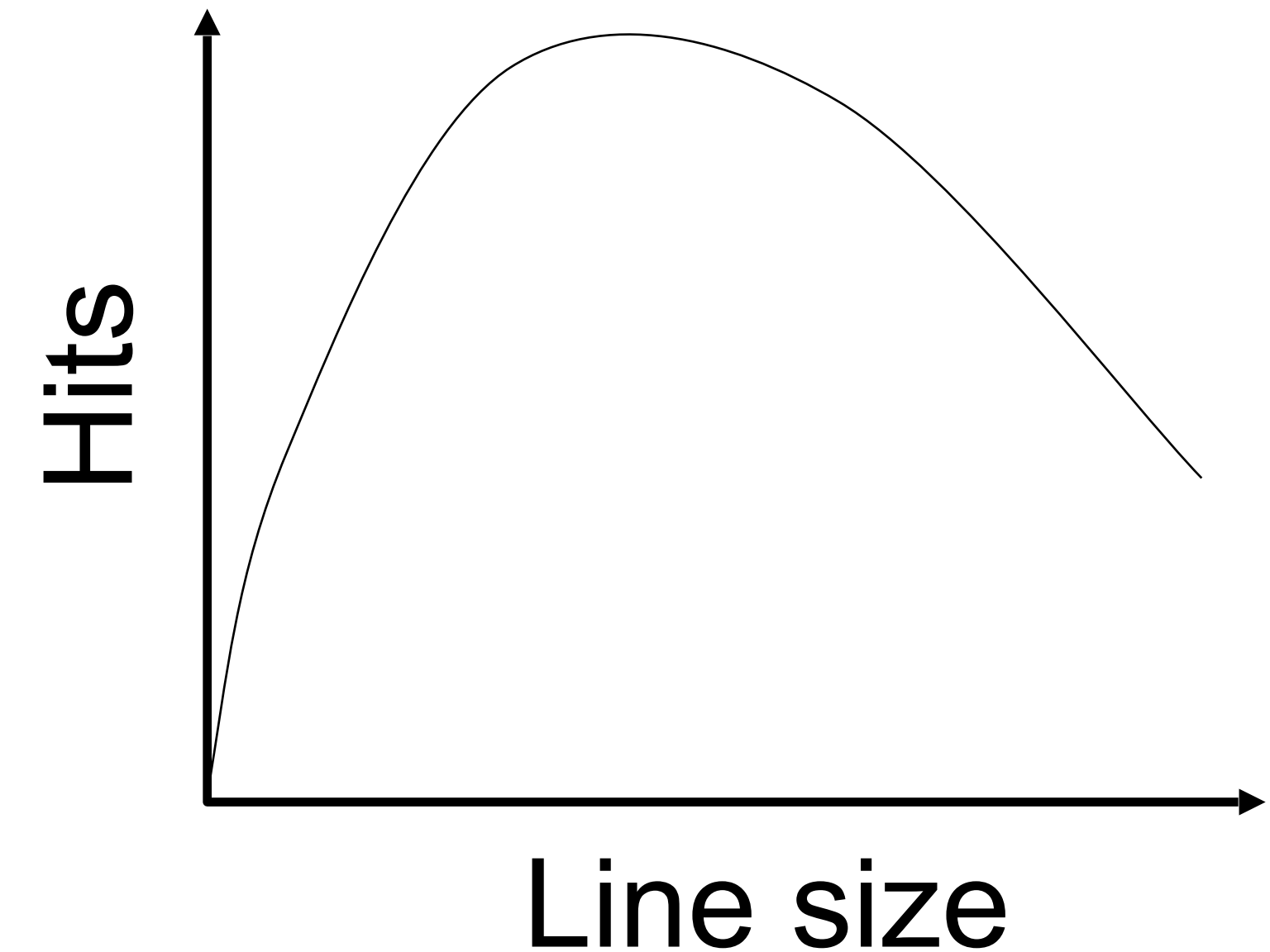
Line size

Too small blocks:

