Memory Hierarchy

15-740 FALL'21

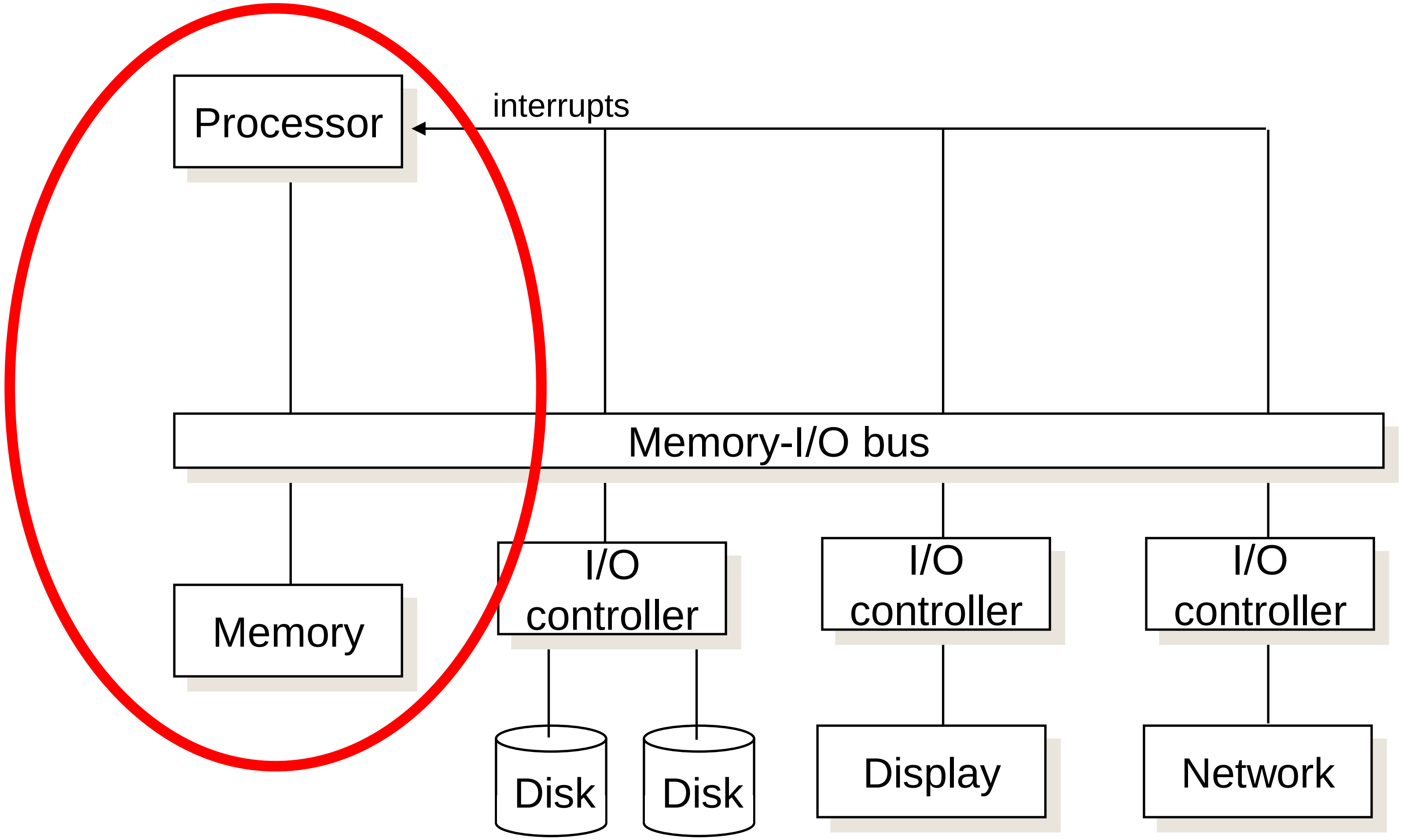
NATHAN BECKMANN

Topics

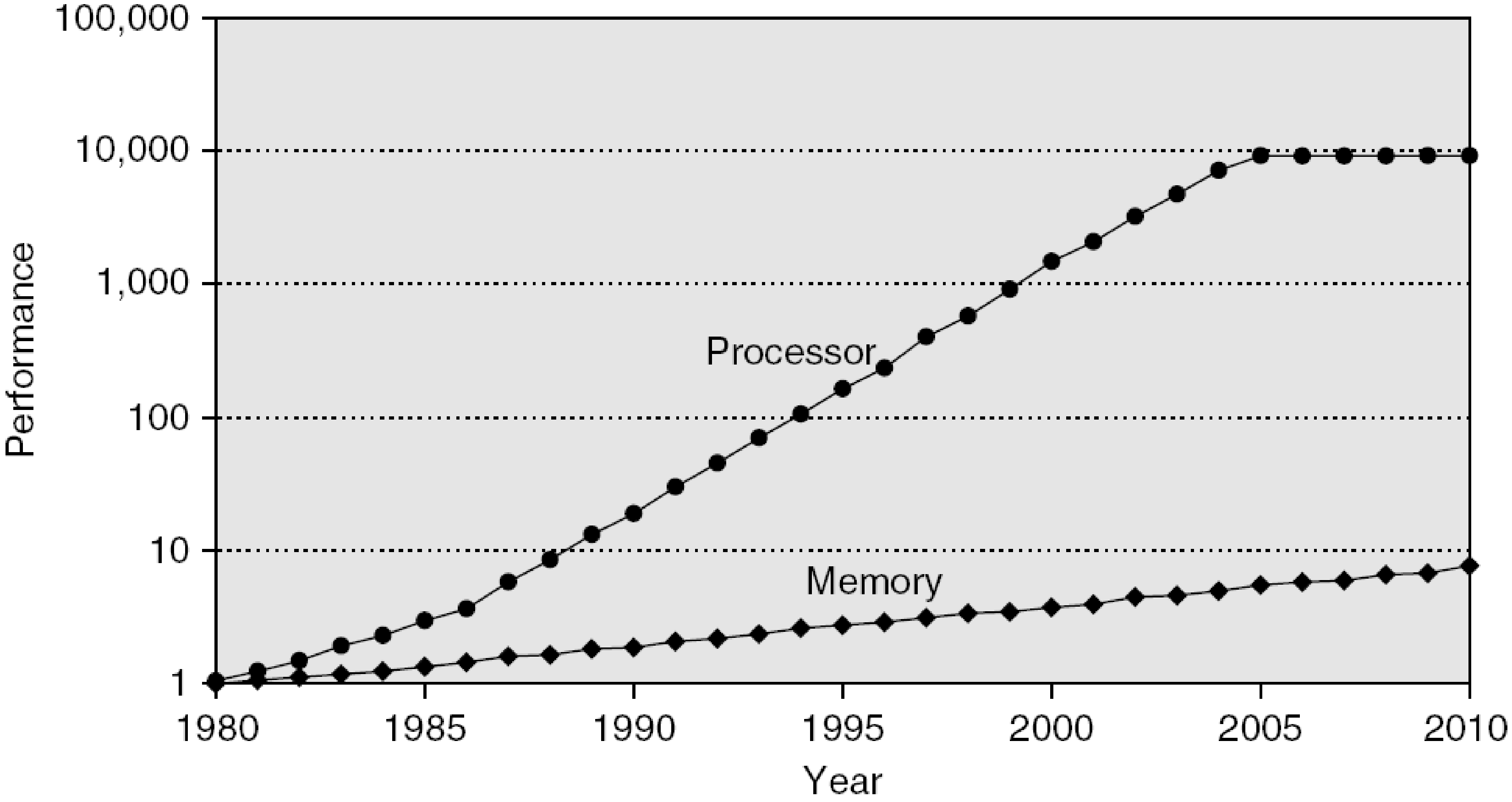
Memories

Caches

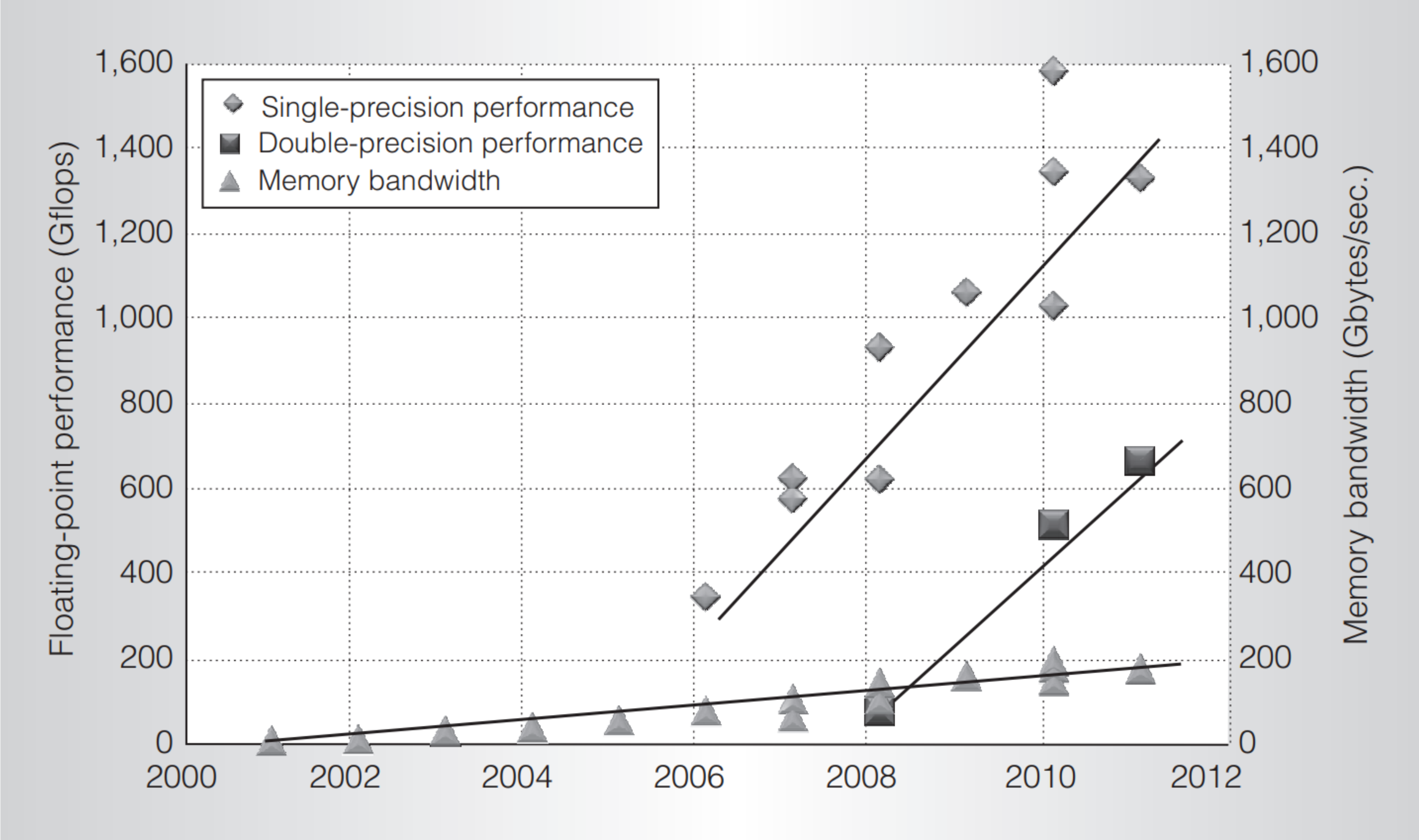
Early computer system



Recall: Processor-memory gap

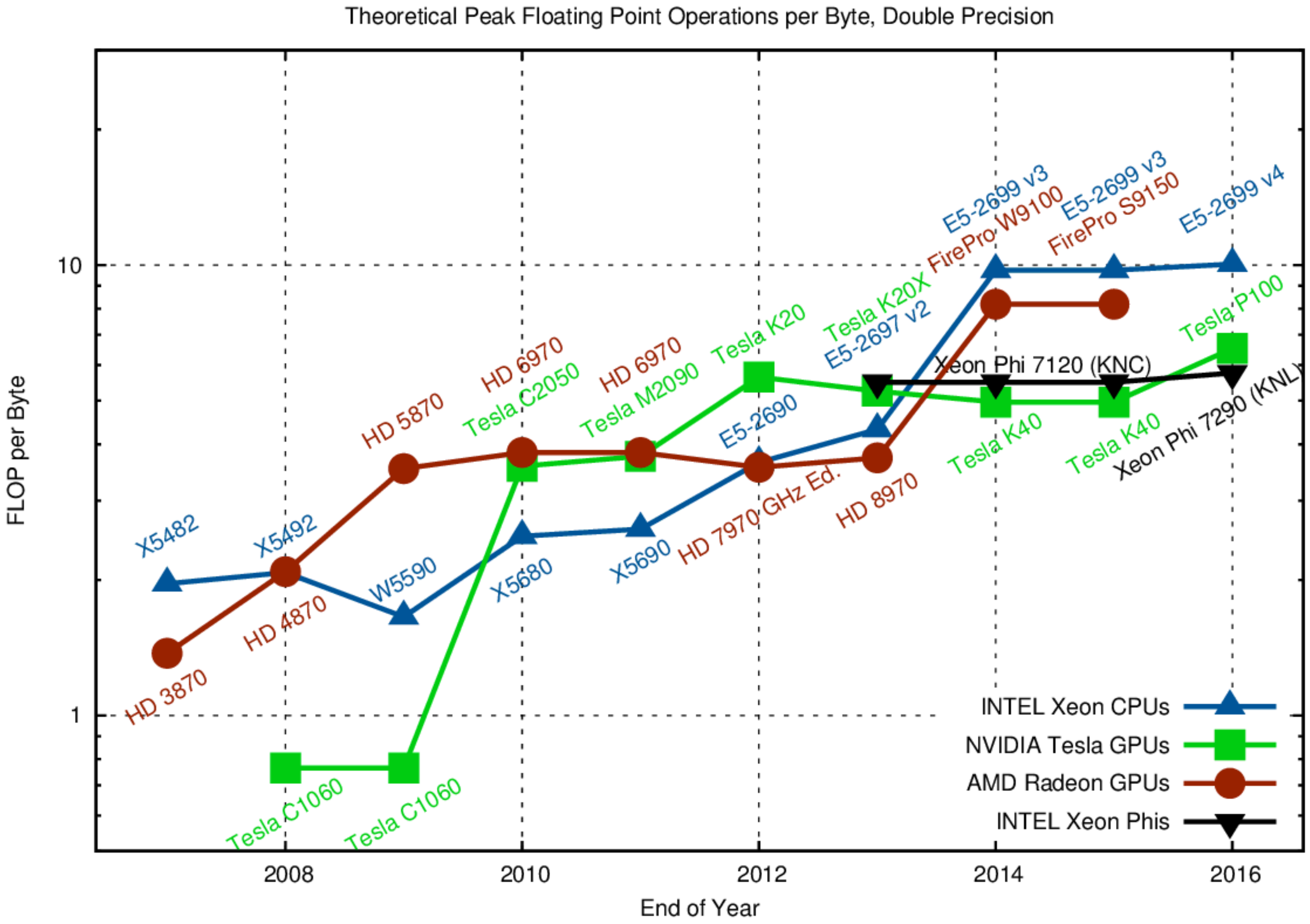


Processor-memory gap not just latency



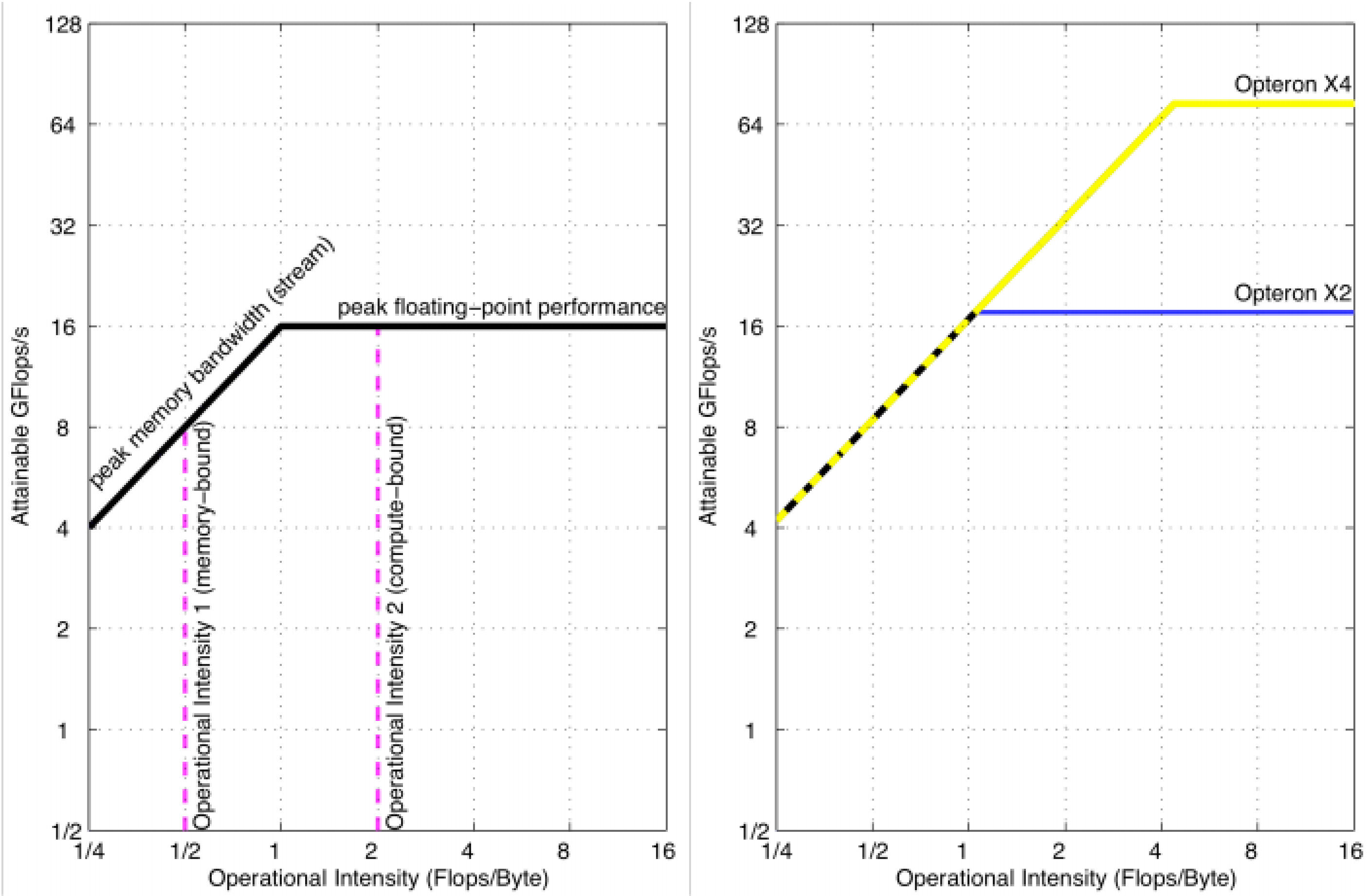
“GPUs and the Future of Parallel Computing” Bill Dally et al, IEEE MICRO 2011

