Memory Hierarchy

15-740 SPRING'18

NATHAN BECKMANN

Topics

Memories

Caches

L3 Reading

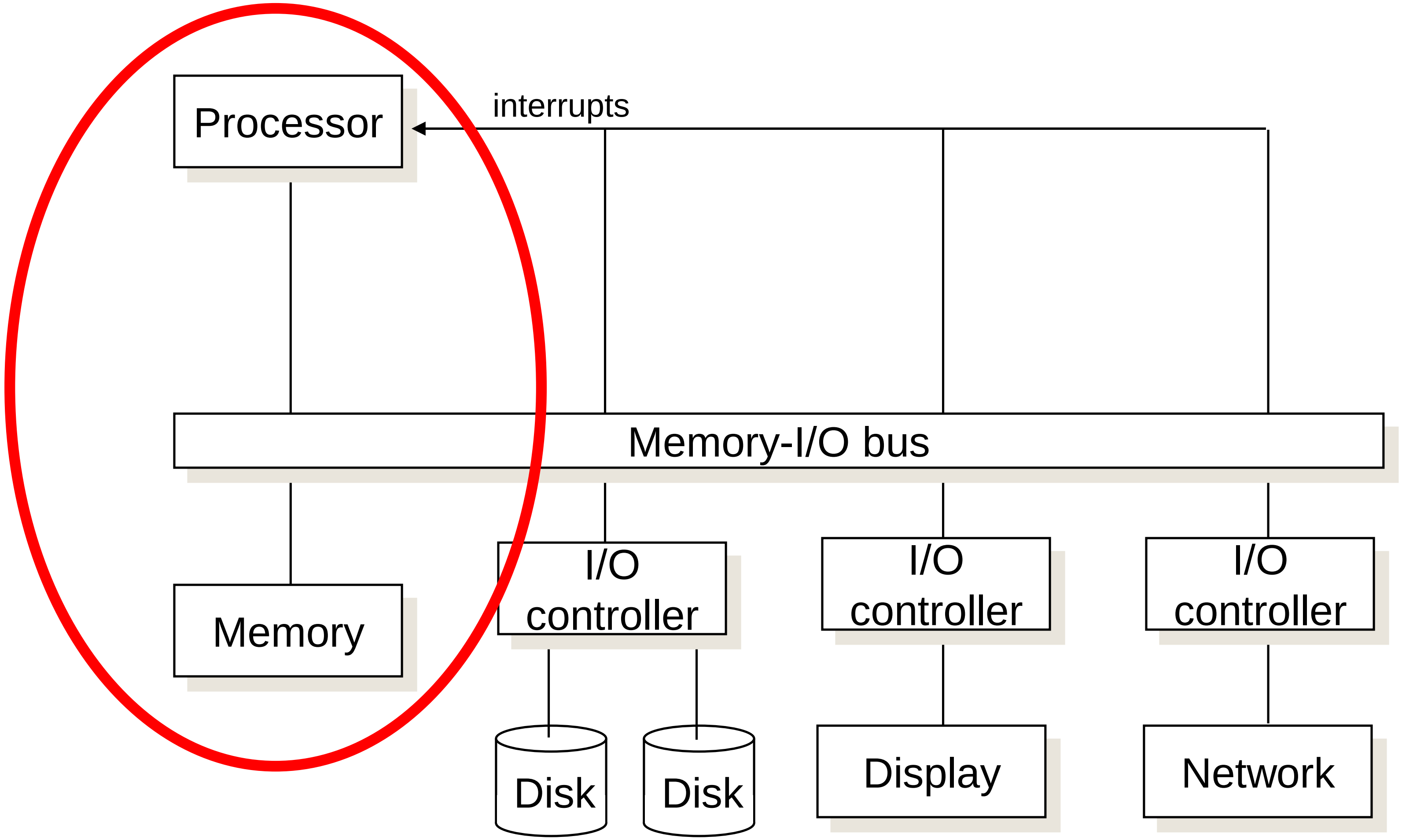
Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers

Famous paper; won “Test of Time Award”