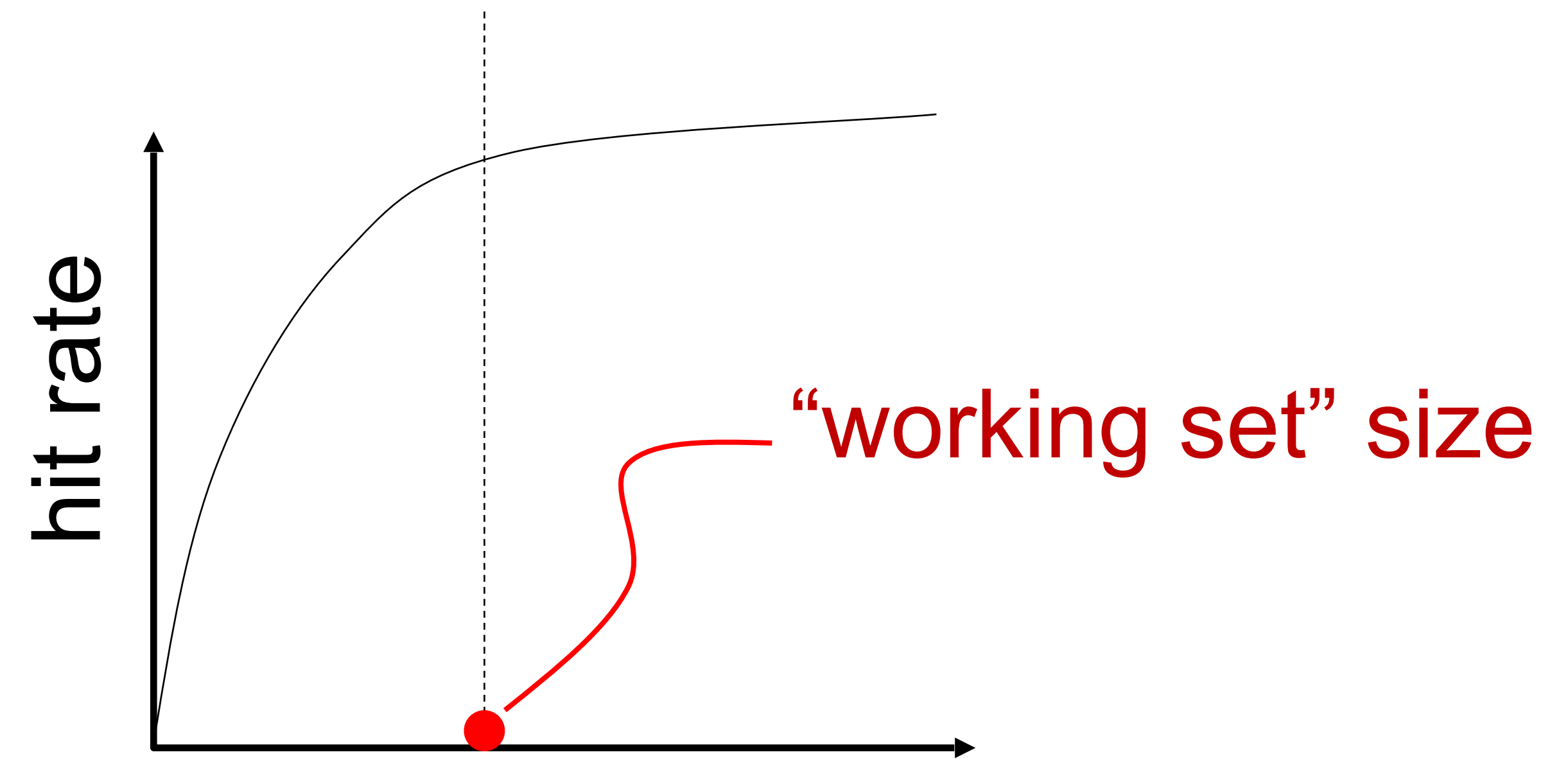
- don't exploit spatial locality well
- have larger tag overhead

Too large blocks:

- too few total # of blocks
- likely-useless data transferred
- Extra bandwidth/energy consumed