Compute-per-memory is increasing fast



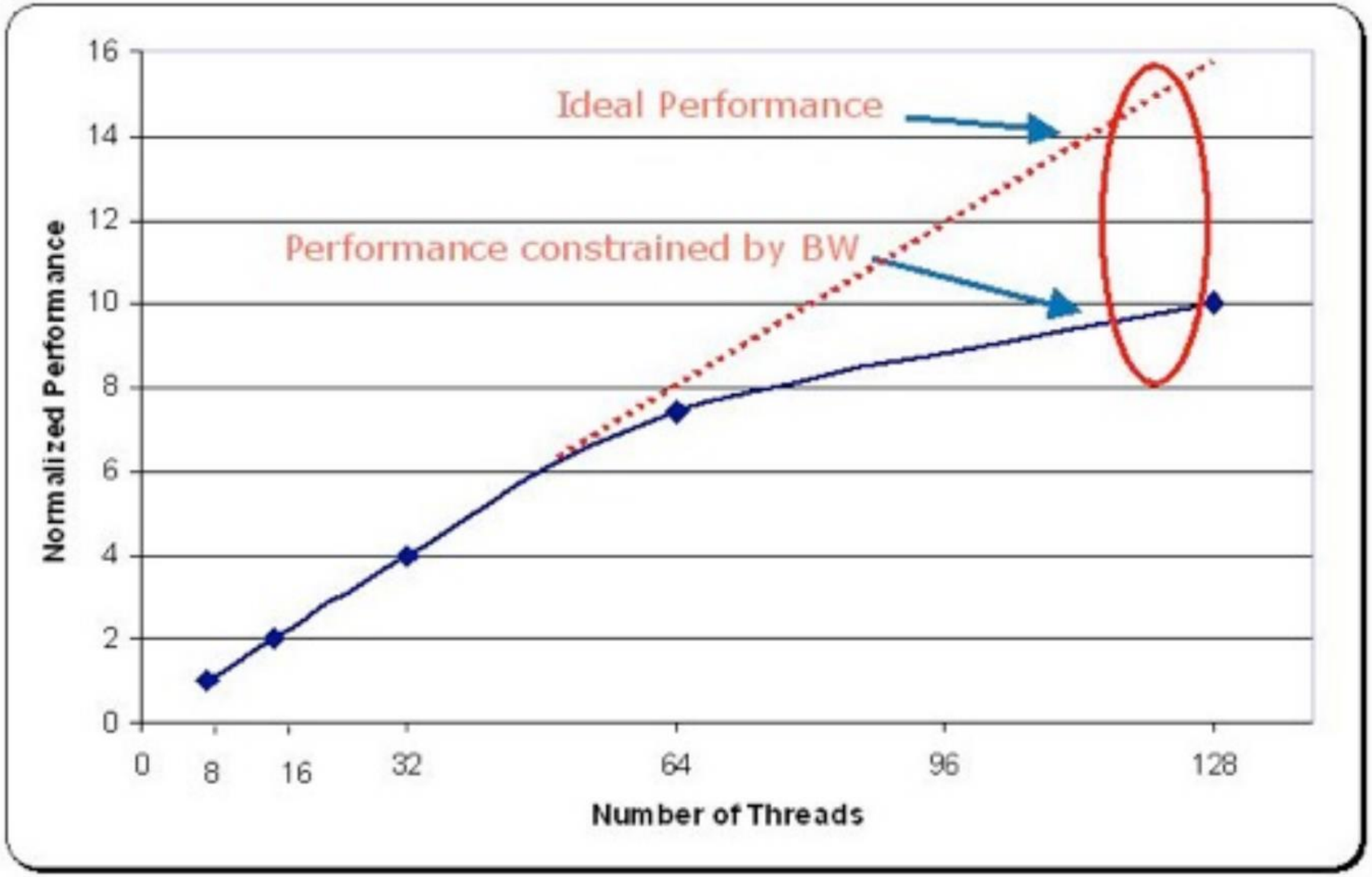
Karl Rupp, https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Processor often limited by memory



“The Roofline Model: An Insightful Visual Performance Model for Multicore Architectures” by Sam Williams et al, CACM’08

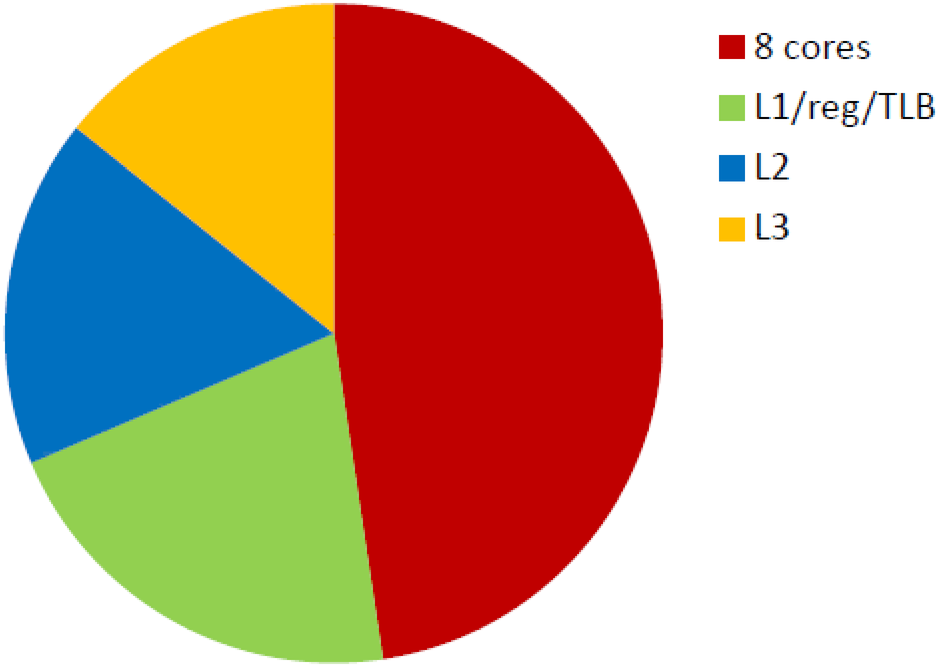
Processor often limited by memory



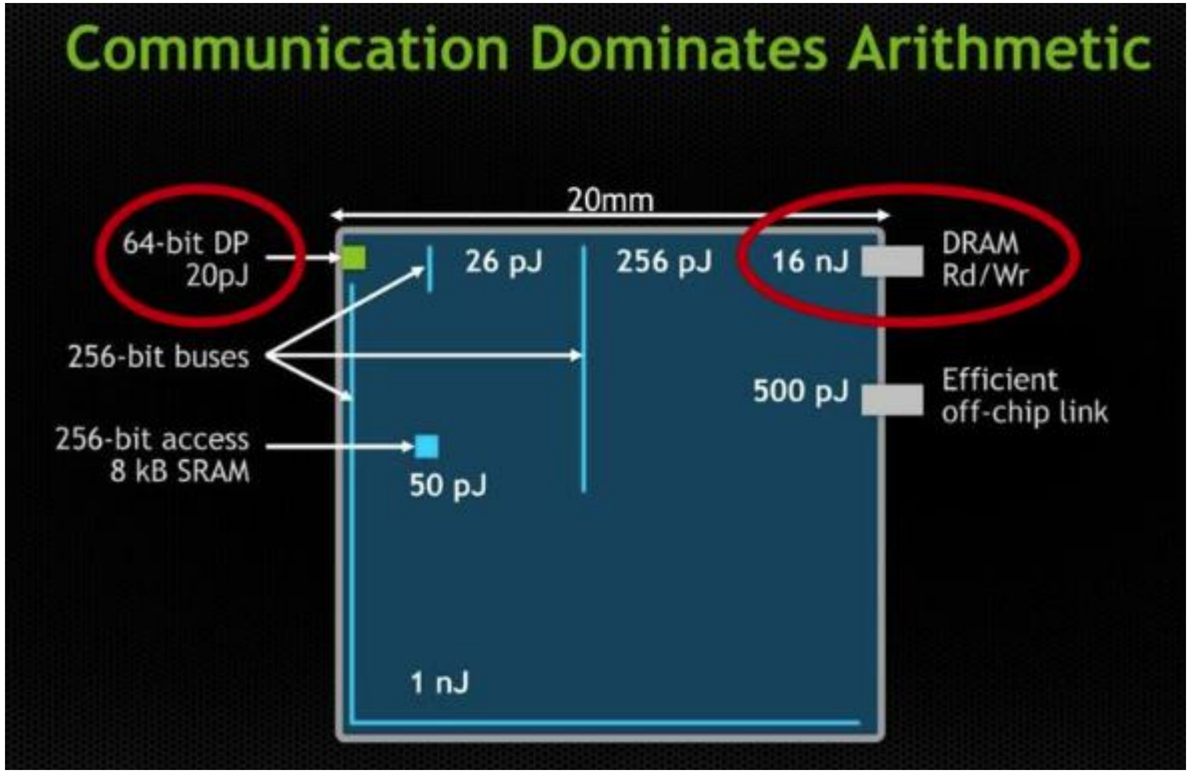
Bruce Jacob, https://ece.umd.edu/~blj/talks/Sun-Workshop.pdf

Most energy (power) spent on memory

8-core Xeon (worse w/ bigger chip)



A fundamental architectural challenge!



Credit: Horowitz, ISSCC’14 keynote; Dally, HiPeac’15 keynote

Ideal memory

We **want** a large, fast memory

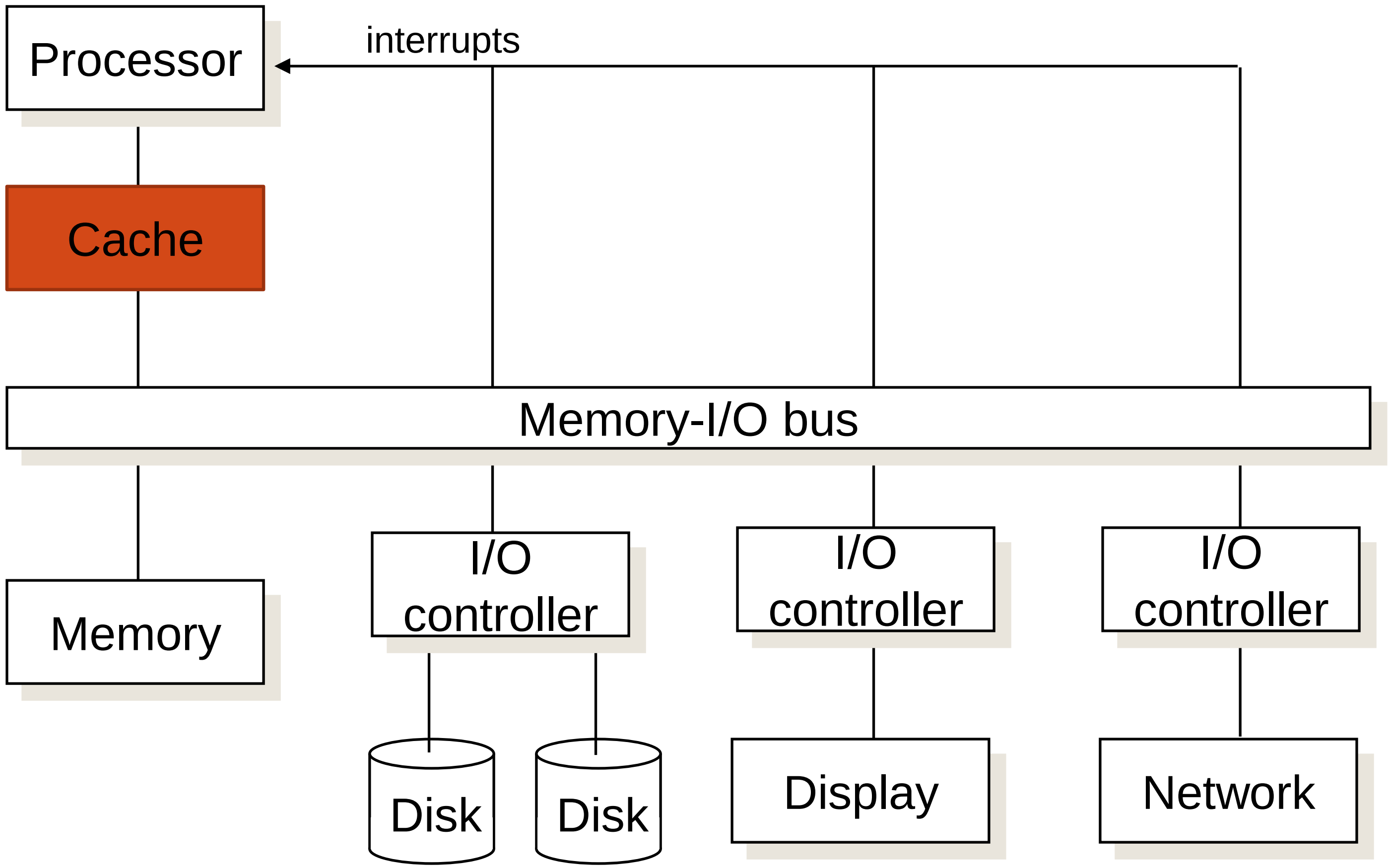
…But technology doesn't let us have this!

Key observation: **<u>Locality</u>**

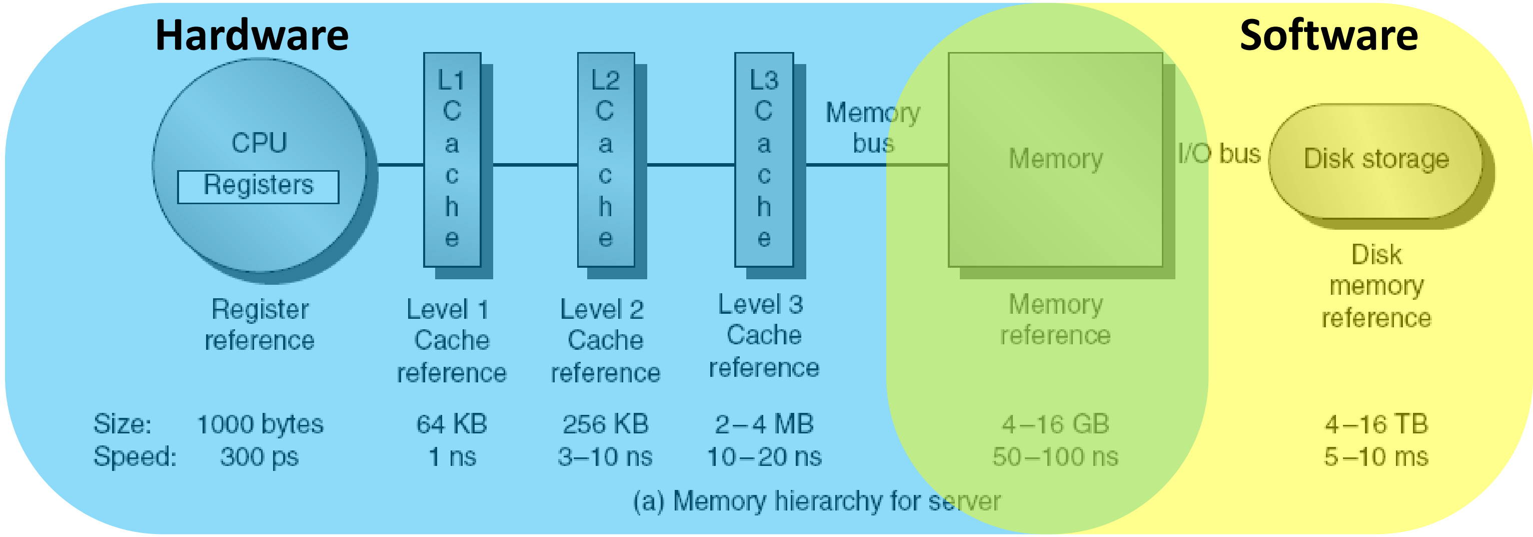
- All data are not equal
- Some data are accessed more often than others

Architects solution: Caches ➔ Memory **<u>hierarchy</u>**

Computer system



Technological tradeoffs in accessing data

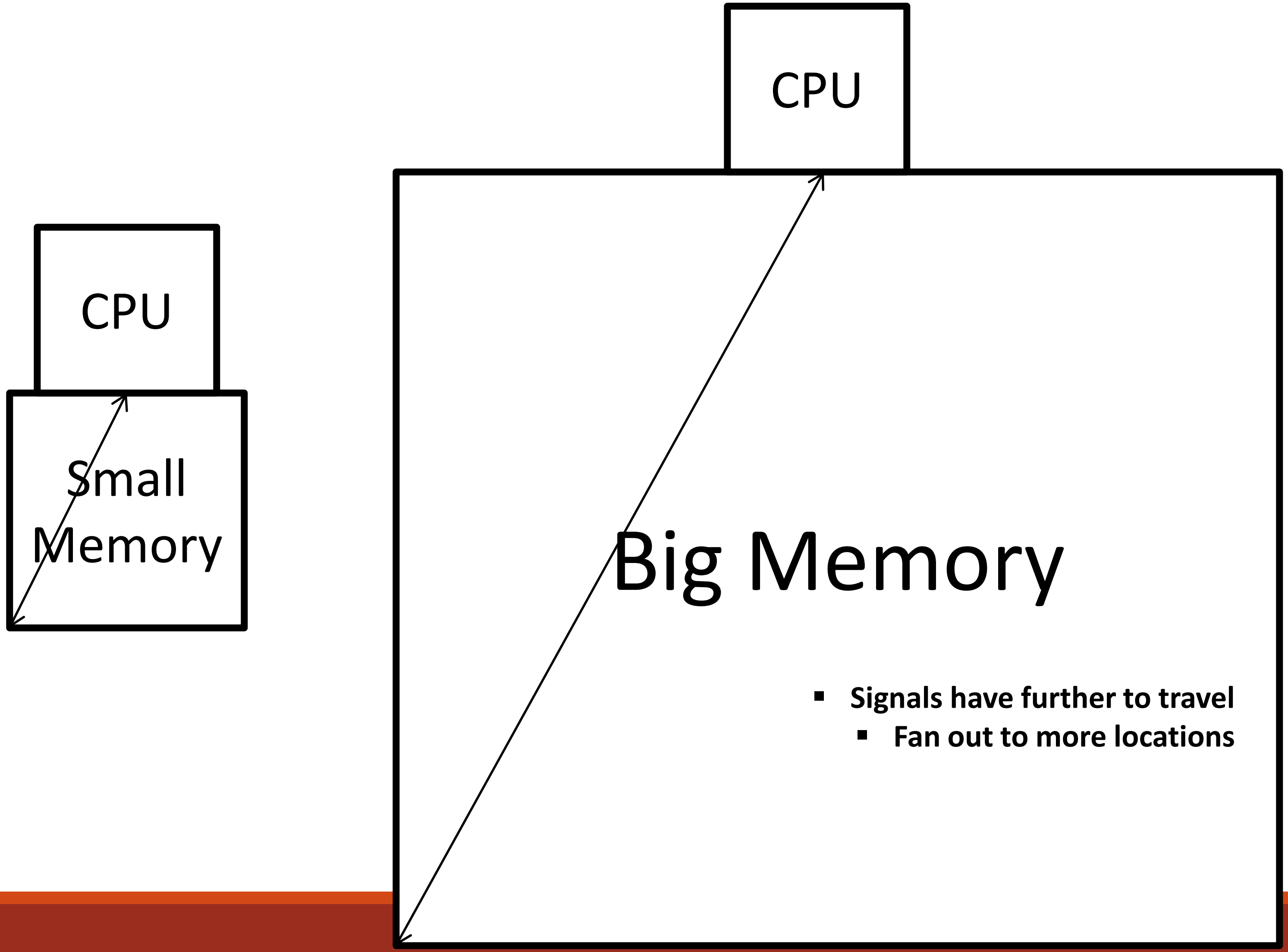


(a) Memory hierarchy for server

Why do hardware/software manage different parts of the hierarchy?