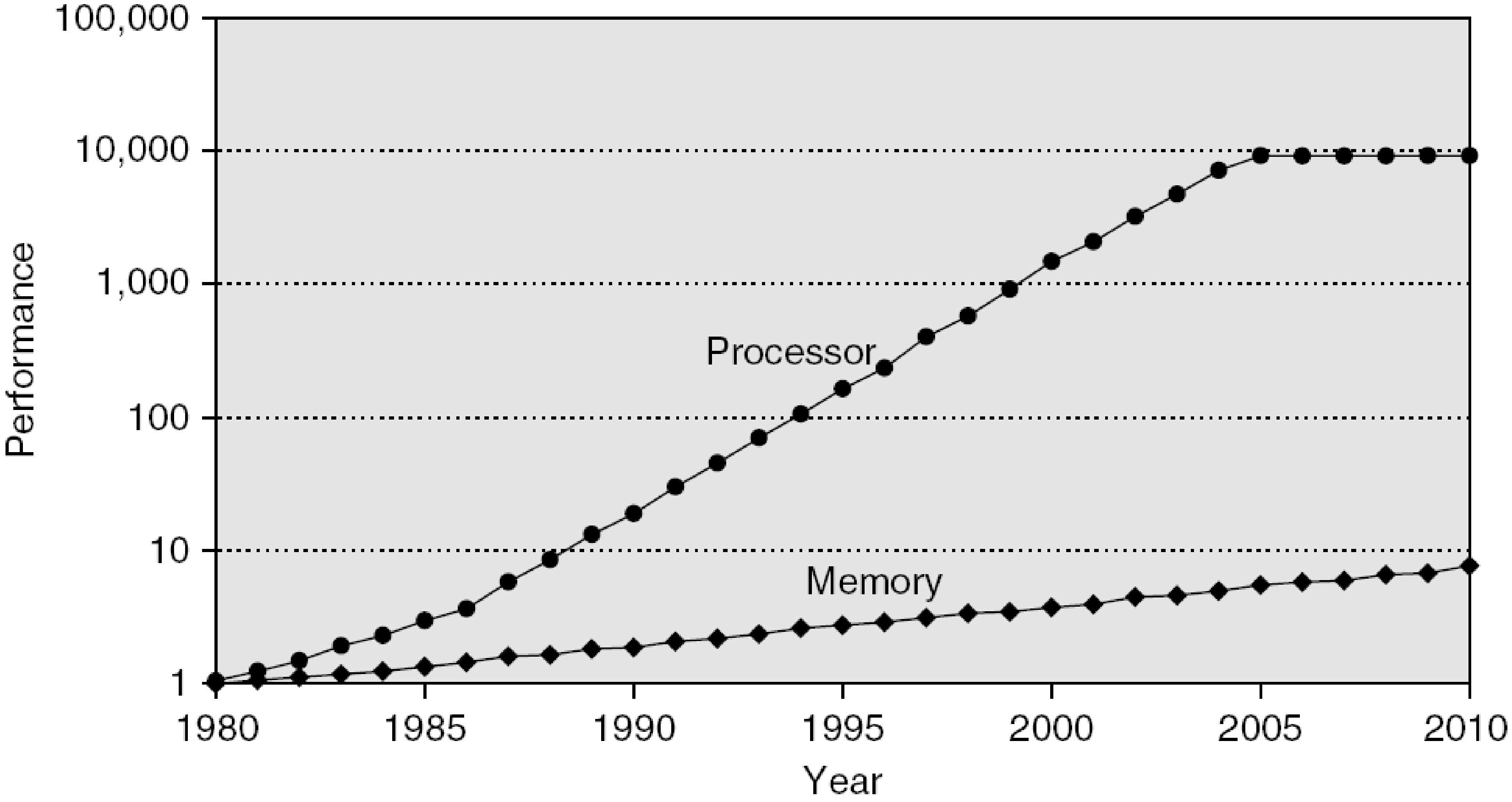
Norm Jouppi

- DEC, Hewlett-Packard, Google (TPUs)
- Winner of Eckert-Mauchly Award
- “The Nobel Prize of Computer Architecture”