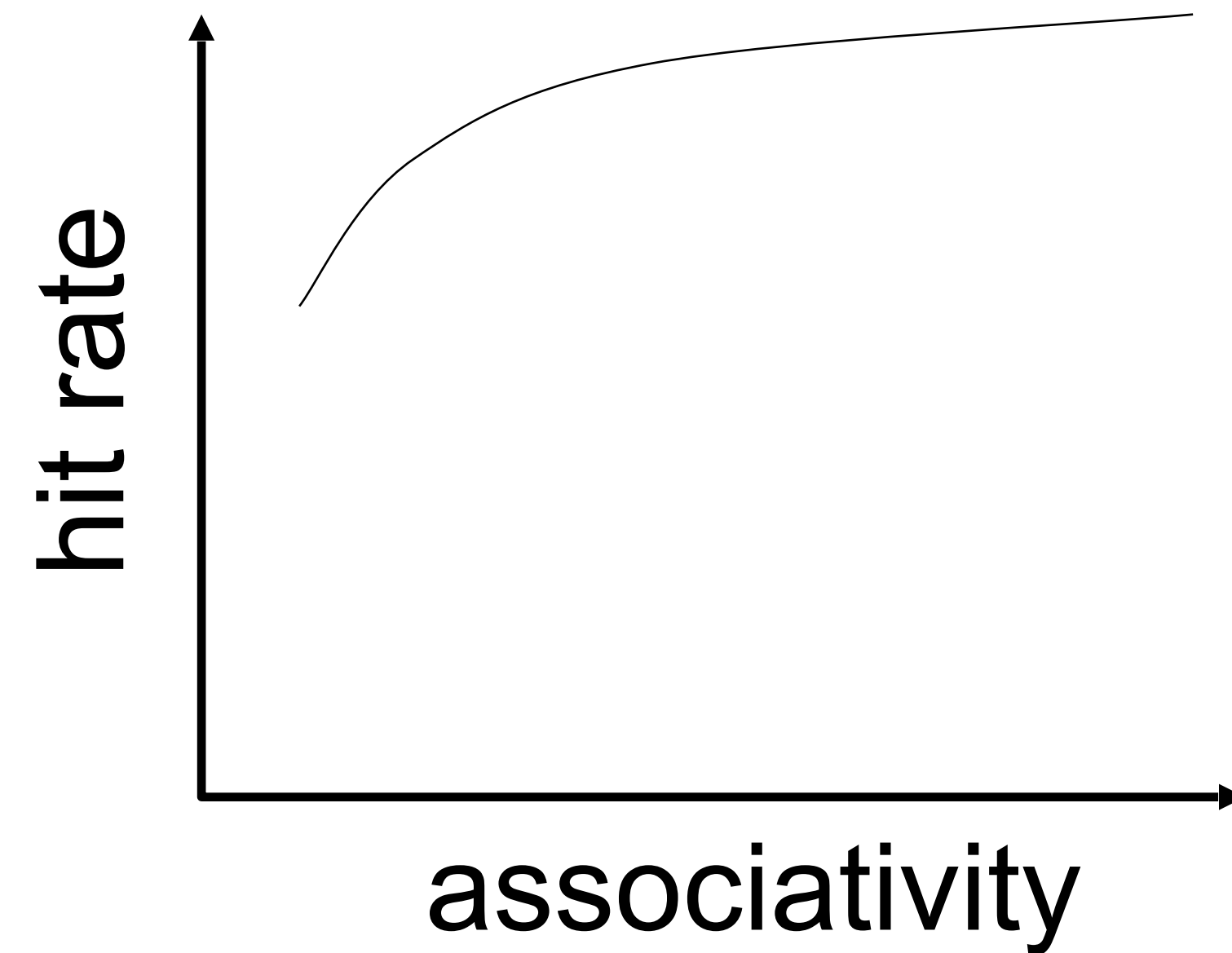


Cache size



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

Associativity



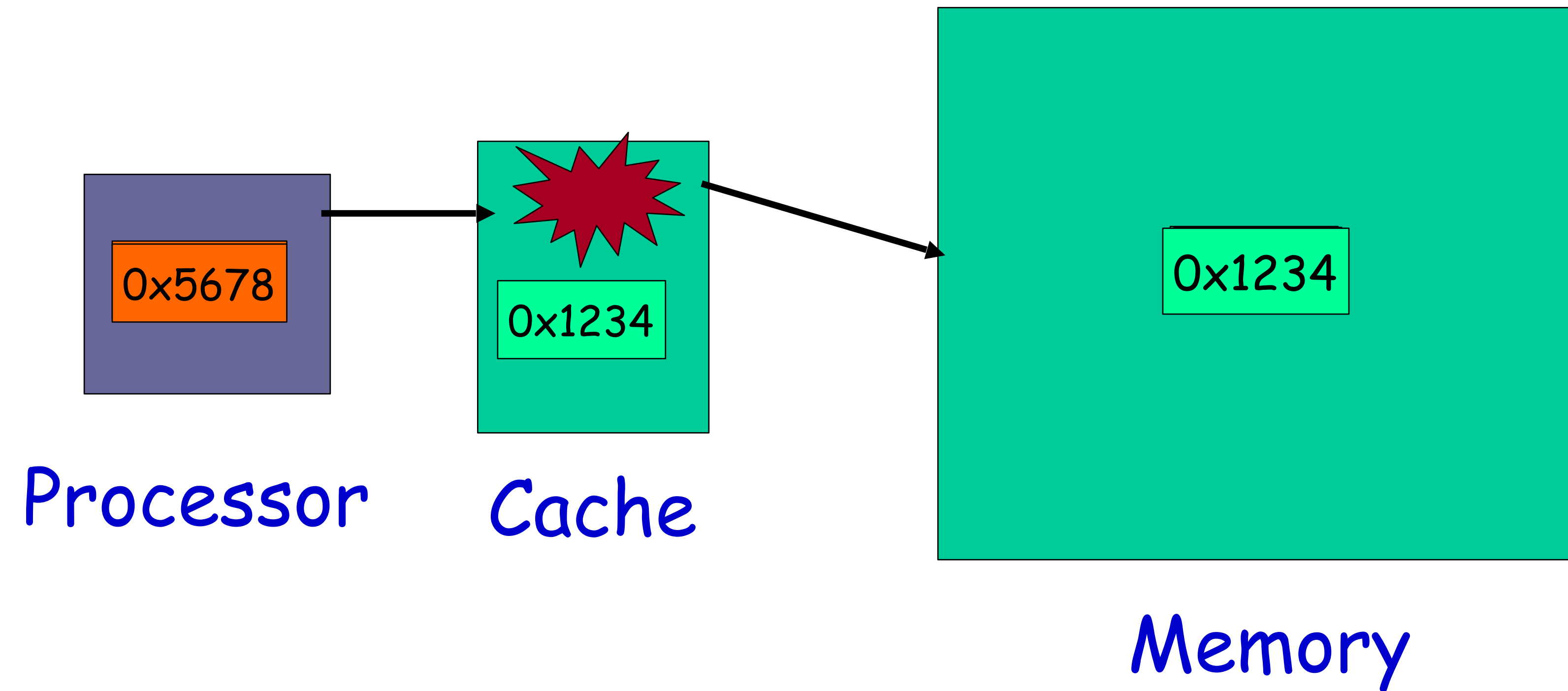
L1 cache: lower associativity, hit time

L3 cache: higher associativity

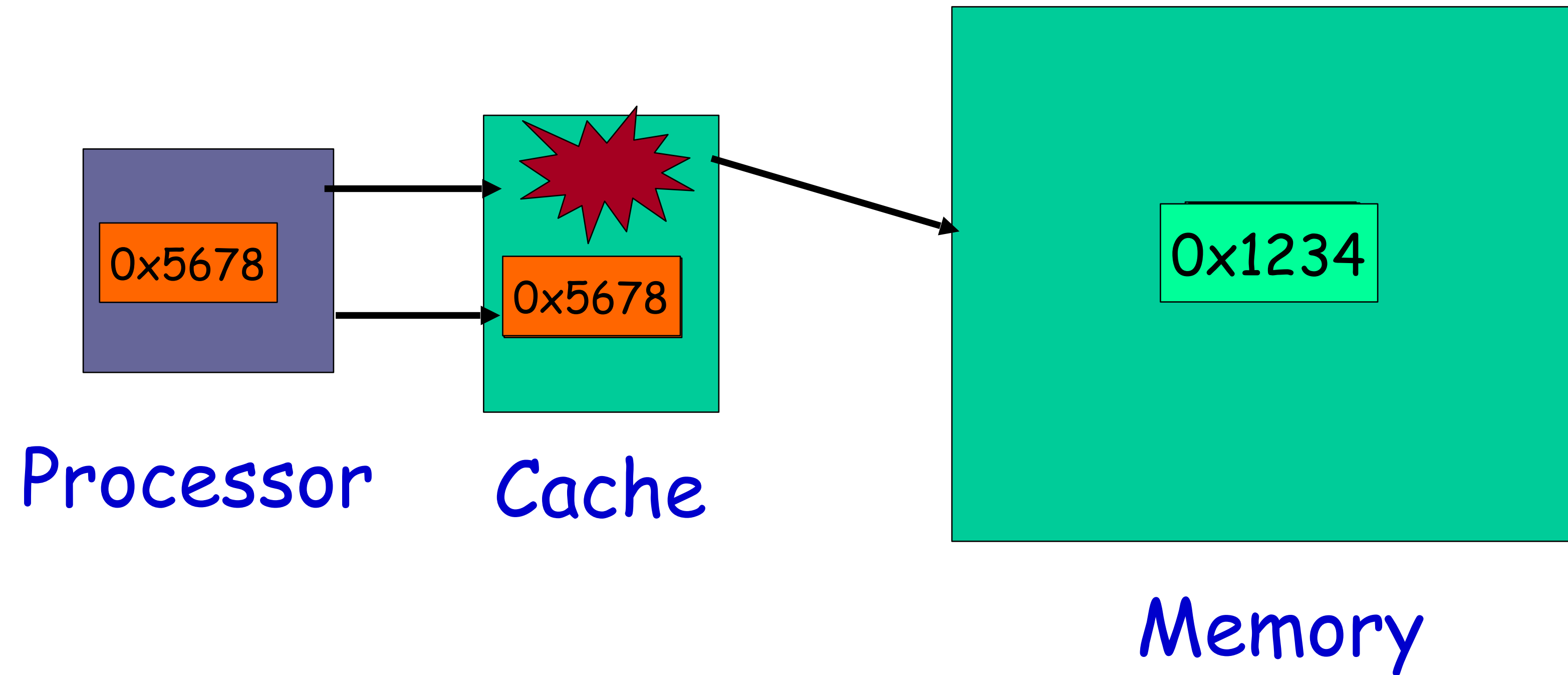
Cache Write Policies

- **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

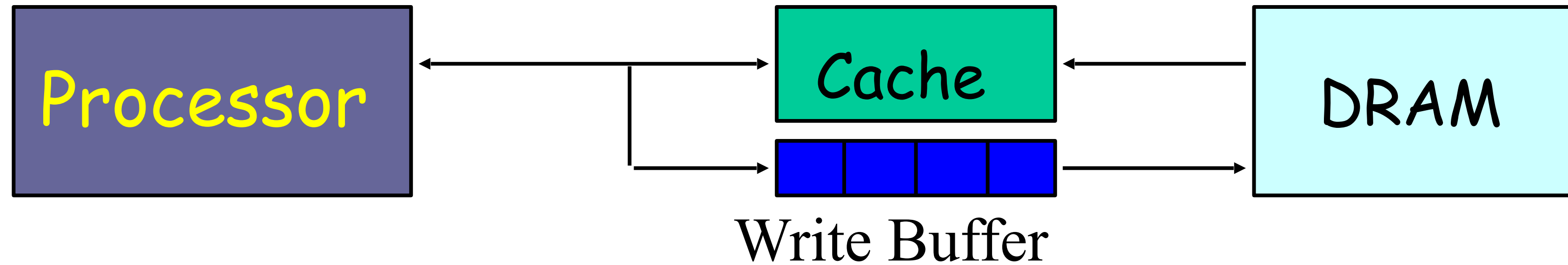
Write Allocate, No-write Allocate

- 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



- **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 $\ll$ DRAM write cycles
- **Memory system designer's nightmare:**
 - Write buffer saturation (i.e., Writes $\sim$ 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.
 - o 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