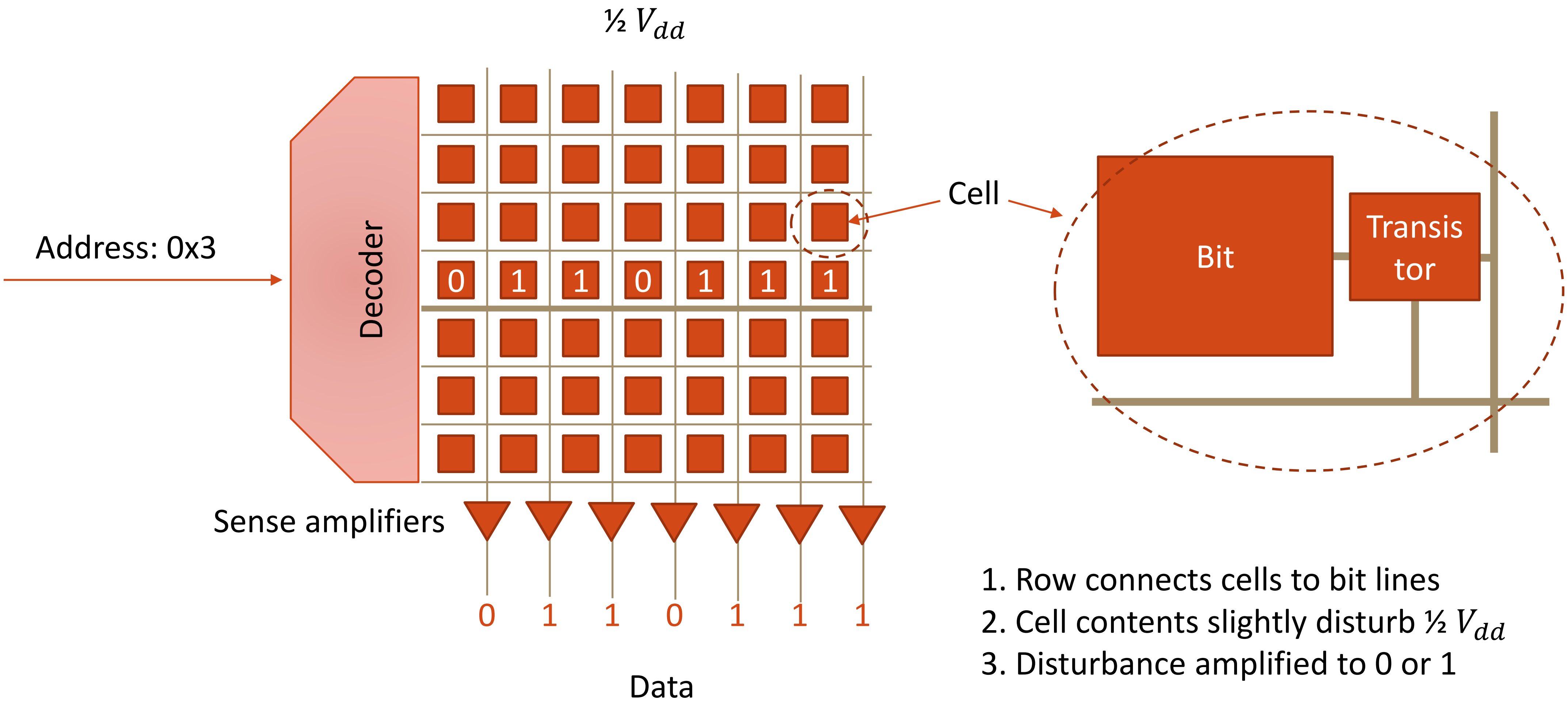
Memory Technology

Physical size affects latency



- Signals have further to travel
  - Fan out to more locations

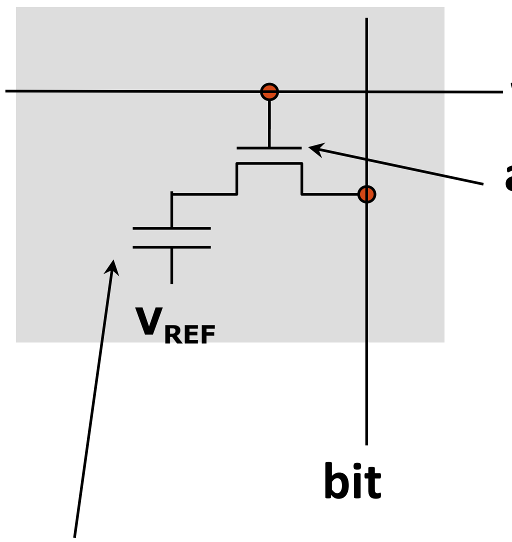
Generic memory design



1. Row connects cells to bit lines
2. Cell contents slightly disturb ½ V_{dd}
3. Disturbance amplified to 0 or 1

Single-transistor (1T) DRAM cell (one bit)

1-T DRAM Cell

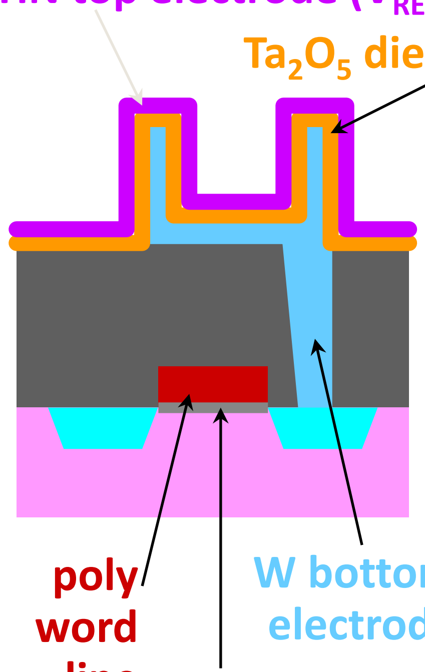


word

access transistor

bit

Storage capacitor (FET gate, trench, stack)

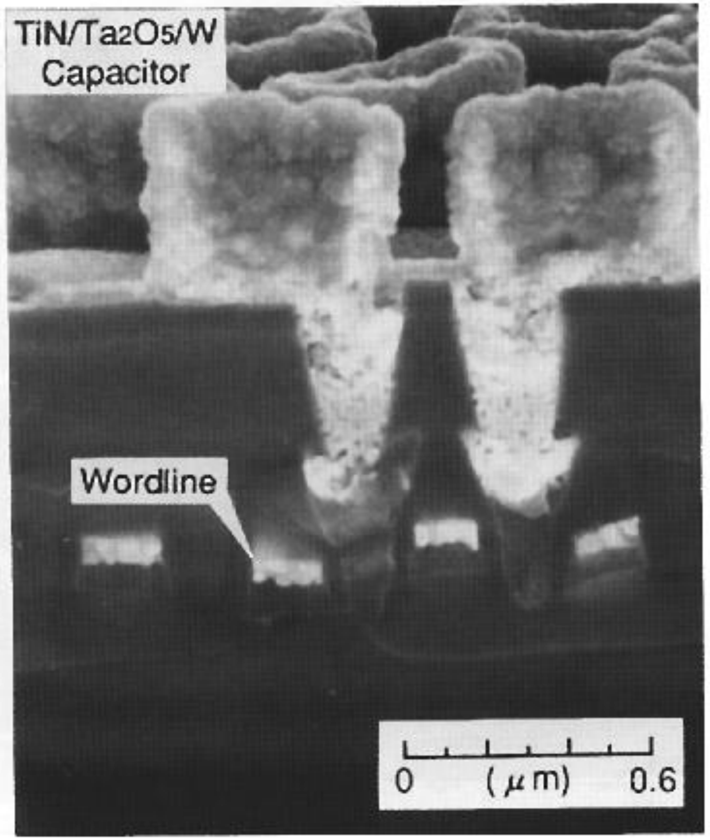

TiN top electrode (V_{REF})

Ta_2O_5 dielectric

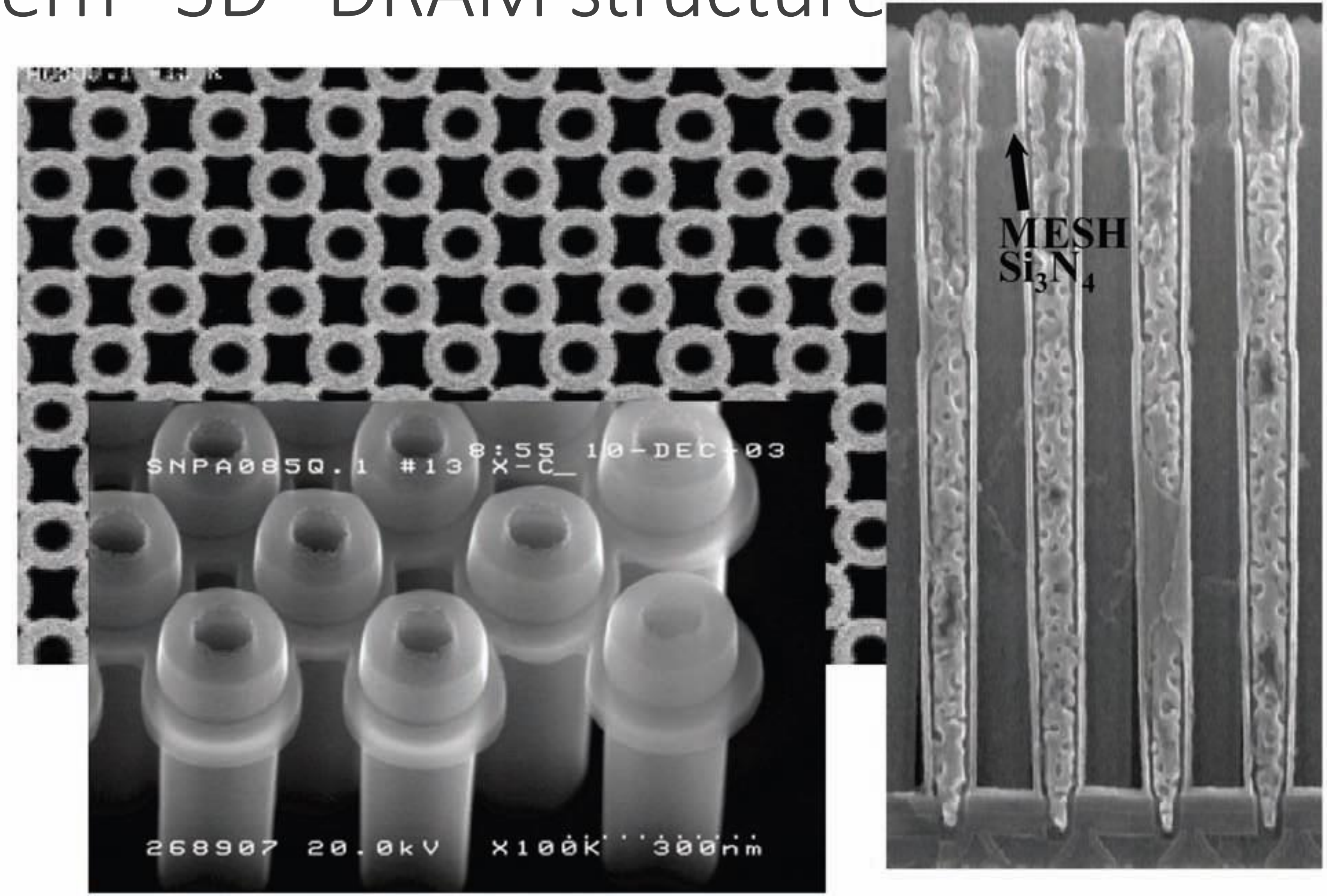
poly word line

W bottom electrode

access transistor



Modern “3D” DRAM structure



[Samsung, sub-70nm DRAM, 2004]

DRAM Physical Layout

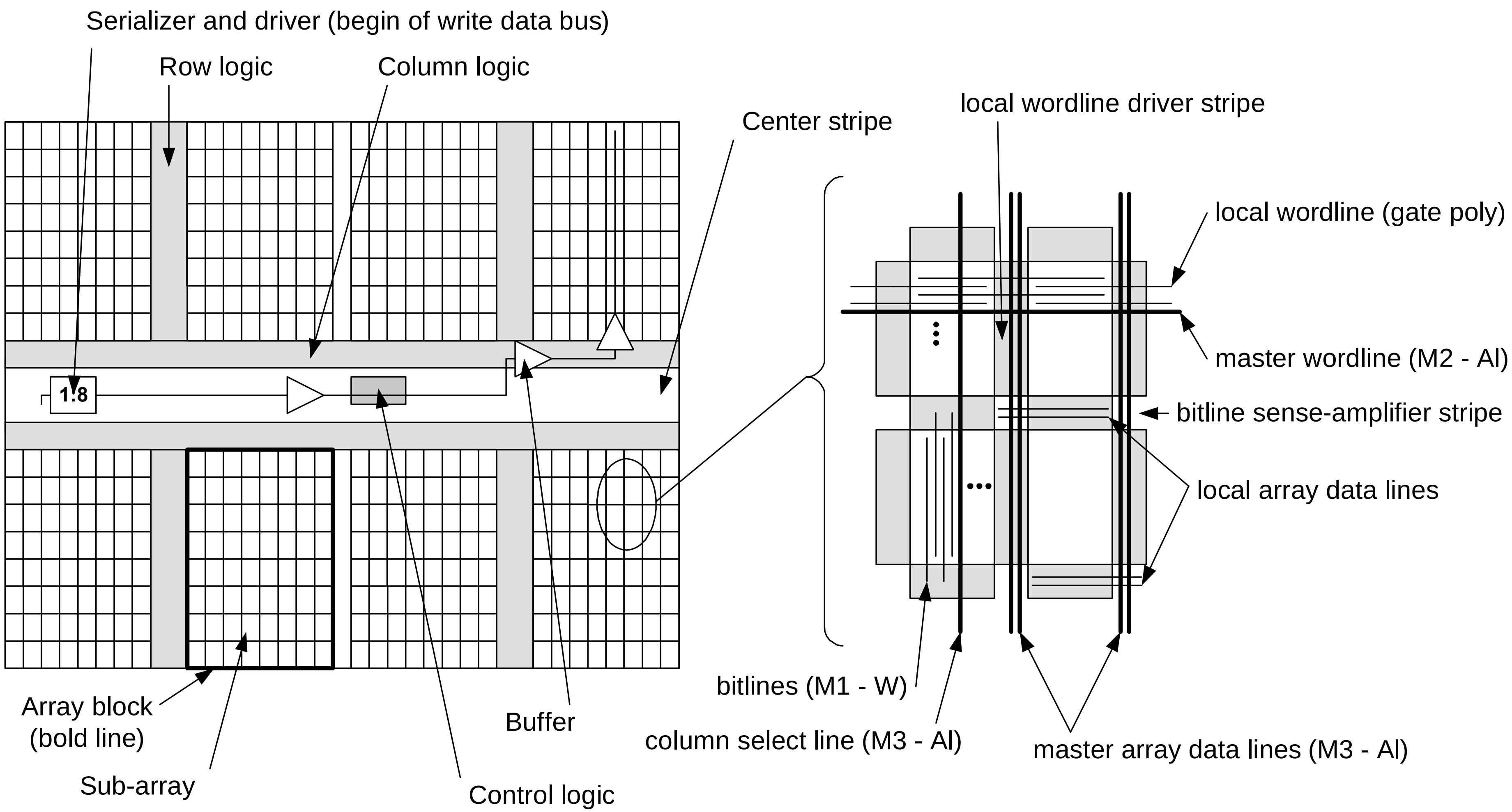

Serializer and driver (begin of write data bus)
Row logic
Column logic
Center stripe
local wordline driver stripe
local wordline (gate poly)
master wordline (M2 - Al)
bitline sense-amplifier stripe
local array data lines
1:8
Array block
(bold line)
Sub-array
Buffer
Control logic
bitlines (M1 - W)
column select line (M3 - Al)
master array data lines (M3 - Al)

Figure 1. Physical floorplan of a DRAM. A DRAM actually contains a very large number of small DRAMs called sub-arrays.

[Vogelsang, MICRO-2010]

DRAM operation

Three steps in read/write access to a given bank

- Precharge
- Row access (RAS)
- Column access (CAS)

Each step has a latency of around 10ns in modern DRAMs

Various DRAM standards (DDR, RDRAM) have different ways of encoding the signals for transmission to the DRAM, but all share same core architecture

DRAM Operation

Three steps in read/write access to a given bank

- Precharge
  - charges bit lines to known value, required before next row access
  - **write back open row** (DRAM reads are *destructive*)

- Row access (RAS)
- Column access (CAS)

Each step has a latency of around 10ns

DRAM Operation

Three steps in read/write access to a given bank

- Precharge

- Row access (RAS)
  - decode row address, enable addressed row (often multiple Kb in row)
  - bitlines share charge with storage cell
  - small change in voltage detected by sense amplifiers which latch whole row of bits
  - sense amplifiers drive bitlines full rail to recharge storage cells

- Column access (CAS)

Each step has a latency of around 10ns

DRAM Operation

Three steps in read/write access to a given bank

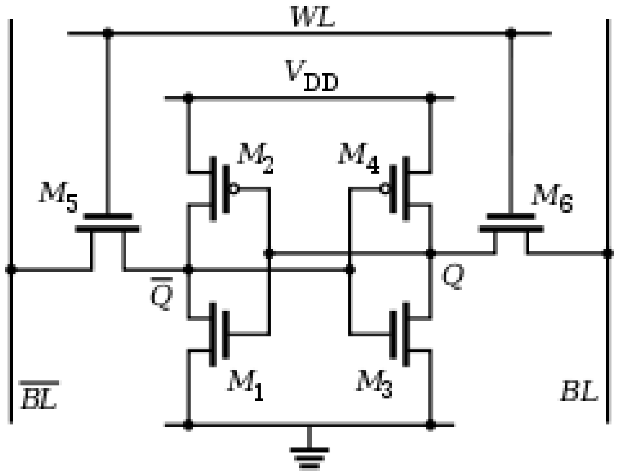
- Precharge
- Row access (RAS)

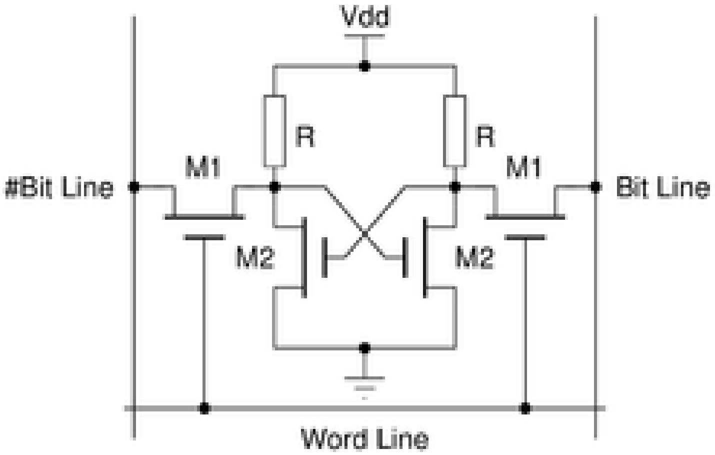
- Column access (CAS)
  - decode column address to select small number of sense amplifier latches (4, 8, 16, or 32 bits depending on DRAM package)
  - on read, send latched bits out to chip pins
  - on write, change sense amplifier latches which then charge storage cells to required value
  - **can perform multiple column accesses on same row without another row access** (burst mode / row buffer locality)

Each step has a latency of around 10ns

Static RAM cell (one bit)

Different varieties based on # transistors





Fewer transistors ➔ more bits / mm^2, but harder to manufacture

- An SRAM cell is *much* bigger than a DRAM cell (≈ 100 ×)

Standby: M5 & M6 disconnected, M1-M4 make self-reinforcing inverters

Read: connect M5 & M6, sense + amplify signal on bitlines

Write: connect M5 & M6, bias bitlines to desired value

Memory parameters

Density

- Bits / mm^2

Latency

- Time from initiation to completion of one memory read (e.g., in nanoseconds, or in CPU or DRAM clock cycles)

Bandwidth

- Rate at which requests can be processed (accesses/sec, or GB/s)

Occupancy

- Time that a memory bank is busy with one request (esp. writes)
- *Not the same as latency – inversely related to bandwidth*

Energy

Performance can vary significantly for reads vs. writes, or address, or access history

SRAM vs DRAM

	SRAM	DRAM
Simplicity		
Latency		
Bandwidth		
Density		

Q: *When does an architect use DRAM? SRAM?*

SRAM vs DRAM

	SRAM	DRAM
Simplicity	✓	
Latency	✓	
Bandwidth	✓	
Density		✓

Q: *When does an architect use DRAM? SRAM?*

SRAM used for on-chip caches, register file

DRAM used for main memory

- Often with a different manufacturing process, optimized for density not speed
- That’s why single chips with main memory + logic are rare
- “3D stacking” is changing this (kind of)

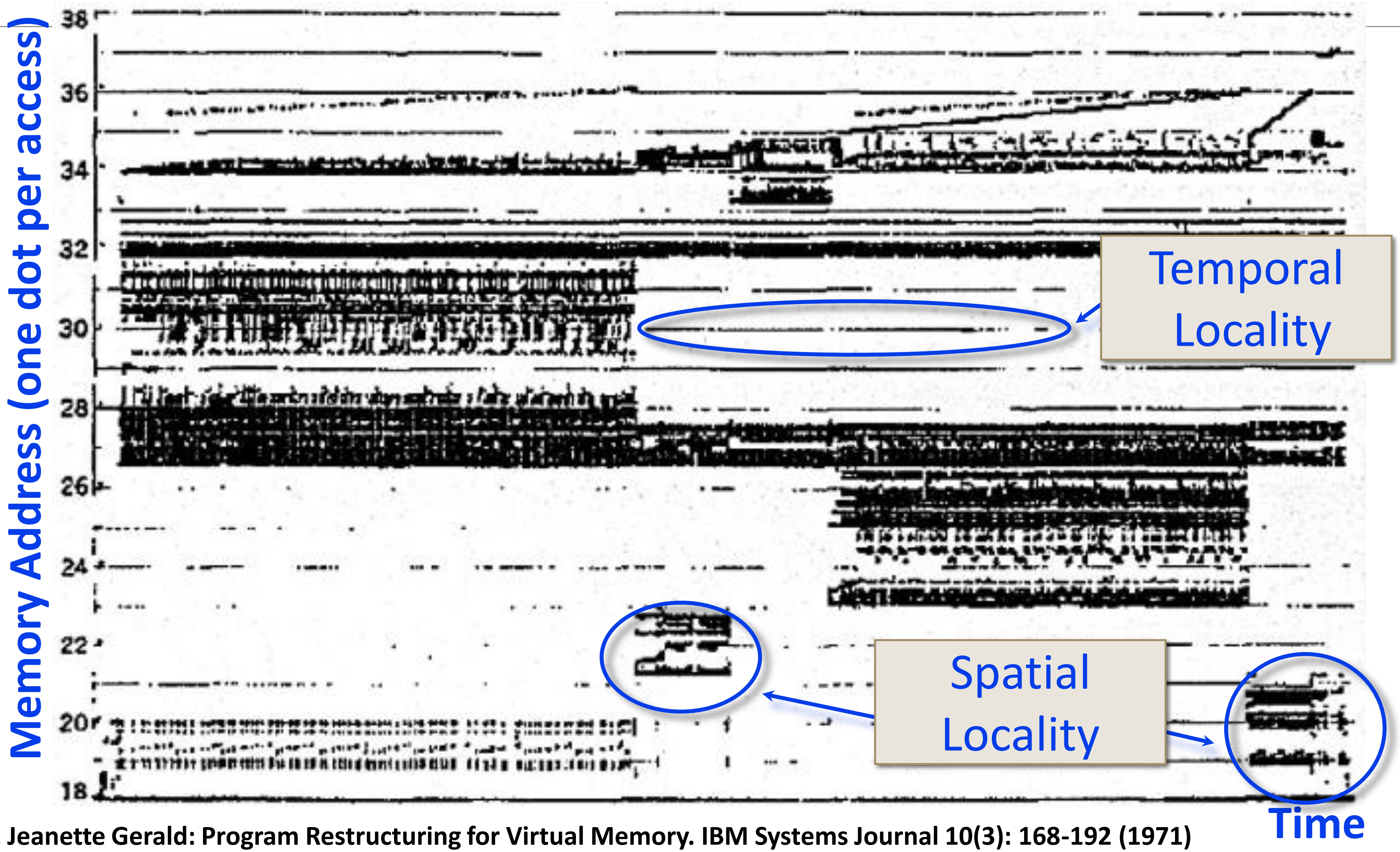
Memory Hierarchy

Why does memory hierarchy work?