Early computer system



Recall: Processor-memory gap



Ideal memory

We **want** a large, fast memory

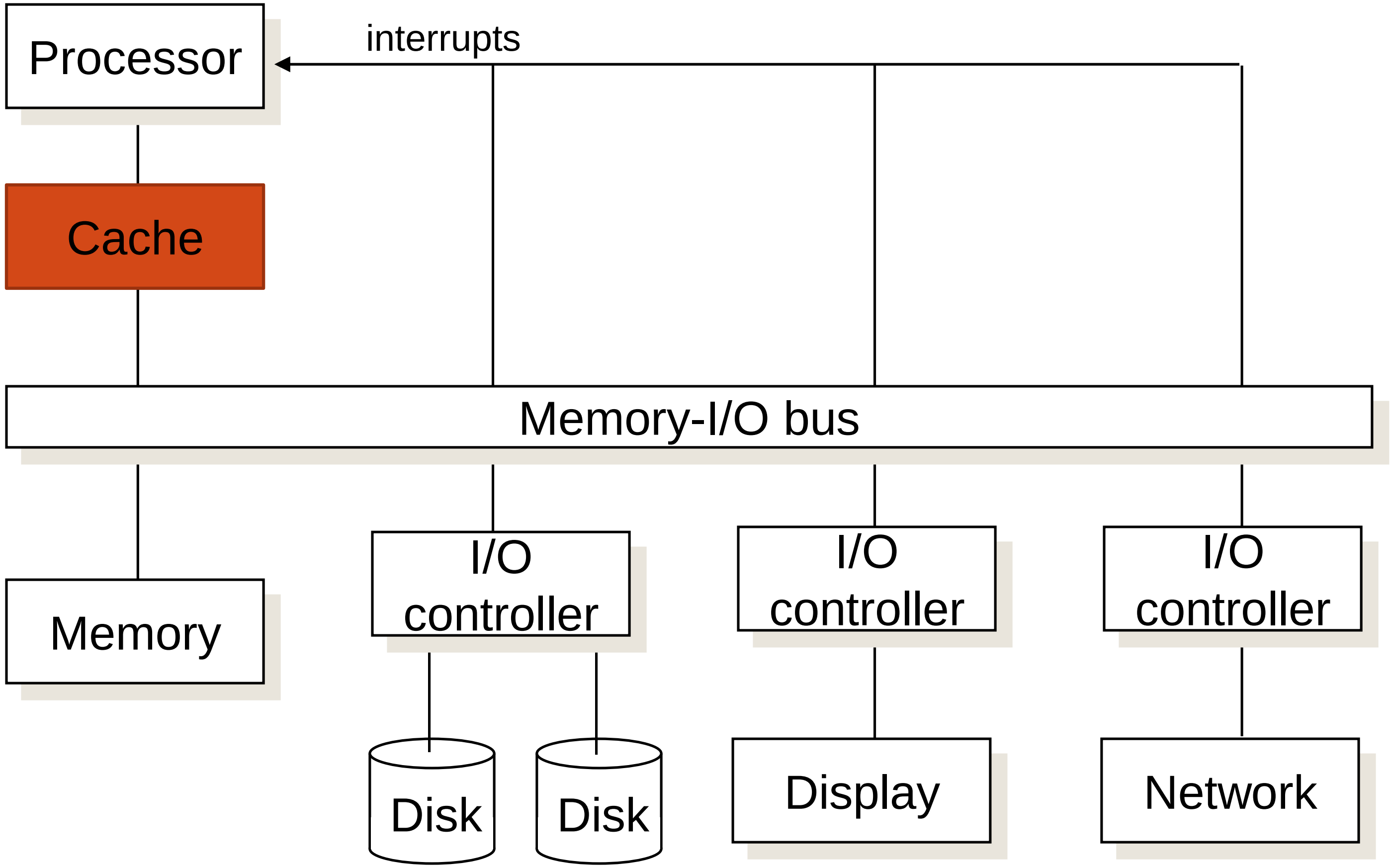
…But technology doesn’t let us have this!