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

Memory Stall CPI

$$= \text{Miss per inst} \times \text{miss penalty}$$

$$= \% \text{ Memory Access/Instr} \times \text{Miss rate} \times \text{Miss Penalty}$$

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

$$\text{Memory Stall CPI} = 0.2 * 0.02 * 400 = 1.6 \text{ cycles}$$

CPU Performance with Memory Stall

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

$$\text{CPU time} = \text{IC} \times (\text{CPI}_{\text{execution}} + \text{CPI}_{\text{mem_stall}}) \times \text{Cycle Time}$$

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

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

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 1

- $$\begin{aligned} \text{CPI} &= \text{ideal CPI} + \text{average stalls per instruction} \\ &= 1.1(\text{cycles/ns}) + [0.30 (\text{DataMops/ins}) \\ &\quad \times 0.10 (\text{miss/DataMop}) \times 50 (\text{cycle/miss})] + [1 \\ &\quad (\text{InstMop/ins}) \\ &\quad \times 0.01 (\text{miss/InstMop}) \times 50 (\text{cycle/miss})] \\ &= (1.1 + 1.5 + .5) \text{ cycle/ins} = 3.1 \end{aligned}$$
- $$\text{AMAT} = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.1 \times 50] = 2.54$$

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?
 - Average memory access time?
 - CPI with and without cache?

Exercise 2:Solution

- Average memory accesses per instruction
= 1.2
- $AMAT = 1 * (1 + 0.01 * 100) + 0.2 * (1 + 0.01 * 100) = 2.4$ cycles
- Stall cycles = 1.2 cycles
- CPI with cache = $1 + 1.2 = 2.2$
- CPI without cache = $1 + 1.2 * 100 = 121$

Book

P & H, Chapter 4