Temporal Locality: If a location is referenced it is likely to be referenced again in the near future.

Spatial Locality: If a location is referenced it is likely that locations near it will be referenced in the near future.

Memory Reference Patterns



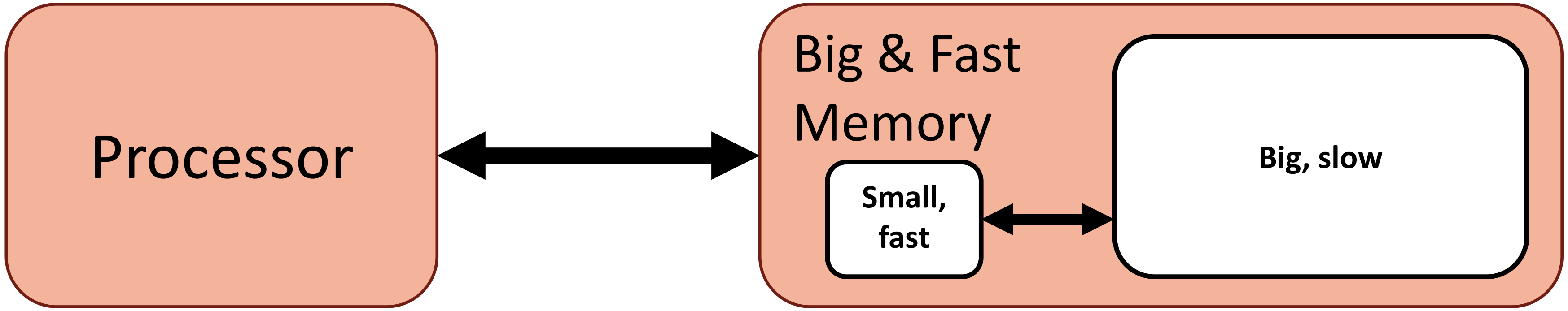
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Memory hierarchy

Implement memories of different sizes to serve different latency / latency / bandwidth tradeoffs

Keep frequently accessed data in small memories & large datasets in large memories

Provides illusion of a large & fast memory



Design choice #1: Cache vs Memory

How to manage the hierarchy?

As memory (aka "scratchpads"): software must be aware of different memories and use them well

- In theory: most efficient
- In practice: inconvenient and difficult except for highly specialized arch (eg, IBM "Cell" in PS3)

As cache: transparent to software; hardware moves data between levels of memory hierarchy

- In theory: overheads and performance loss
- In practice: convenient and h/w does a good job (with software help)

Cache vs Memory in real systems

Small/fast storage, e.g., registers

- Address usually specified in instruction
- Generally implemented directly as a register file
  - *…But hardware might do things behind software's back, e.g., stack management, register renaming, …*

Larger/slower storage, e.g., main memory

- Address usually computed from values in register
- Generally implemented as a hardware-managed cache hierarchy (hardware decides what is kept in fast memory)
  - *…But software may provide "hints", e.g., prefetch or don't cache*

Design choice #2: Instructions vs data

Where to store instructions & data?

Harvard architecture:

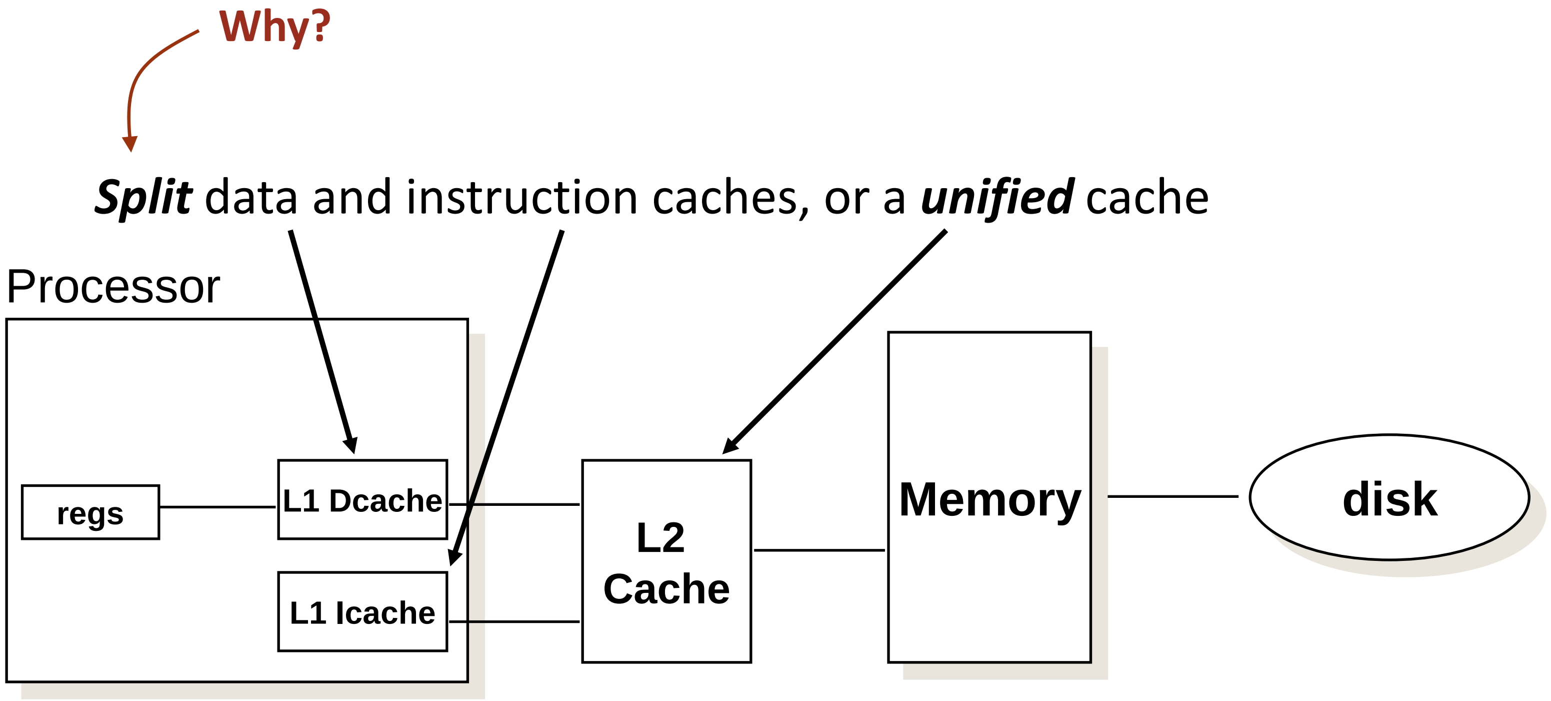
- In early machines, instructions were hard-wired (switchboards) or punchcards
- Data was kept in memory

Princeton/von Neumann architecture:

- Instructions and data are both in memory
- “Instructions are data”

Modern architecture:

Split vs. unified caches

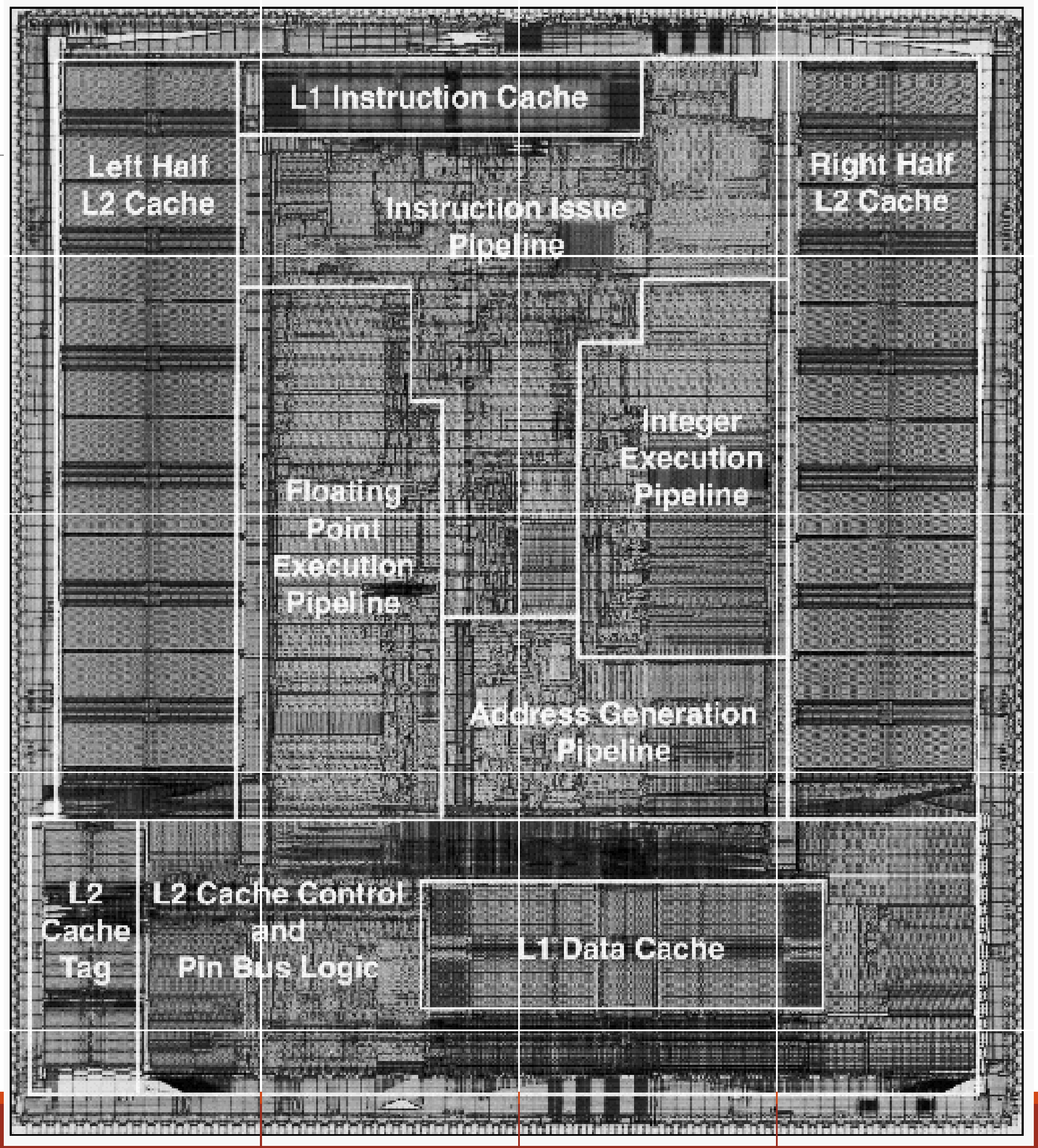

Why?
Split data and instruction caches, or a ***unified*** cache
Processor
regs
L1 Dcache
L1 Icache
L2 Cache
Memory
disk

Alpha 21164

Microprocessor Report 9/12/94

Caches:

- L1 data
- L1 instruction
- L2 unified
- + L3 **off-chip**

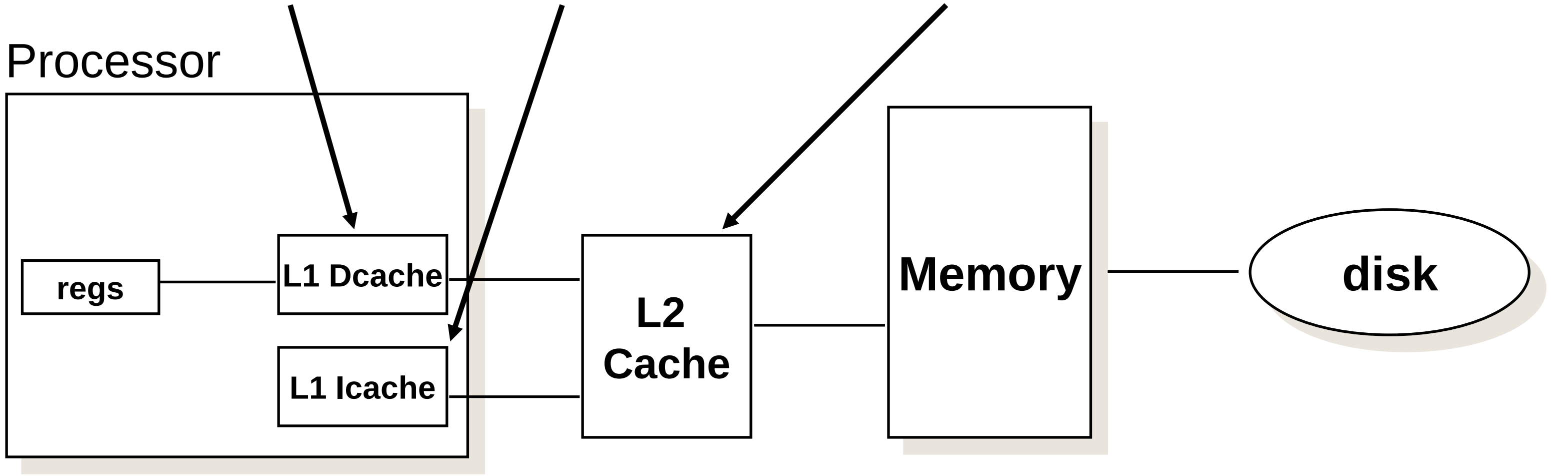


L1 Instruction Cache
Left Half L2 Cache
Instruction Issue Pipeline
Right Half L2 Cache
Integer Execution Pipeline
Floating Point Execution Pipeline
Address Generation Pipeline
L2 Cache Tag
L2 Cache Control and Pin Bus Logic
L1 Data Cache

Split vs. unified caches

Why?

Split data and instruction caches, or a ***unified*** cache



How does this affect self-modifying code?

Caches exploit locality

Temporal locality:

- Hardware decides what to keep in cache
- *Replacement/eviction policy* evicts a *victim* upon a cache miss to make space
- Least-recently used (LRU) most common eviction policy

Spatial locality:

- Cache stores multiple, neighboring words per *block*

  Also amortizes overheads for tracking state.
- *Prefetchers* speculate about next accesses and fetch them into cache

Note: Cache contents are <u>not</u> "architectural" ➔ *Guessing is fine!*

Example: Locality of reference

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently.
- *Temporal locality:* recently referenced items are likely to be referenced in the near future.
- *Spatial locality:* items with nearby addresses tend to be referenced close together in time.

Locality in Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

- **Data**
  - – Reference array elements in succession (spatial)
  - – sum and index variable (temporal, allocated to register)
- **Instructions**
  - – Reference instructions in sequence (spatial)
  - – Cycle through loop repeatedly (temporal)

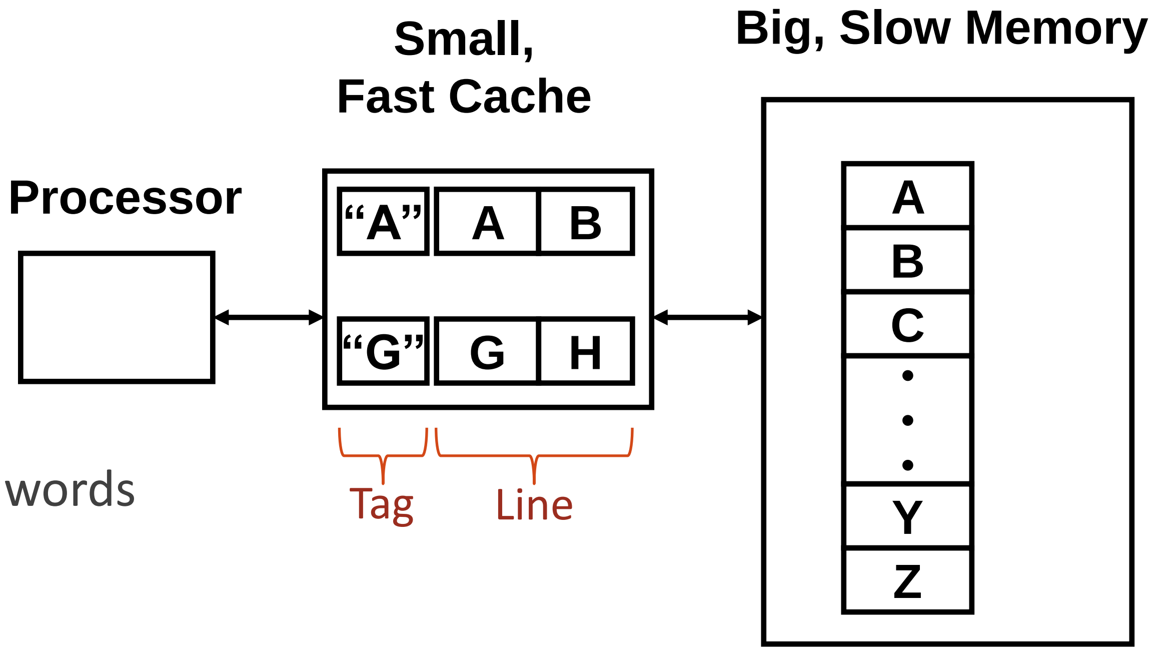
Caching: The basic idea

Main Memory

- Stores words
  - A–Z in example

Cache

- Stores subset of next level
  - E.g., ABGH in example
  - An **inclusive** hierarchy
  - **Tags** track what's in the cache
- Organized in **lines/blocks** of multiple words
  - Exploit spatial locality
  - Amortize overheads



Access

- Processor requests address from cache, which handles misses **itself**
- *What happens when processor accesses A? B? C?*

Cache metrics

$$\text{Miss ratio} = \frac{\text{Misses}}{\text{Memory accesses}}$$

$$\text{MPKI} = \frac{\text{Misses}}{\text{KiloInstruction}} = \frac{\text{Miss ratio} \times \text{Memory accesses}}{\text{Instructions}} \times 1000$$

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

➔ Three ways to improve memory performance:

1. Reduce hit time
2. Reduce miss ratio
3. Reduce miss penalty

…As always, there’s a tension between these

MPKI and AMAT in parallel programs

$$\text{Miss ratio} = \frac{\text{Misses}}{\text{Memory accesses}}$$

$$\text{MPKI} = \frac{\text{Misses}}{\text{KiloInstruction}} = \frac{\text{Miss ratio} \times \text{Memory accesses}}{\text{Instructions}} \times 1000$$

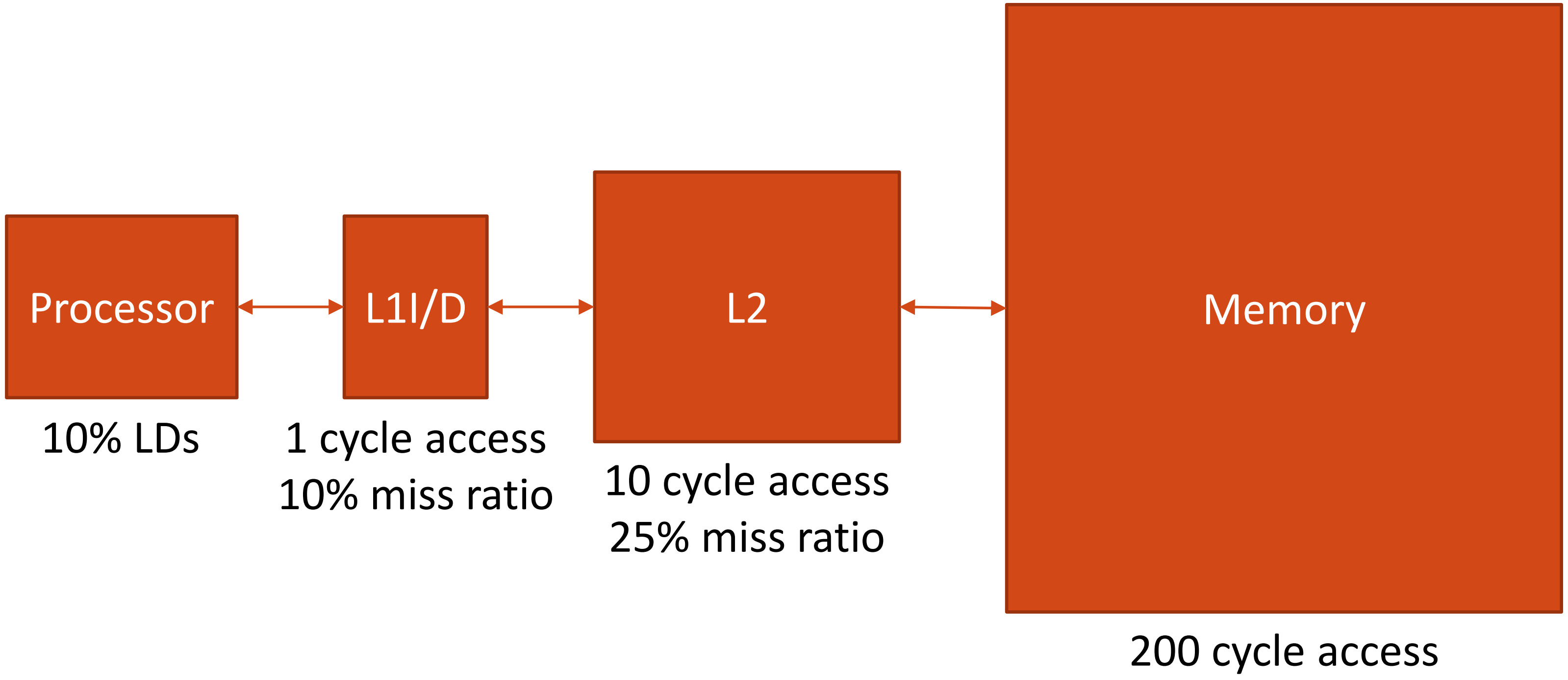
$$\text{AMAT} = \text{Average memory access time} = \text{Hit time} + \text{Miss ratio} \times \text{Miss penalty}$$

Important caveat: Processors can execute other instructions during a miss

- “Hide” memory latency
- Reduces performance impact of misses
- Memory-level parallelism (MLP)

Much more on this later in the semester!