Key observation: **Locality**

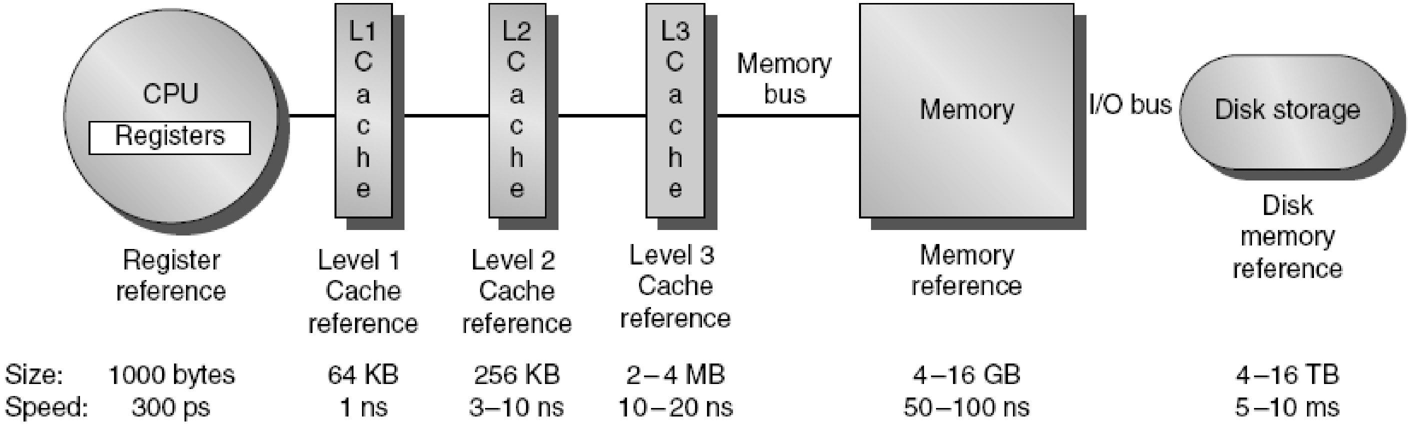
- All data are not equal
- Some data are accessed more often than others

Architects solution: Caches ➔ Memory **hierarchy**

Modern computer system

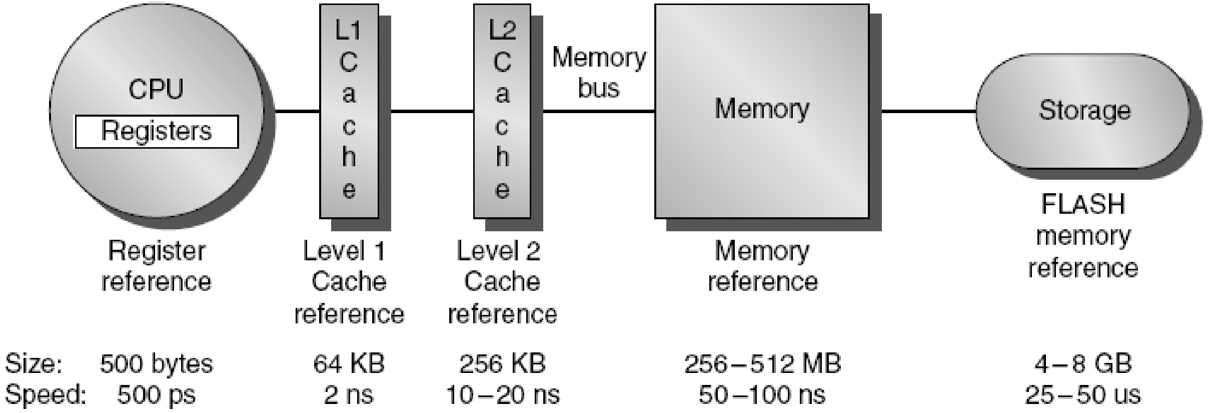


Technological tradeoffs in accessing data



Size:	1000 bytes	64 KB	256 KB	2 – 4 MB	4 – 16 GB	4 – 16 TB
Speed:	300 ps	1 ns	3–10 ns	10 – 20 ns	50 – 100 ns	5 – 10 ms