AMAT example



Memory AMAT = 200 cycles

L2 AMAT = 10 cycles + 0.25 * 200 = 60 cycles

L1 AMAT = 1 cycle + 0.10 * 60 cycles = 7 cycles

Memory CPI = (1 + 0.10) * 7 = 7.7 cycles

Impact of increasing cache size?

- Effect on cache area (tags + data)?
- Effect on hit time?
- Effect on miss ratio?
- Effect on miss penalty?

Self-check questions

What parts of a memory hierarchy (registers, L1i, L1d, private L2, shared L3, memory) are *architectural*?

Why is DRAM used for main memory?

Some processors also use DRAM for their “last-level” shared cache, especially as memory bandwidth grows scarcer. Why?

A common way to avoid unnecessary data movement is *remote memory operations* (RMOs) that perform some simple computation (e.g., an increment) in the cache itself, without moving data to the core. Are RMOs more natural in a CISC or a load-store RISC ISA?

Design issues for caches

Key Questions:

- Where should a line be placed in the cache? (line placement)
- How is a line found in the cache? (line identification)
- Which line should be replaced on a miss? (line replacement)
- What happens on a write? (write strategy)

Constraints:

- Design must be simple
    - Hardware realization
    - All decision making within nanosecond time scale
- Want to optimize performance for “typical” programs
    - Do extensive benchmarking and simulations
    - Many subtle engineering tradeoffs

Fully associative cache

Mapping of Memory Lines

- *Any address can map to any cache line*

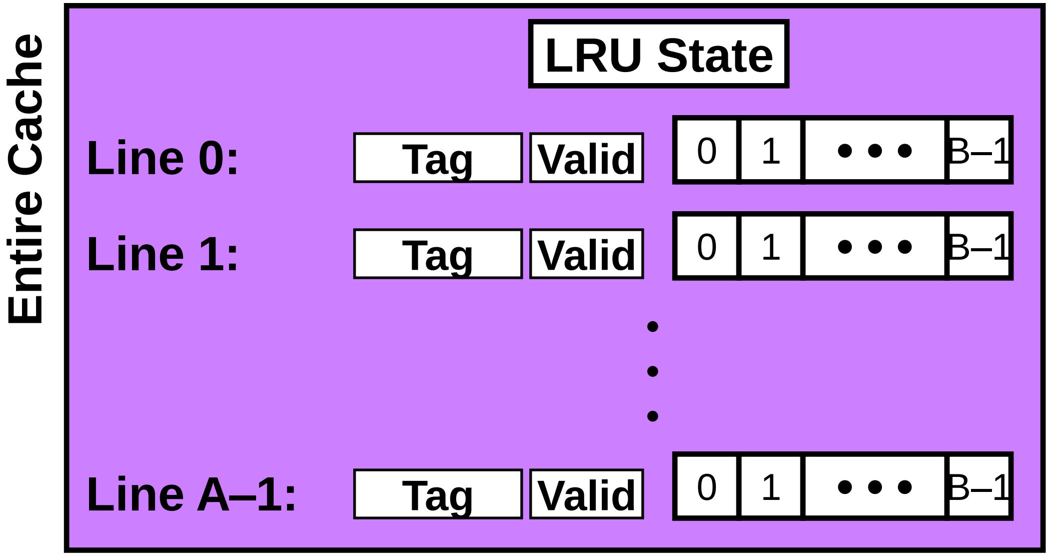
Parameters

- Block = $B = 2^b$ bytes
- Cache size $C = 2^c$
- Cache consists of single **set** holding $A = C/B = 2^{c-b}$ lines

Common in software caches

Only practical for small caches in hardware!

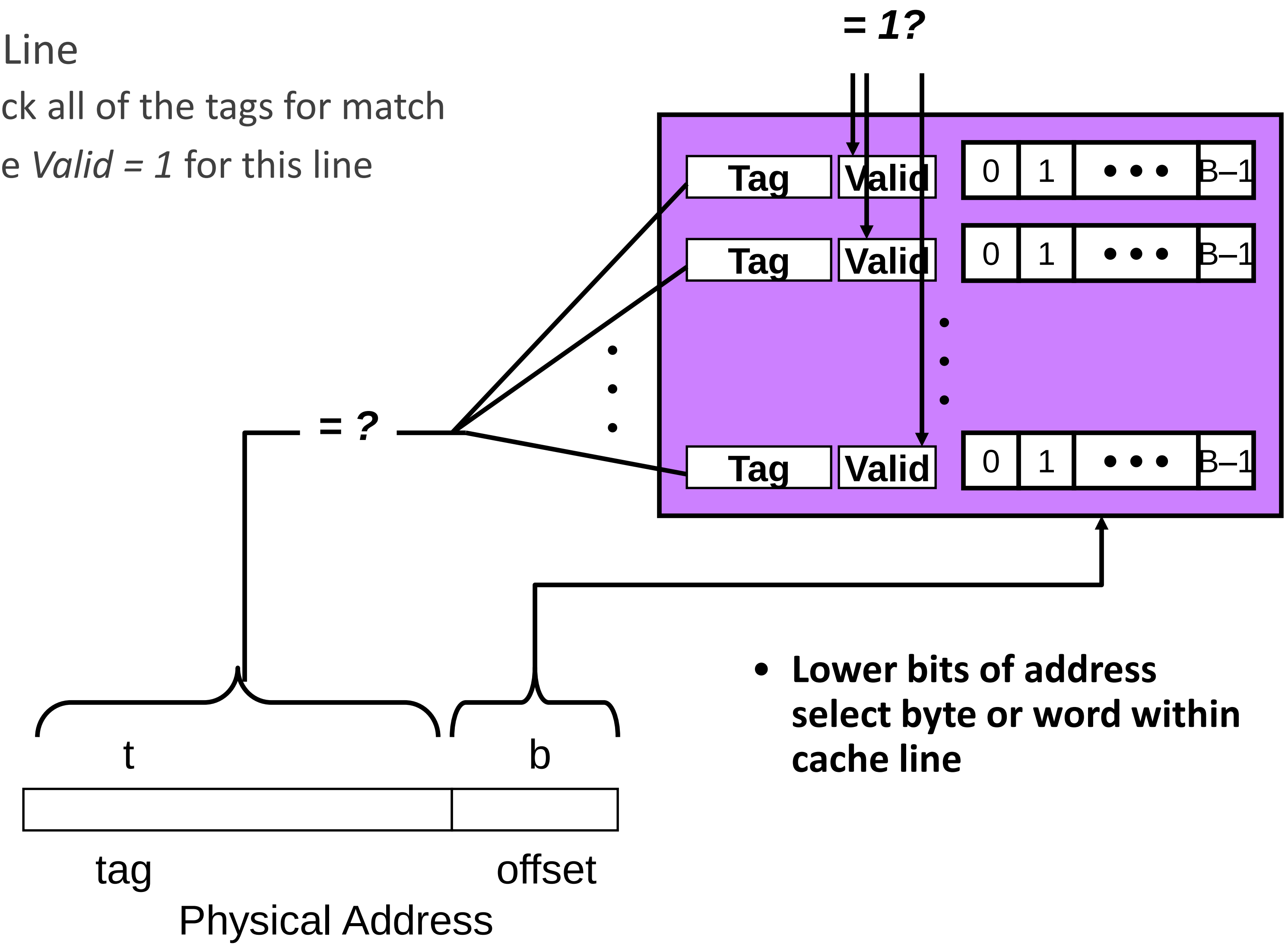
Still useful for analysis and simulation



Fully associative cache tag matching

Identifying Line

- Must check all of the tags for match
- Must have *Valid = 1* for this line



- **Lower bits of address select byte or word within cache line**

Direct-mapped caches

Simplest Design

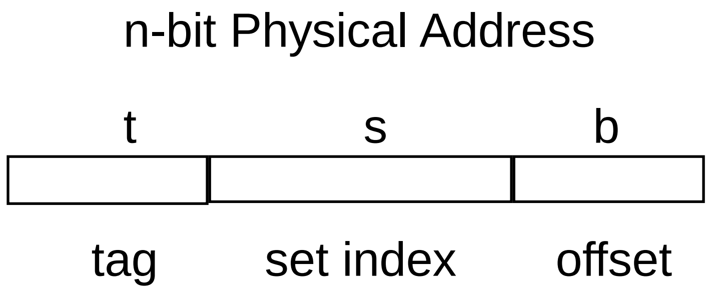
- Each address maps to a **single** cache line

Parameters

- Block = $B = 2^b$ bytes
- Cache size $C = 2^c$
- Cache consists of $S = 2^{c-b} = 2^s$ sets

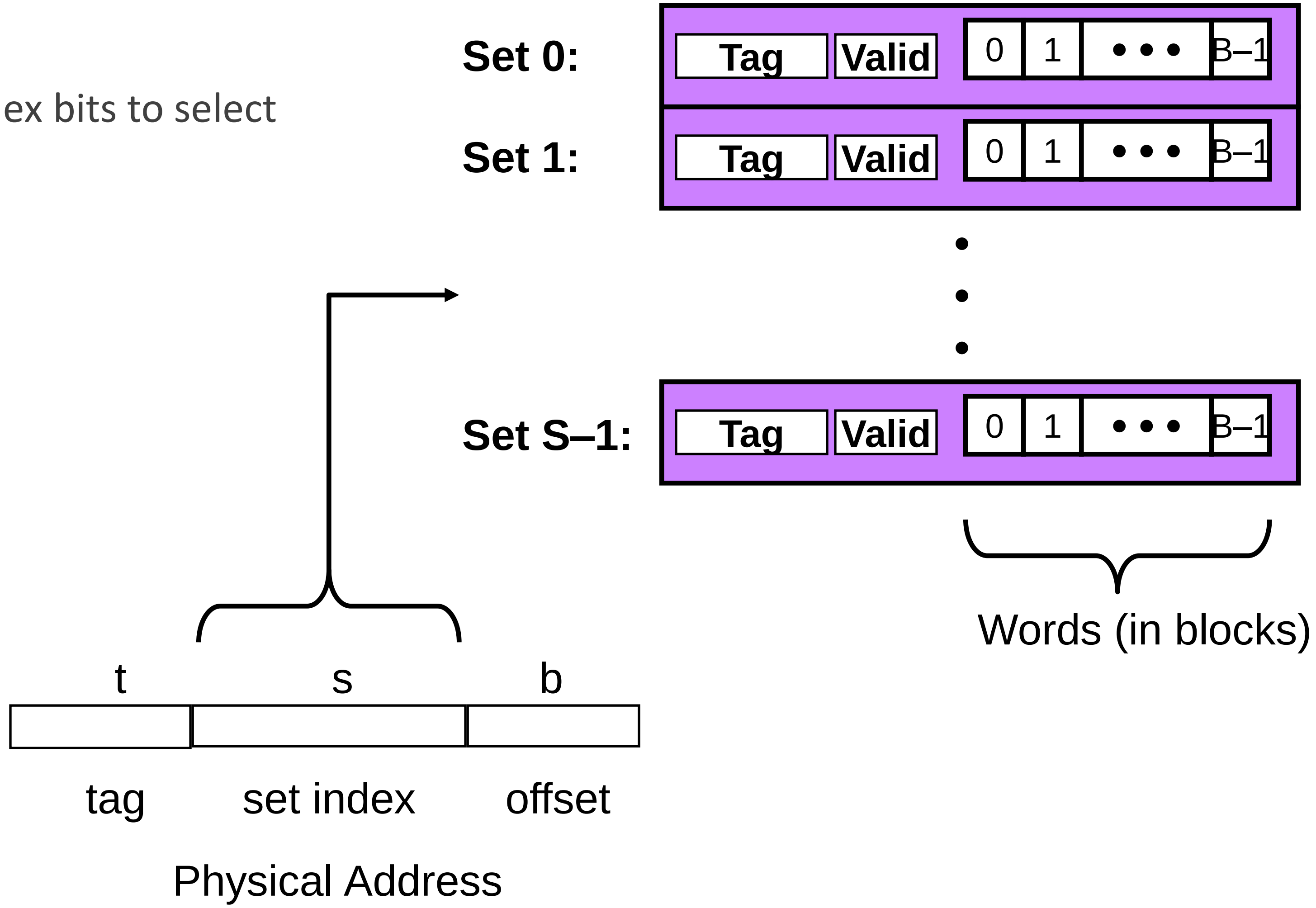
Physical Address

- *n* bits to reference N = 2^n total bytes
- Partition into fields
  - *Offset:* Lower *b* bits indicate which byte within line
  - *Set:* Next *s* bits indicate how to locate line within cache
  - *Tag:* Identifies this line when in cache



Indexing into a direct-mapped cache

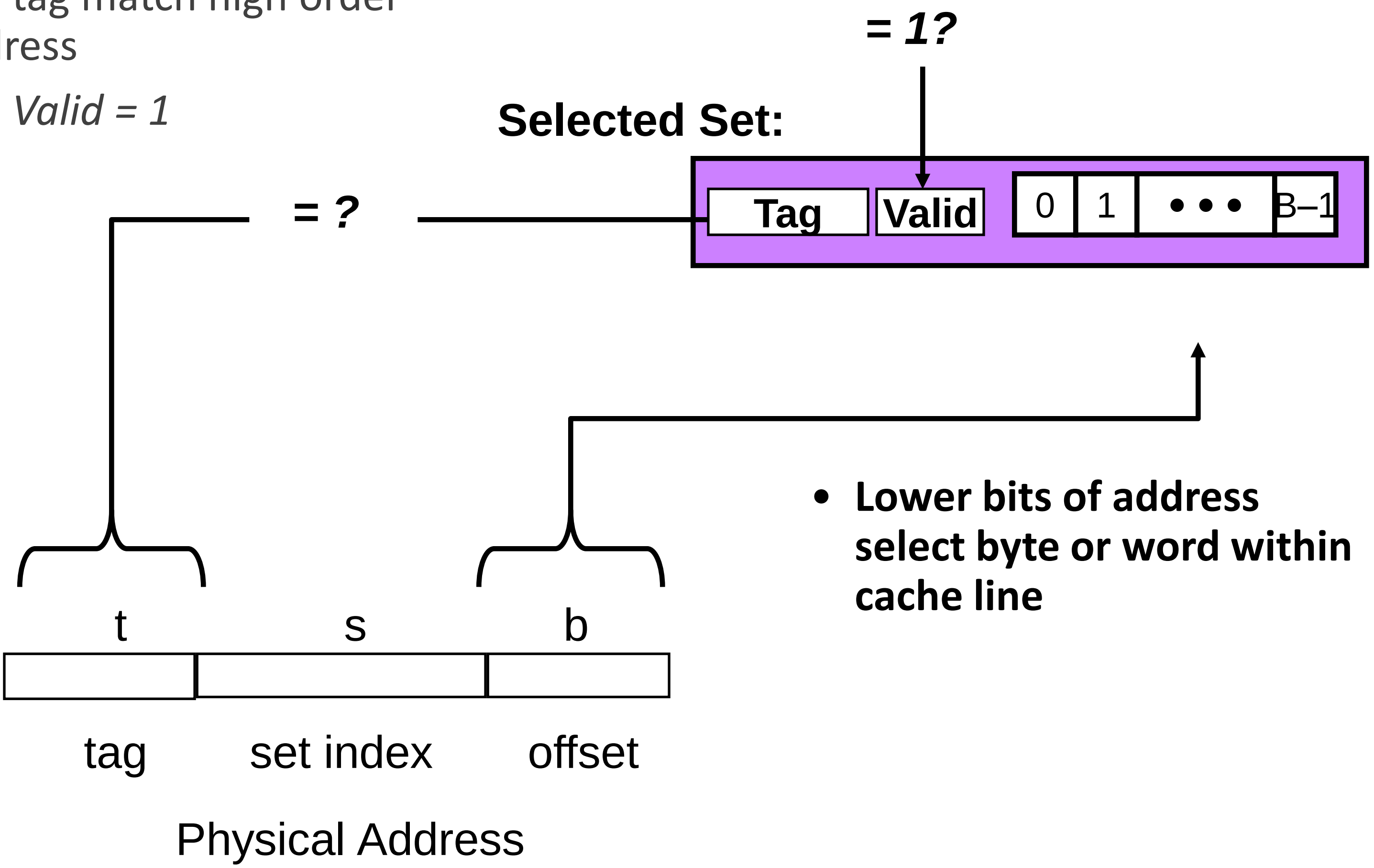
- Use set index bits to select cache set



Direct-mapped tag matching

Identifying Line

- Must have tag match high order bits of address
- Must have *Valid = 1*



- **Lower bits of address select byte or word within cache line**

Conflict example: Dot product

```
float dot_prod(float x[1024], y[1024])
{
  float sum = 0.0;
  int i;
  for (i = 0; i < 1024; i++)
    sum += x[i]*y[i];
  return sum;
}
```

Loop body

```
        …
Loop:   LD R1, 0(R3)
        LD R2, 0(R4)
        MUL R1, R1, R2
        ADD R5, R5, R1
        ADD R3, R3, 4
        ADD R4, R4, 4
        BNEQ R3, R6, Loop
        …
```

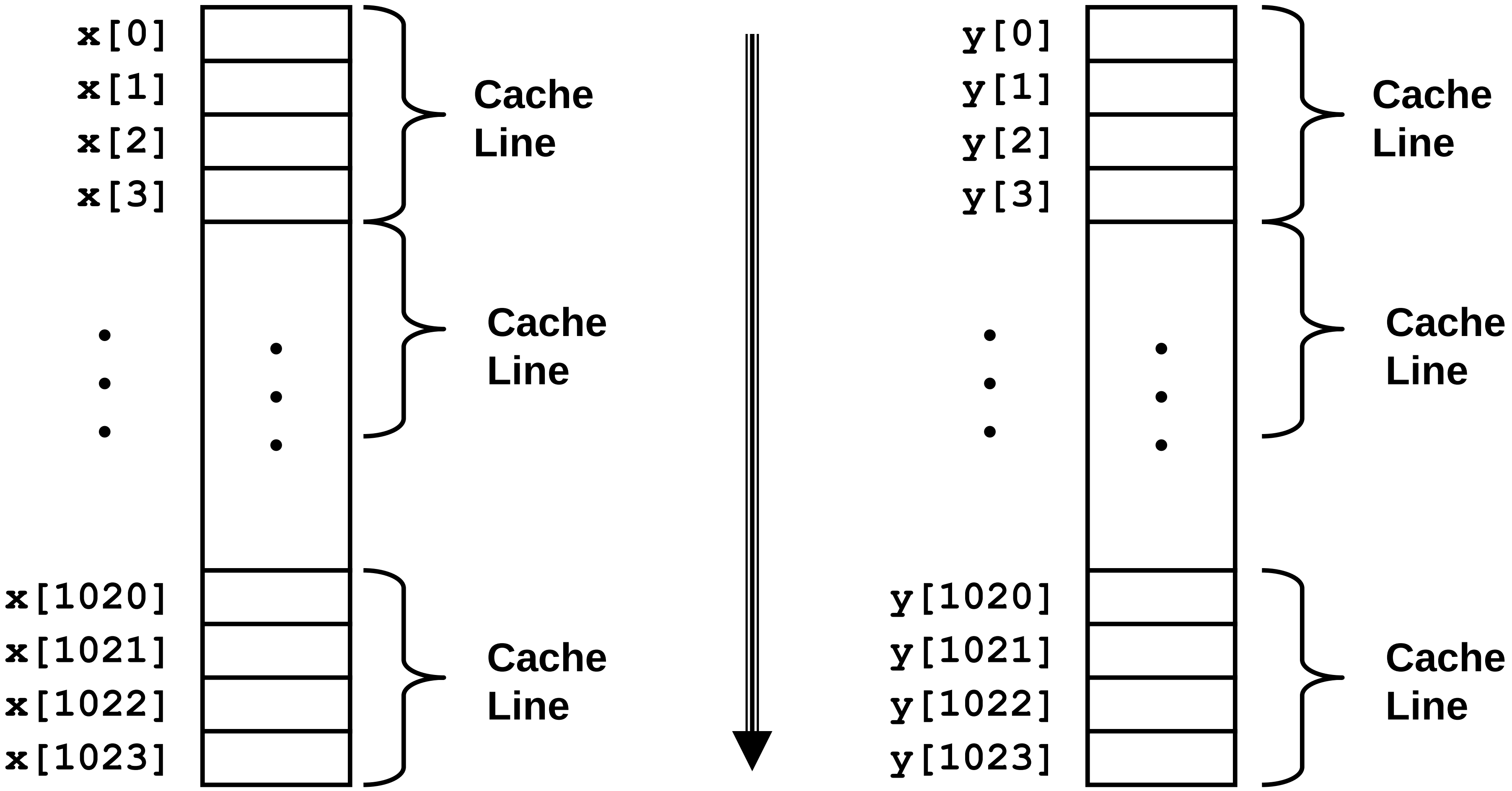
Single-cycle / instruction excluding LD time

64KB direct-mapped data cache, 16 B line size, 10 cycle miss penalty

Performance

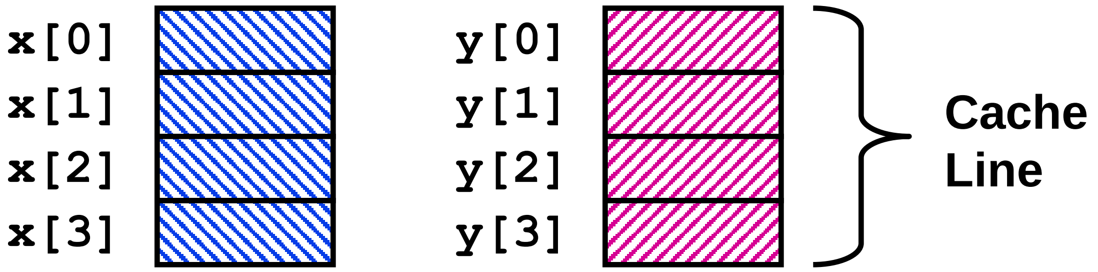
- Good case: 17 cycles / element
- Bad case: 47 cycles / element

Conflict example (cont'd)



- Access one element from each array per iteration

Conflict example (cont’d): Good case



Access Sequence

- Read x[0]
  - x[0], x[1], x[2], x[3] loaded
- Read y[0]
  - y[0], y[1], y[2], y[3] loaded
- Read x[1]
  - Hit
- Read y[1]
  - Hit
- • • •
- 2 misses / 8 reads

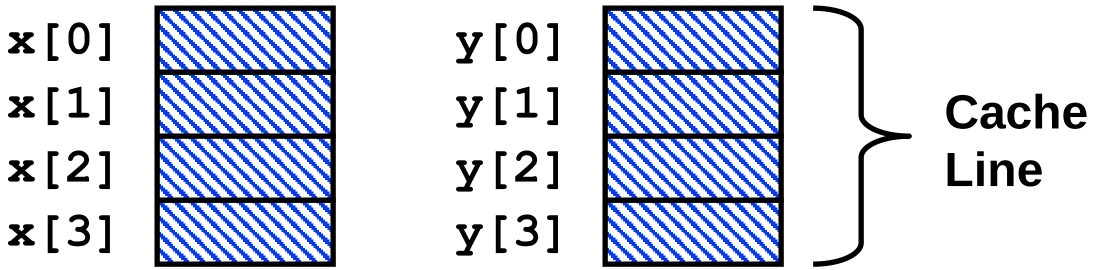
Analysis

- x[i] and y[i] map to different cache lines
- Miss rate = 25%
  - Two memory accesses / iteration
  - On every 4th iteration have two misses

Timing

- 7 cycle loop time
- 20 cycles / cache miss
- Average time / iteration = 7 + 0.25 * 2 * 20

Conflict example (cont’d): Bad case



Access Pattern

- Read x[0]
  - x[0], x[1], x[2], x[3] loaded
- Read y[0]
  - y[0], y[1], y[2], y[3] loaded
- Read x[1]
  - x[0], x[1], x[2], x[3] loaded
- Read y[1]
  - y[0], y[1], y[2], y[3] loaded

• • •

- 8 misses / 8 reads

Analysis

- x[i] and y[i] map to same cache lines
- Miss rate = 100%
  - Two memory accesses / iteration
  - On *every* iteration have two misses

Timing

- 7 cycle loop time
- 20 cycles / cache miss
- Average time / iteration =
  7 + 1.0 * 2 * 20

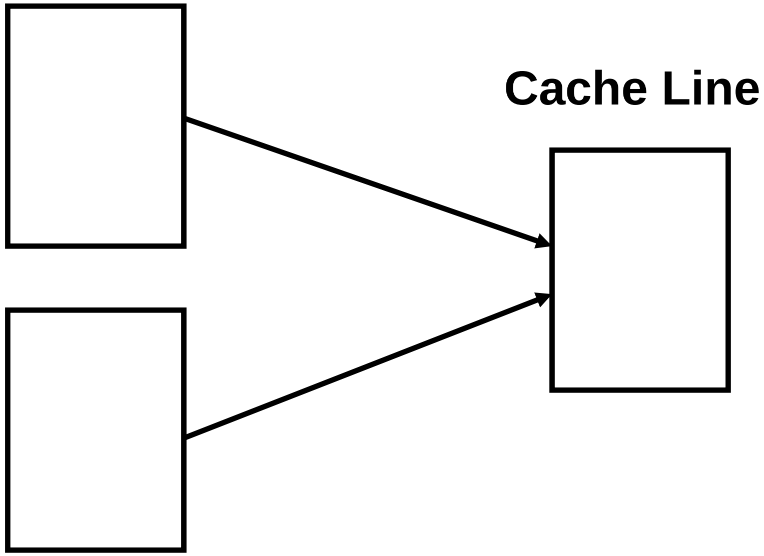
Tradeoffs of direct-mapped caches

Strength

- Minimal control hardware overhead
- Simple design
- Good hit time

Weakness

- Vulnerable to **conflicts** (i.e., thrashing)
- Two heavily used lines have same cache index
- Repeatedly evict one to make room for other



Impact of increasing block size

- Effect on cache area (tags + data)?
- Effect on hit time?
- Effect on miss rate?
- Effect on miss penalty?

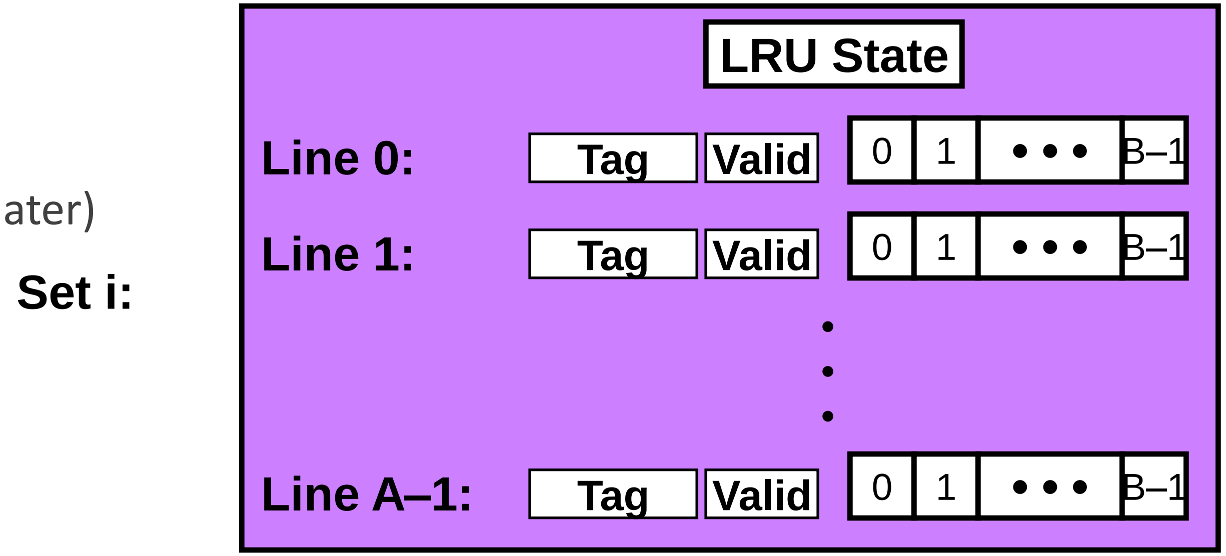
Set-associative cache

Mapping of Memory Lines

- Each address maps to one of A possible locations (a set)
  - $A = 2 - 8$ for L1s, $A = 8 - 32$ for other caches
  - Can map to any entry within its set

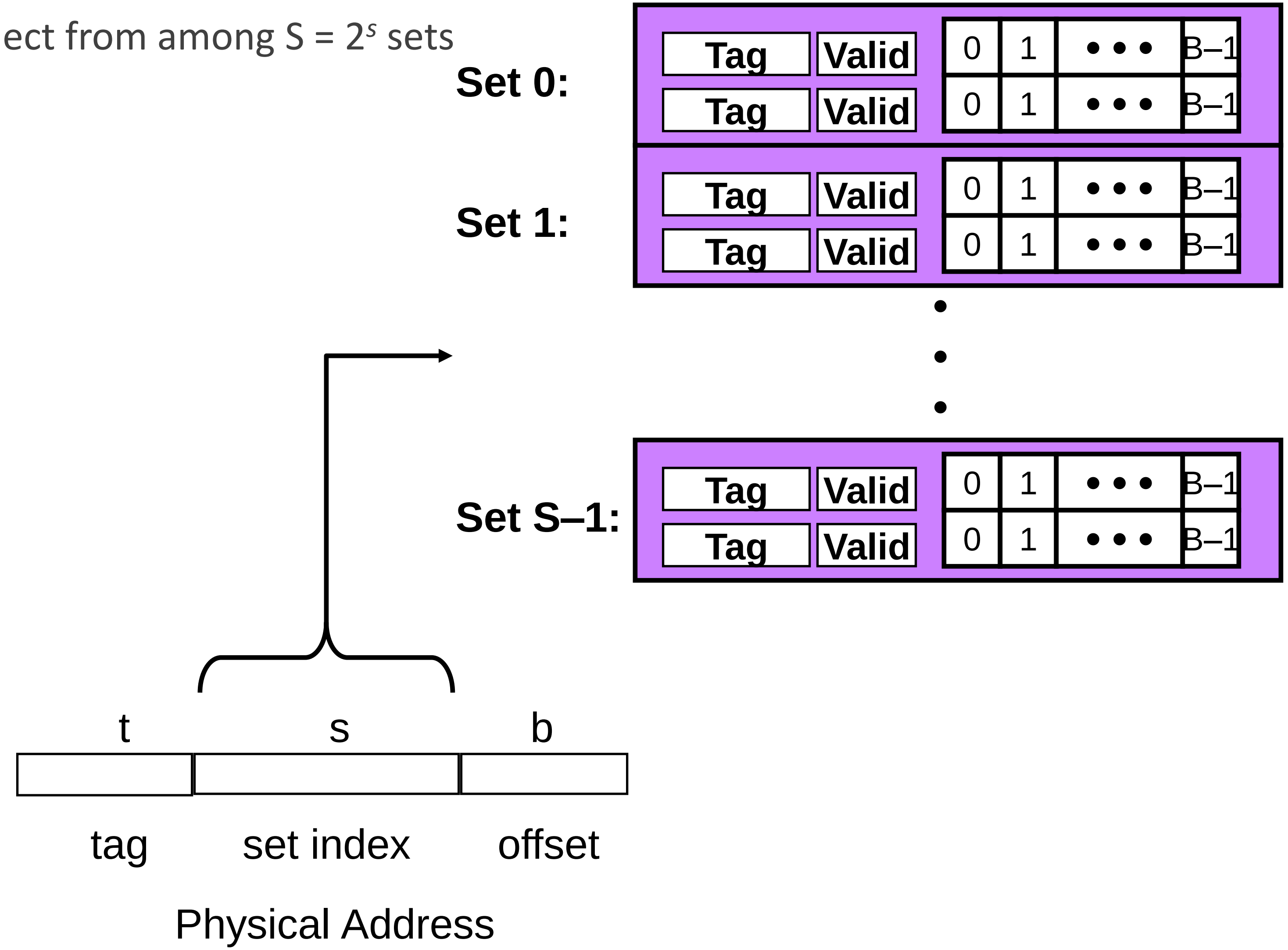
Tradeoffs

- Fewer conflict misses
- Longer access latency
- More complex to implement
- *Forced by virtual memory* (more on this later)



Indexing a 2-way set-associative cache

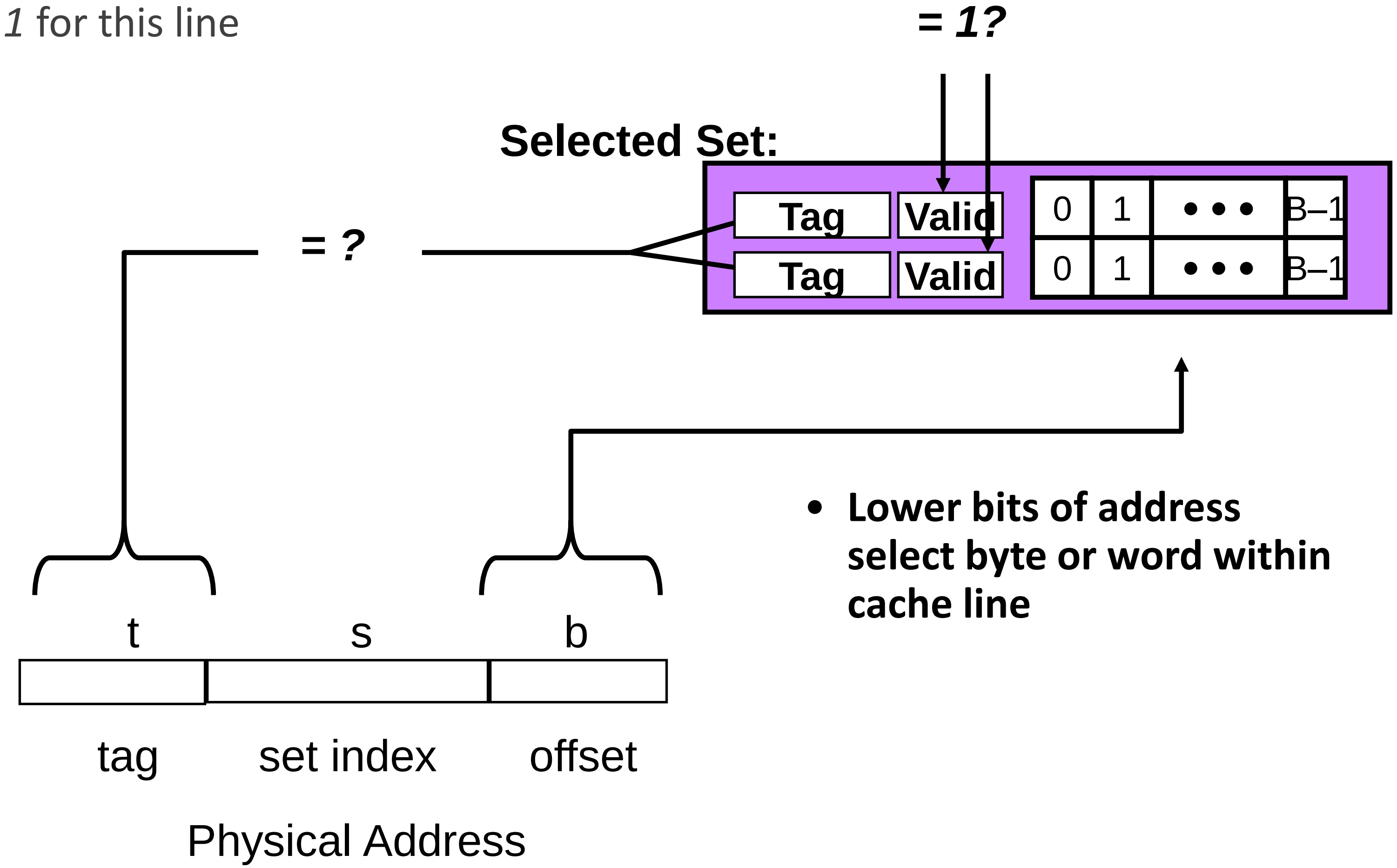
Use middle *s* bits to select from among S = 2^s sets



Set-associative tag matching

Identifying Line

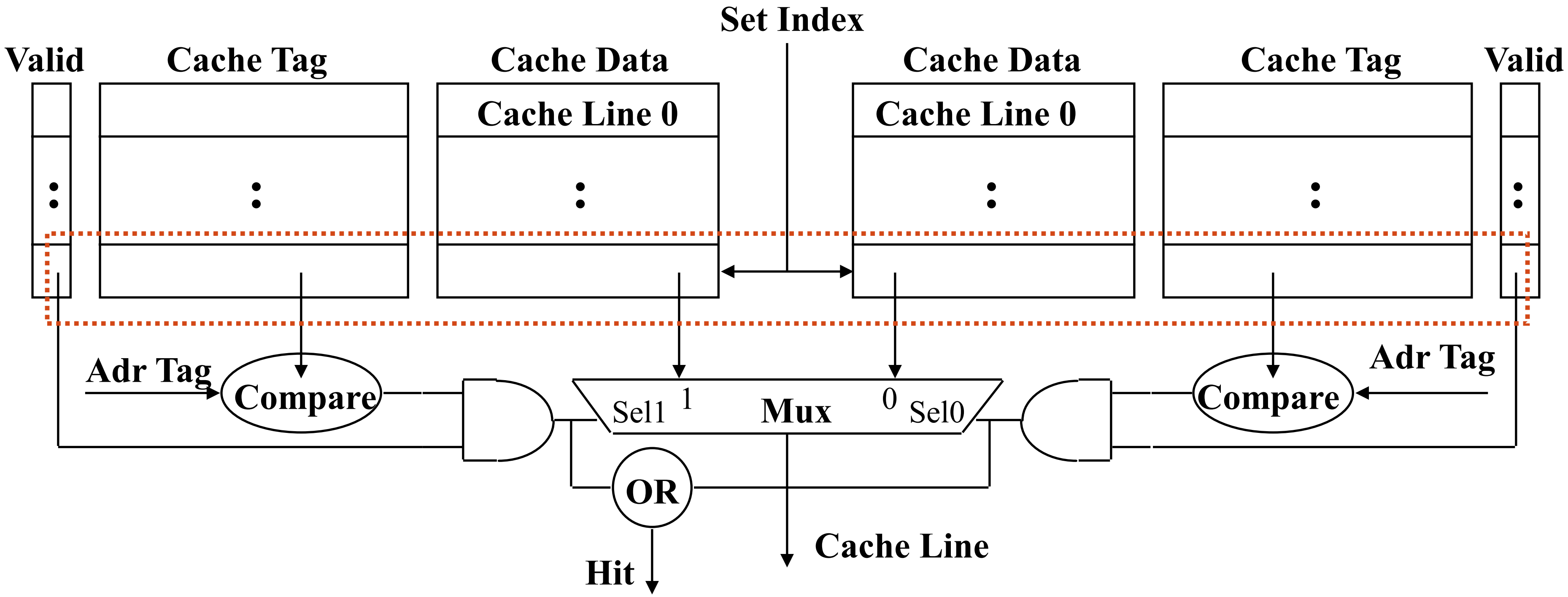
- Must have one of the tags match high order bits of address
- Must have *Valid = 1* for this line



- **Lower bits of address select byte or word within cache line**

Implementation of 2-way set-associative

- Set index selects a set from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result



Impact of increasing associativity

(eg, direct-mapped ➔ set associative ➔ fully associative)

Effect on cache area (tags+data)?