(a) Memory hierarchy for server

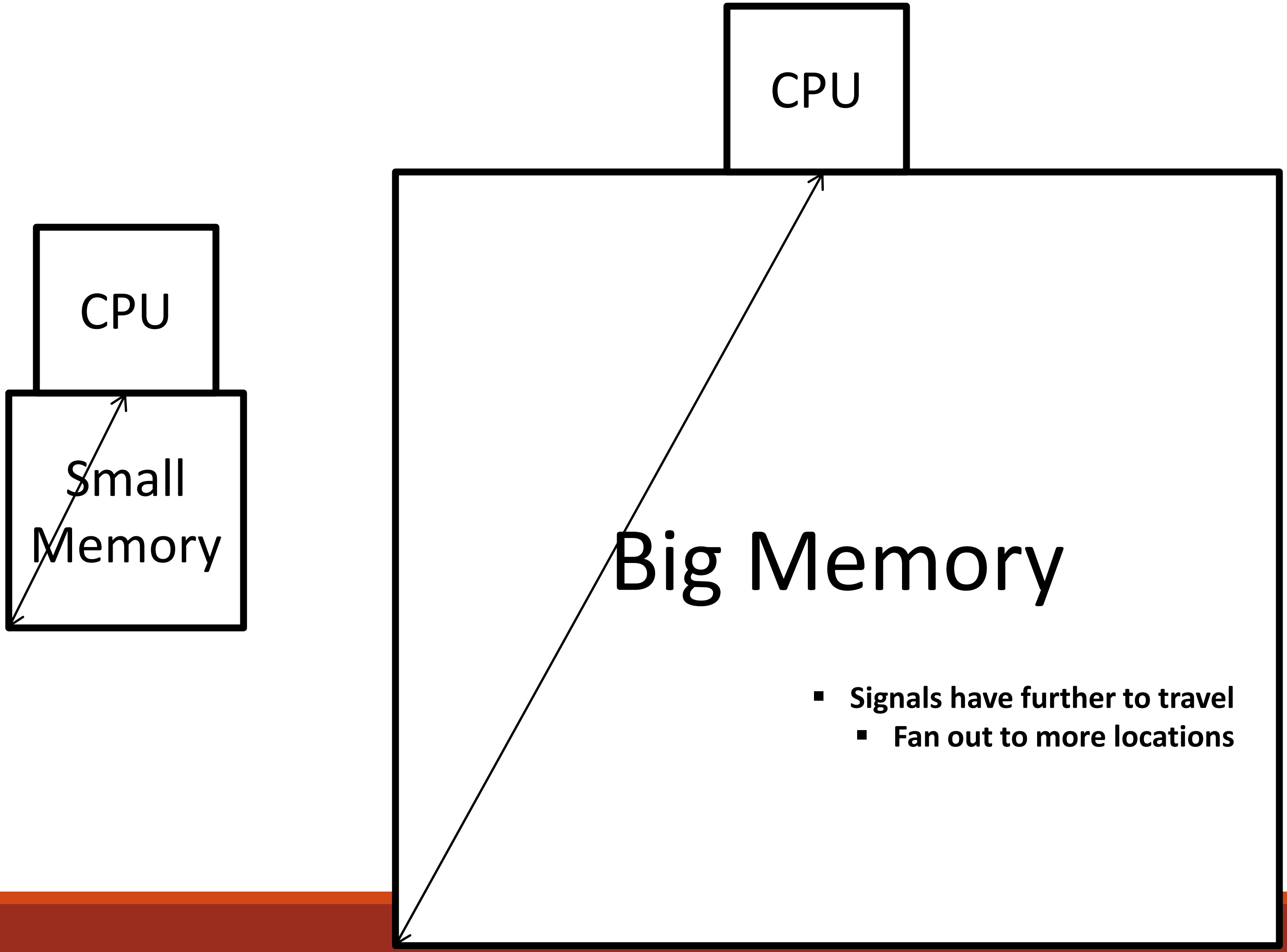


Size:	500 bytes	64 KB	256 KB	256 – 512 MB	4 – 8 GB
Speed:	500 ps	2 ns	10 – 20 ns	50 – 100 ns	25 – 50 us