Hit time?

Miss rate?

Miss penalty?

Categorizing misses: The “3 Cs”

Compulsory/Cold-start Misses – address not seen previously; difficult to avoid (not impossible!)

- *Compulsory misses = misses @ infinite size*

Capacity Misses – cache not big enough; larger cache size

- *Capacity misses = fully associative misses – compulsory misses*

Conflict/Collision Misses – poor block placement evicts useful blocks

- *Conflict misses = actual misses – capacity misses*

Mark Hill, 2019 Eckhert-Mauchly Winner

What is associativity?

Simple answer: number of replacement candidates

More associativity ➔ better hit rates

- 1-way < 2-way < 3-way < … < fully associative

But what about…

Victim caches

Jouppi, ISCA’90

- Candidates include recently evicted blocks
- Does 1-way + 1-entry victim cache == 2-way?

- 1-way < 1-way + 1-entry victim cache < 2-way

Column-associative caches

Agarwal & Pudar, ISCA'93

- Direct-mapped cache, except…
- On a miss, check *secondary location* (flipping MSB of set index)
- On a hit, migrate to primary location

- 1-way < column-associative < 2-way ??????
  - Area?
  - Hit time?
  - Miss ratio?
  - Miss penalty?

Hashing

- Hash address to compute set index

- Reduces conflict misses
- But adds latency + tag size + complexity

- 1-way < 1-way hashed < 2-way ??????
- 8-way < 8-way hashed ??????

Hashing often used in last-level cache (LLC)

Skew-associative caches

Seznec, ISCA’93

- Use different hash function for each *way*
- Mixes candidates across sets for diff addresses

- ➔ No coherent concept of a “set” anymore

- 2-way < 2-way hash < 2-way skew < 3-way ?????

Zcaches

Sanchez and Kozyrakis, MICRO’10

Distinguish two concepts:

- Number of locations a line can reside (e.g., A for set-associative)
- Number of possible victims upon a cache miss

Zcaches use *cuckoo hashing* to greatly expand # of possible victims

- Skew-associative baseline organization
- Compute possible locations for incoming address
- Re-hash addresses in those locations (they could move to any of these locations)
- Re-hash the addresses in *their* other locations
- …Repeat as desired

Zcaches get *miss ratio of highly-associative cache* with *hit time of low-associative cache*

- E.g., 4-ways & 52 replacement candidates

Associativity through the lens of probability

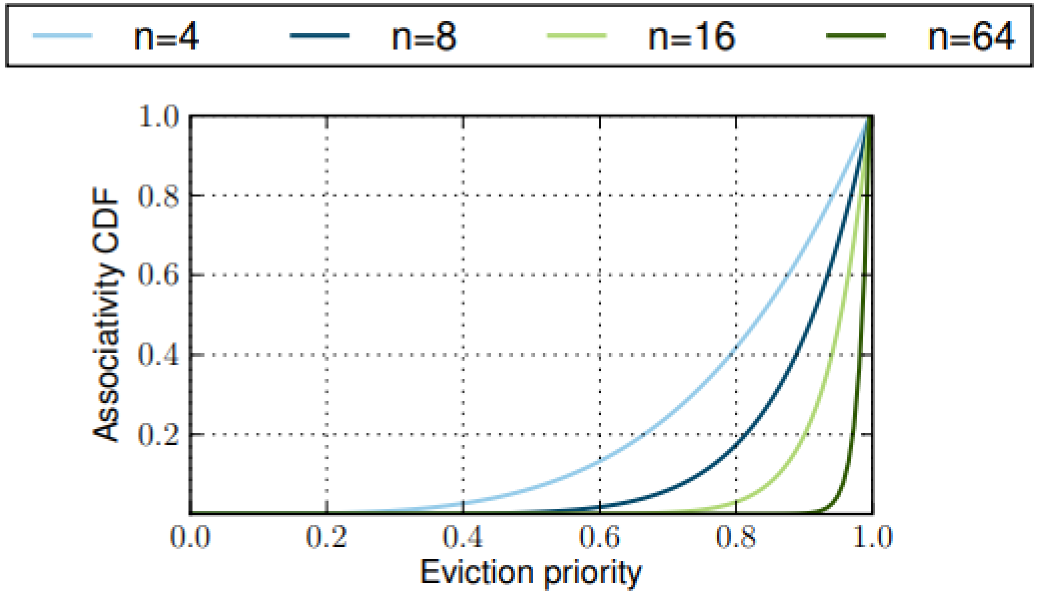
We need a better way to think about associativity!

Associativity can be thought as a distribution of victims’ eviction priority [Sanchez+, MICRO‘10]

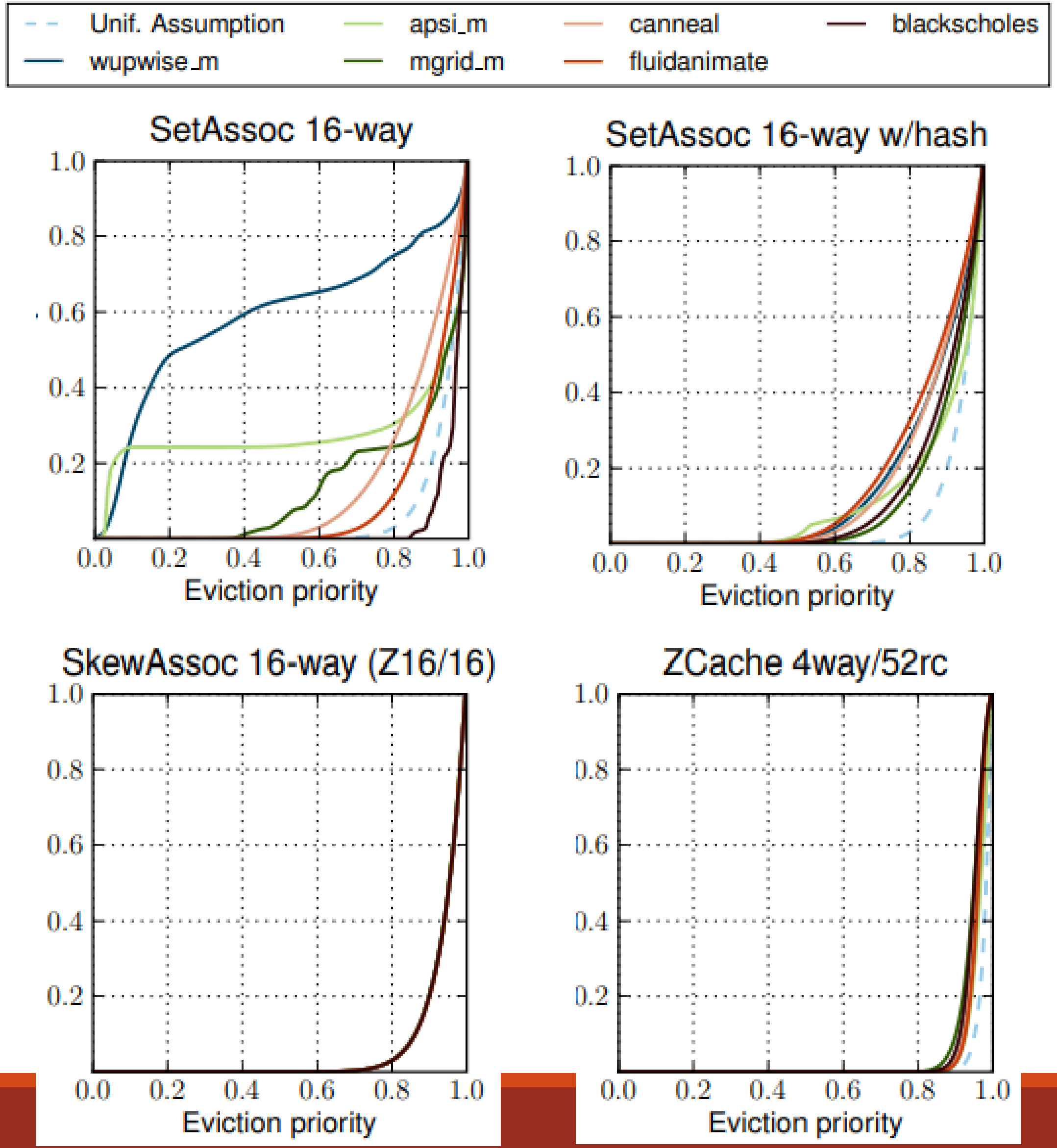
- Distribution answers two questions: *Among all cached blocks, how much did I want to evict the victim? (y-axis) How likely was that? (x-axis)*
- Fully associative always evicts the highest rank
- Random sampling converges toward fully associative with larger samplers
- Can plot associativity distribution (eg, through simulation) for different cache organizations

Some associativity distributions

[Sanchez+, MICRO’10]



Idealized (fully randomized) set-associative model
(analytical model)



Real applications
on different cache
organizations
(in simulation)

Replacement/eviction algorithms

If there's not enough space in the cache, what should we kick out?

Optimal algorithm (Belady/MIN/OPT)

Usage based algorithms

Non-usage based algorithms

Replacement/eviction algorithms

If there’s not enough space in the cache, what should we kick out?

Optimal algorithm (Belady/MIN/OPT)

- Replace the block that is next referenced furthest in the future
- **<u>Must know the future</u>** (can’t be implemented)
- Tricky to prove optimality; only optimal under “vanilla” cache designs

Usage based algorithms

Non-usage based algorithms

Replacement/eviction algorithms

If there’s not enough space in the cache, what should we kick out?

Optimal algorithm (Belady/MIN/OPT)

Usage based algorithms

- **Least-recently used (LRU)**
  - Replace the block that has been referenced least recently (longest ago)
  - Seen as hard to implement (but isn’t, really)
- Least-frequently used (LFU)
  - Replace the block that has been referenced the fewest times
  - Even harder to implement (“true” LFU—track blocks not in cache?)
- Many approximations: CLOCK, tree-based pseudo-LRU, etc

Non-usage based algorithms

Replacement/eviction algorithms

If there's not enough space in the cache, what should we kick out?

Optimal algorithm (Belady/MIN/OPT)

Usage based algorithms

Non-usage based algorithms

- First-in First-out (FIFO)
  - Weird pathologies (eg, hit rate degrades at larger cache size)
- Random (RAND)
  - Bad hit ratio, but sometimes necessary (eg, when updating tags is expensive)

Dynamic Insertion Policies (DIP)

[Qureshi+, ISCA'07]

LRU is usually good, but sometimes really bad

- Especially at last-level cache (LLC) – why?

LLC can't afford to perform pathologically ➔ worth investing in smarter policy

BIP (bimodal insertion policy) fixes LRU on these apps

- Insert ("admit") new lines at LRU position most of the time (31/32)

DIP (dynamic insertion policy) chooses between LRU and BIP

- Introduces **set sampling** – a clever & highly influential technique

[Jaleel+, ISCA'10]

Re-reference interval prediction (RRIP)

Builds on ideas from DIP, with new mechanism

Motivation: predict whether a line will be used near, middle, or far from now

- Reality: assigns high/medium/low value ("prediction" doesn't mean much)

Mechanism: multi-bit CLOCK

- Each tag tracks N bits for replacement (fewer bits is better)
- Evict a line at max priority = $2^N - 1$
- If none have max priority, increment all by 1 until something does + evict it

Policies: SRRIP

- *Insert* lines at priority 2^{N-1}
- *Promote* lines (on hit) to 0 priority

BRRIP

Insert most lines at priority $2^N - 1$ (like BIP)

Promote lines (on hit) to 0 priority

DRRIP chooses between SRRIP / BRRIP just like DIP

- Rumored to be implemented in Intel processors

[Qureshi+, MICRO’06]

Utility-based Cache Partitioning

Multicores have shared last-level caches (much more on this later)

➔ Misbehaving applications harm performance for everyone

Idea: Partition cache space among applications to maximize LLC benefit

- Partitioning has many other uses in security, fairness, etc…

Mechanisms:

- Way-partitioning of set-associative cache (since implemented as “Intel CAT”)
- Utility monitors (UMONs) gather apps’ *miss curves*
- Simple, local-search algorithm allocates ways to apps that get highest *marginal benefit*

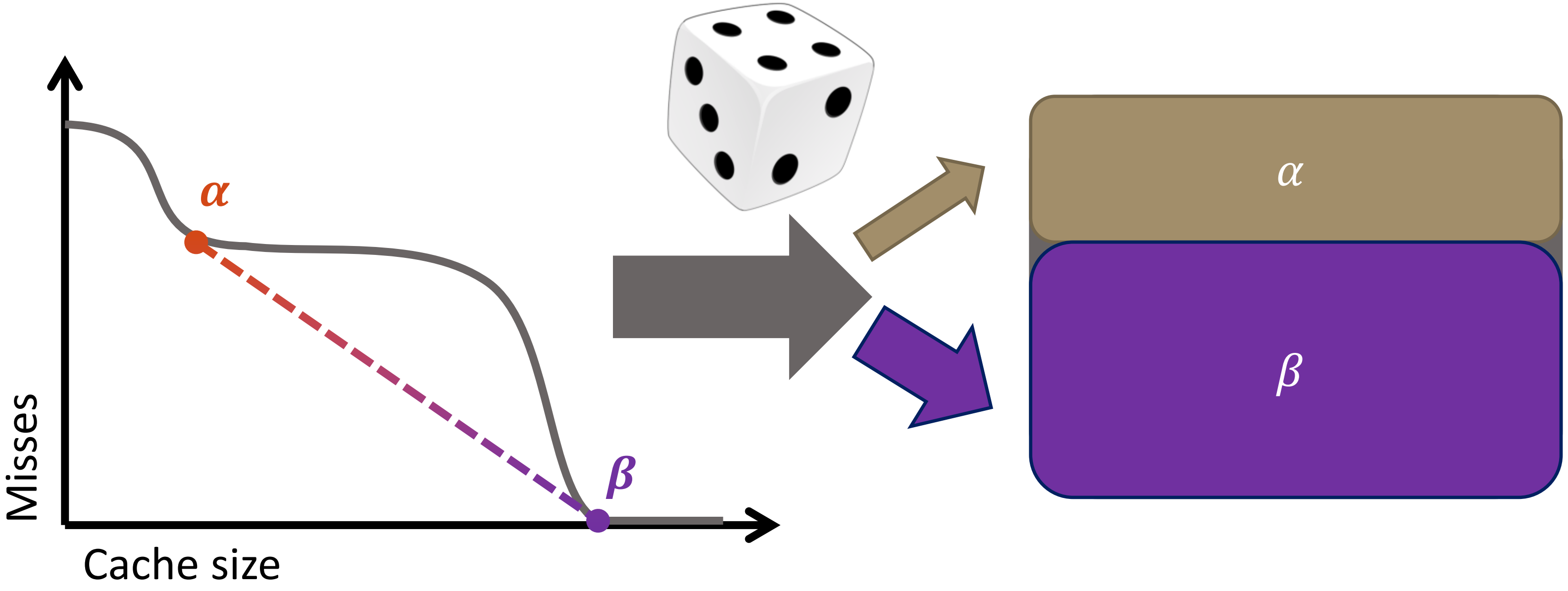
[Beckmann+, HPCA'15]

Talus: Avoiding Cache Performance Cliffs

A partitioning-based approach to cache replacement

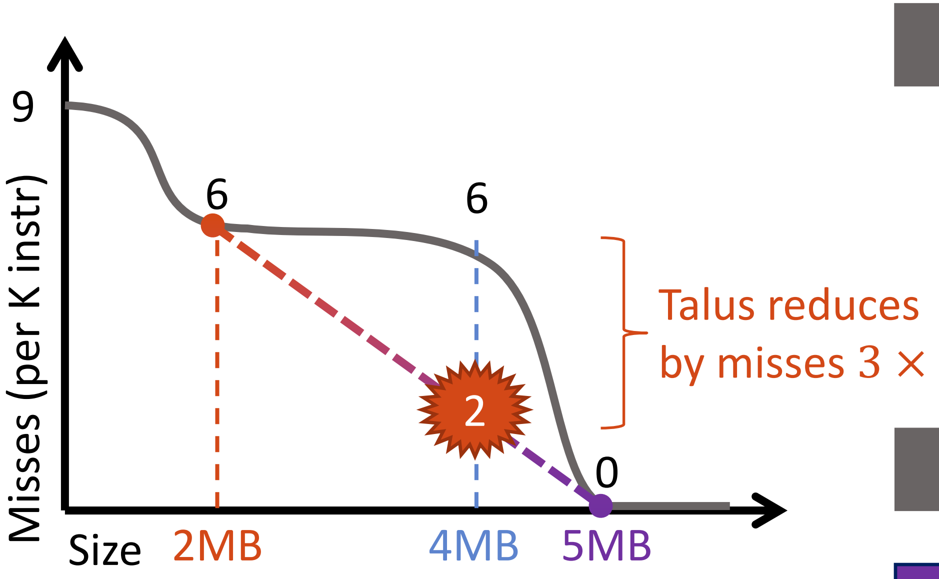
- Partition accesses from a *single* application, not different ones
- Choose partition at random

Provably achieves *convex hull* of LRU's miss curve

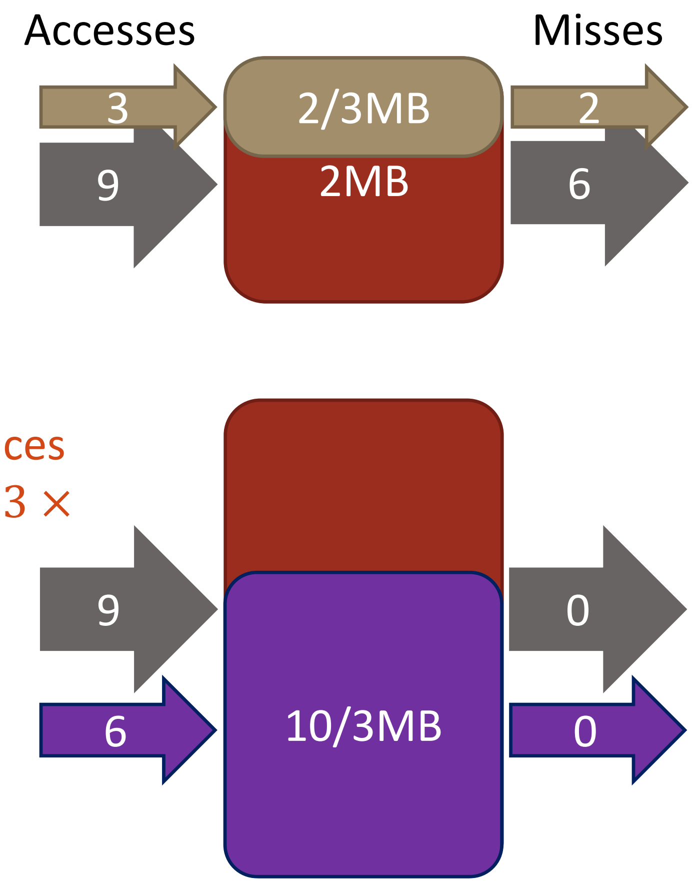


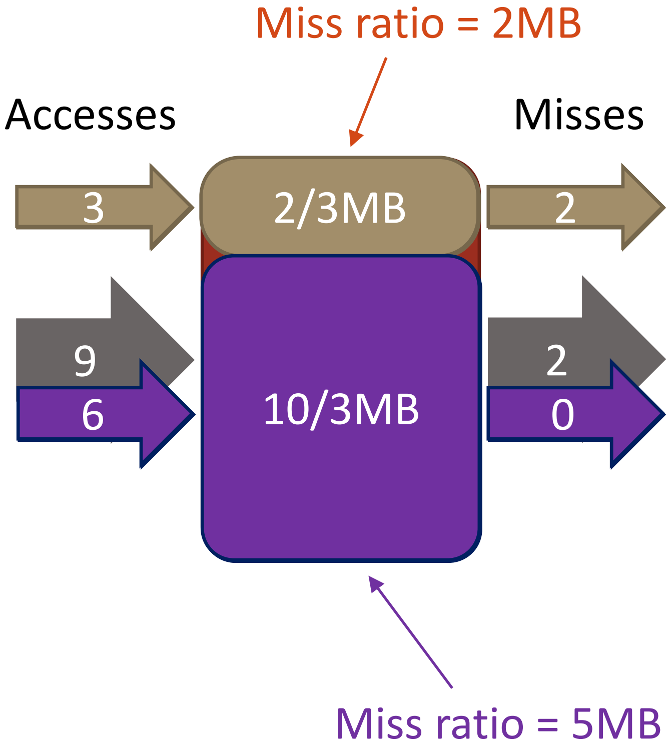
[Beckmann+, HPCA’15]

Talus: Avoiding Cache Performance Cliffs



Talus reduces by misses 3 ×



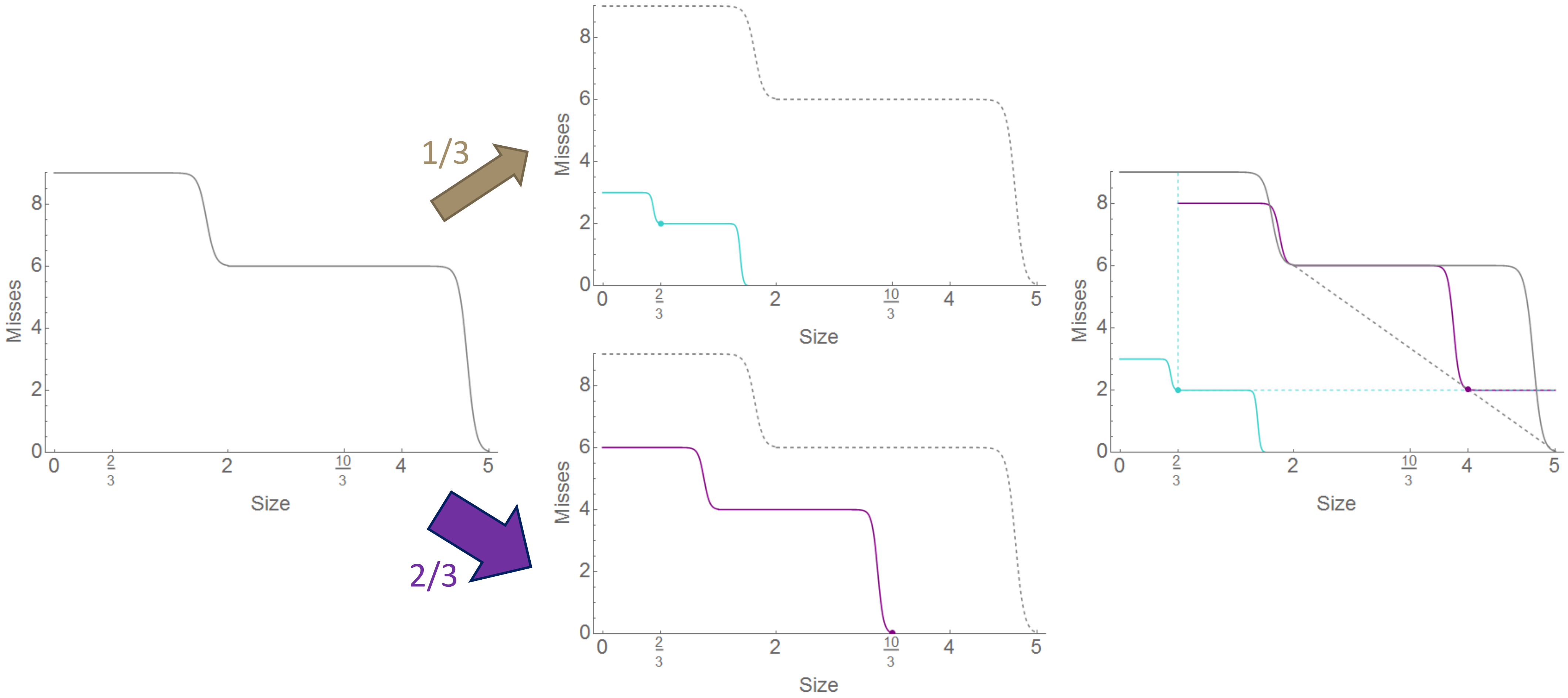


Miss ratio = 2MB

Miss ratio = 5MB

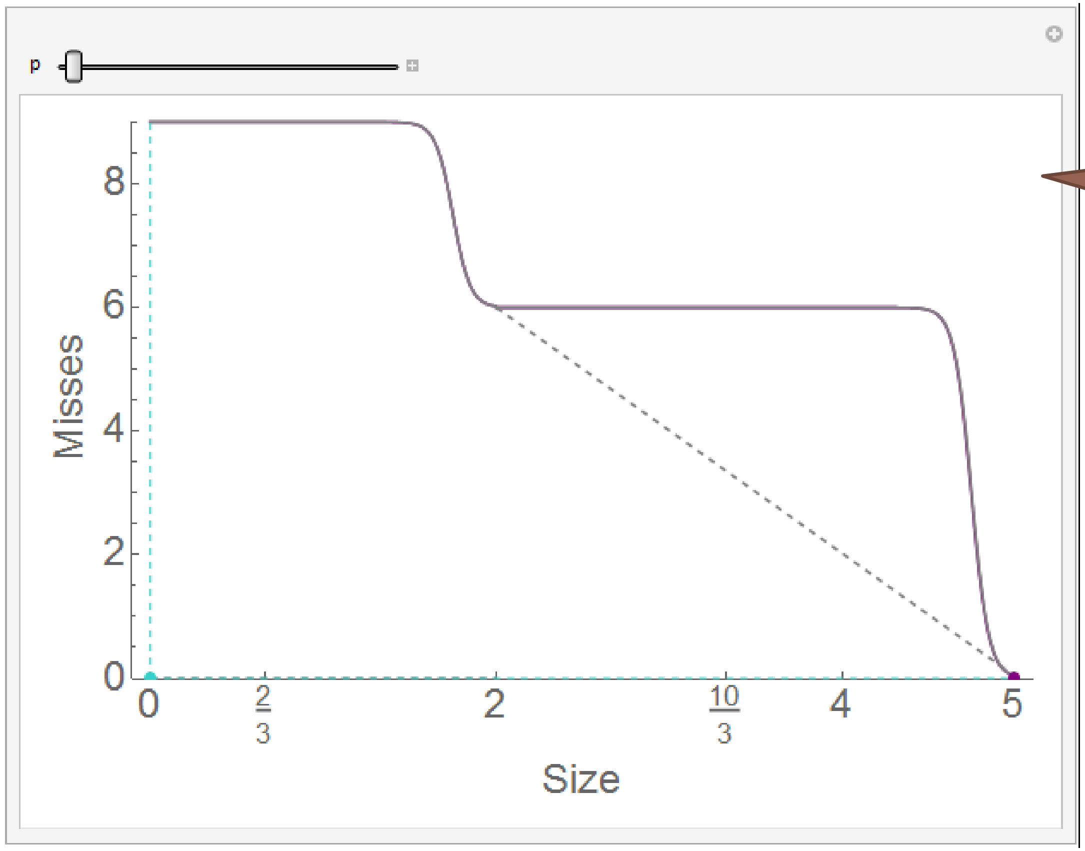
[Beckmann+, HPCA'15]

Talus: Avoiding Cache Performance Cliffs



[Beckmann+, HPCA’15]

Talus: Avoiding Cache Performance Cliffs



Generalize + prove convexity!

Simple hardware

Categorizing misses: The 3 C’s++

Compulsory misses - unchanged

Capacity Misses – cache not big enough

- Capacity misses = fully associative misses <u>with optimal replacement </u>– compulsory misses

Replacement misses: those due to sub-optimal replacement decisions

- Replacement misses = fully associative misses – capacity misses

Conflict/Collision Misses – poor block placement

- Conflict misses = actual misses – <u>replacement</u> misses

Impact of Replacement Policy

Improving replacement policy
(eg, random ➔ LRU)

Effect on cache area (tags+data)?

Hit time?

Miss rate?

Miss penalty?

Prefetching

Applications only care about memory latency on the *critical path*

- Only load latency matters
- “Load criticality” is its own whole research area…

Prefetchers can hide load misses by loading data into the cache before apps request it

- Prefetching is an example of *speculation*

Classic paper: [Jouppi, ISCA’90] proposed victim caches + stride prefetcher

Next-Line Prefetching

Simplest scheme: when A is accessed, fetch A + 1

- Or up to N lines ahead of A

Why is this useful?

- Sequential accesses to data
- Instruction locality
- (Why not just increase block size?)

+ Simple to implement

+ Surprisingly good coverage…

– But coverage still pretty poor

Stride Prefetching

When A is accessed, fetch A + N * d

Why is this useful?

- E.g., column-major accesses in a matrix

Details:

- How to detect a stream? Usually a table indexed by load PC, but doesn’t have to be
- On modern systems, must fetch many accesses ahead of program to keep up

+ Still pretty simple to implement

– Only covers streaming accesses

Irregular Prefetchers

Other access patterns fall into broad category of “irregular prefetchers”

Example: Indirect Memory Prefetcher (IMP) [Yu+, MICRO’15]

- Detects patterns like: A[i], A[i+1], A[i+2] ➔ prefetches A[i+N]

Why is this useful?

- Sparse linear algebra, neural networks, graph analytics

Details:

- How can you detect A[i]…? Complex interactions with virtual memory

+ Big coverage gains for some apps

+ Fairly specialized

– Complex to implement

Prefetching metrics

Accuracy: used prefetches / prefetches issued

Coverage: prefetched misses / total misses

Timeliness: on-time prefetches / used prefetches

- More generally, *fraction of miss latency covered by prefetches*

Prefetching summary

+ Hide memory latency ➔ better performance

– Extra data movement on mispredictions

When to enable prefetchers?

- **Good:** When applications are latency-limited
- **Bad:** When applications are bandwidth- or power-limited

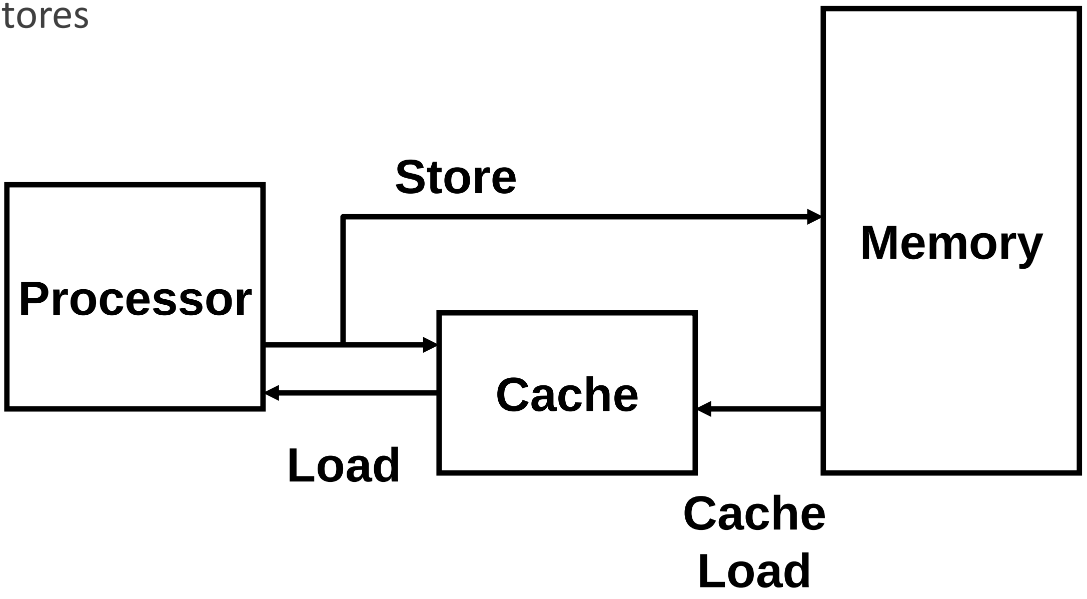
Write policy

What happens when processor writes to the cache?

Should memory be updated as well?

Write Through:

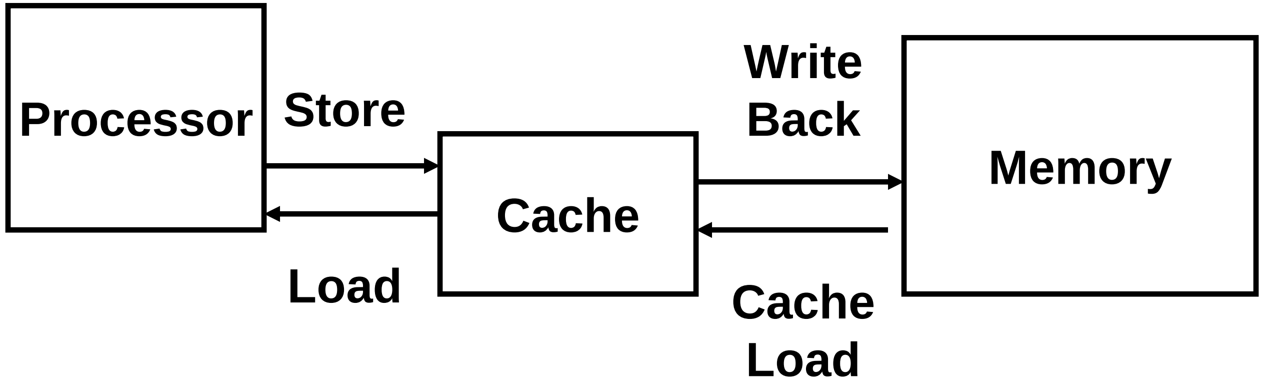
- Store by processor updates cache *and* memory
- Memory always consistent with cache
- Never need to store from cache to memory
- ~2X more loads than stores



Write policy (cont'd)

Write Back:

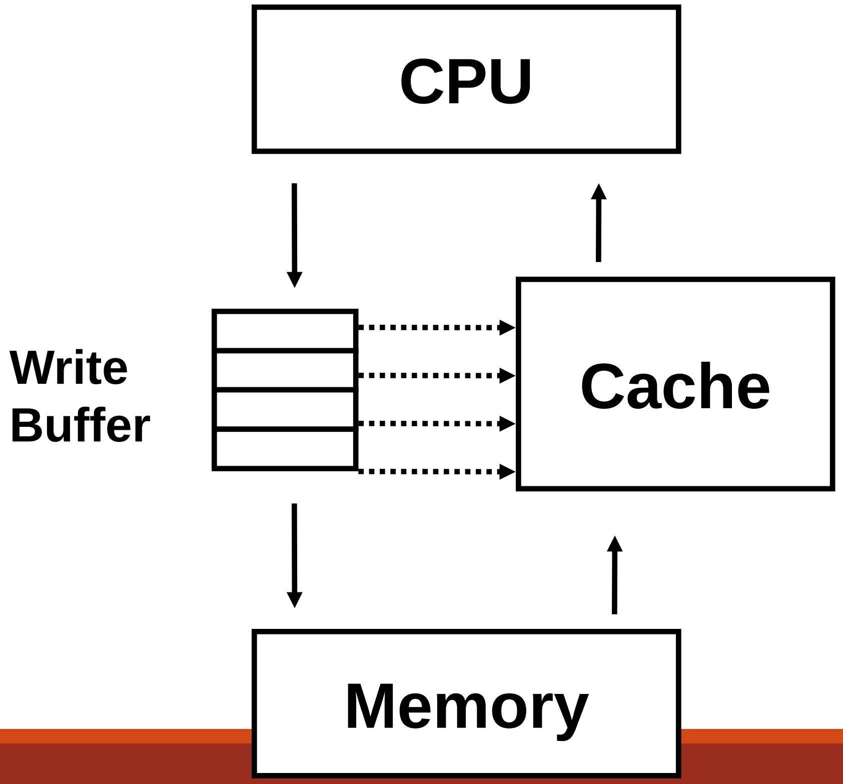
- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
  - Requires "dirty bit" for each line
    - Set when line in cache is modified
    - Indicates that line in memory is stale
- Memory not always consistent with cache



Write buffering

Write Buffer

- Common optimization for all caches
- Overlaps memory updates with processor execution
- Read operation must check write buffer for matching address

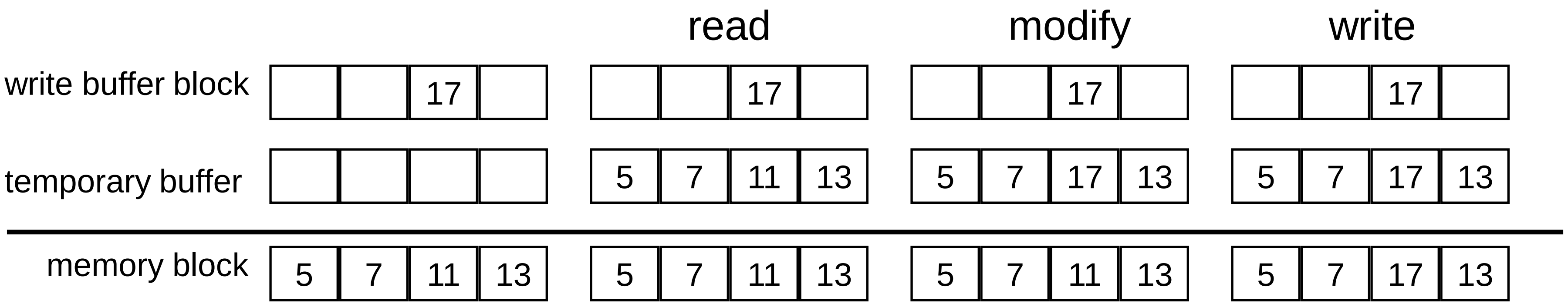


Allocation strategies

On a write miss, is the block loaded from memory into the cache?

Write Allocate:

- Block is loaded into cache on a write miss.
- Write requires read-modify-write to replace word within block



					read				modify				write			
write buffer block			17				17				17				17	
temporary buffer					5	7	11	13	5	7	17	13	5	7	17	13
memory block	5	7	11	13	5	7	11	13	5	7	11	13	5	7	17	13

- But if you've gone to the trouble of reading the entire block, why not load it in cache?
- ➔ Temporal locality! (Data is often read soon after being written)

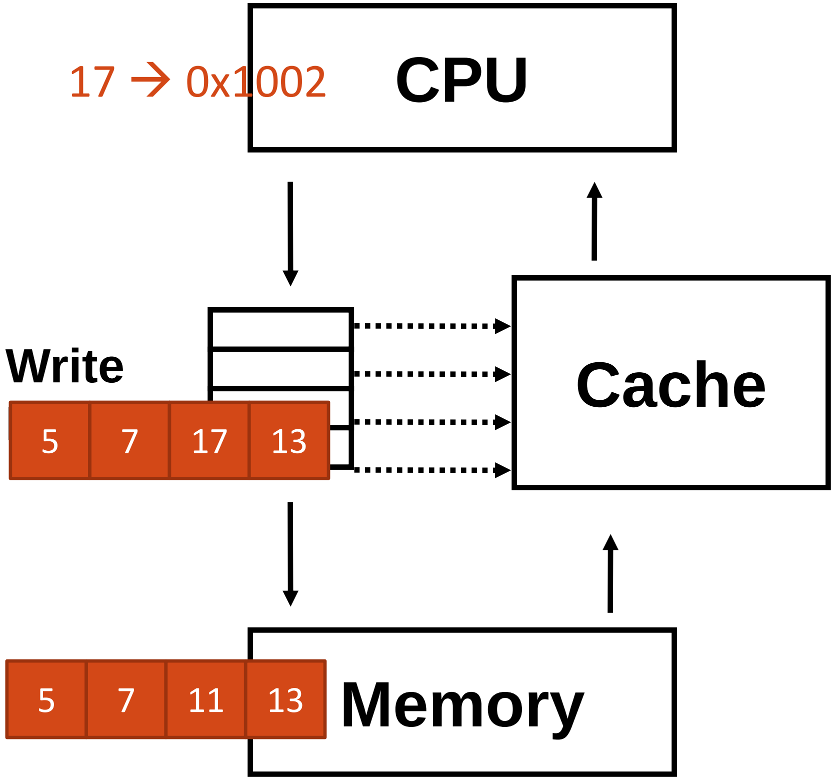
Allocation strategies

On a write miss, is the block loaded from memory into the cache?

But if you've gone to the trouble of reading the entire block, why not load it in cache?

Write Allocate:

- Block is loaded into cache on a write miss.
- Write requires read-modify-write to replace word within block
- Also, temporal locality! (Data is often read soon after being written)



Allocation strategies (cont'd)

On a write miss, is the block loaded from memory into the cache?

No-Write Allocate (Write Around):

- Block is not loaded into cache on a write miss
- Memory system directly handles word-level writes

Common strategies:

1. Writeback + Write-allocate
   - You already allocated space for the line, might as well keep it until evicted
2. Write-through + No-write-allocate
   - You are writing through to memory anyway (typically because you don't expect any data reuse), so you might as well save cache space

Impact of write policy

Writeback vs write-through

Effect on cache area (tags+data)?

Hit time?

Miss rate?

Miss penalty?

Summary: Memory hierarchy

Gap between memory + compute is growing

Processors often spend most of their time + energy waiting for memory, not doing useful work

Hierarchy and *locality* are the key ideas to scale memory performance

Most systems use caches, which introduce many parameters to the design with many tradeoffs

- E.g., associativity—hit rate vs hit latency

Self-check questions

Advanced replacement policies often assume that the cache is *not* inclusive. Why does this matter?

Complex cache organizations (e.g., skew-associative) are almost always proposed for the last-level cache. Why are these worthwhile at the LLC? Why does the same argument not apply to the L1?