(b) Memory hierarchy for a personal mobile device

Memory Technology

Physical size affects latency

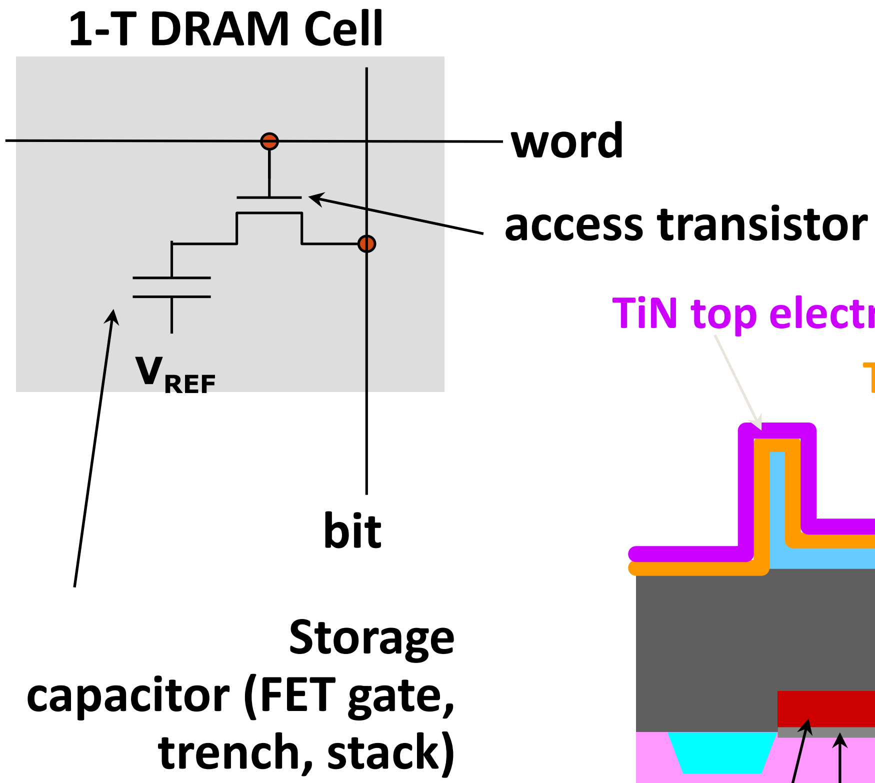


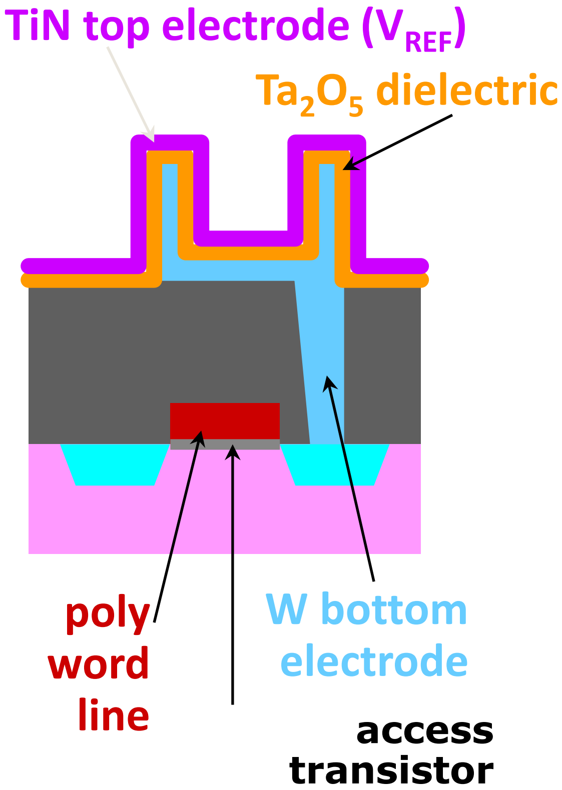
- Signals have further to travel
  - Fan out to more locations

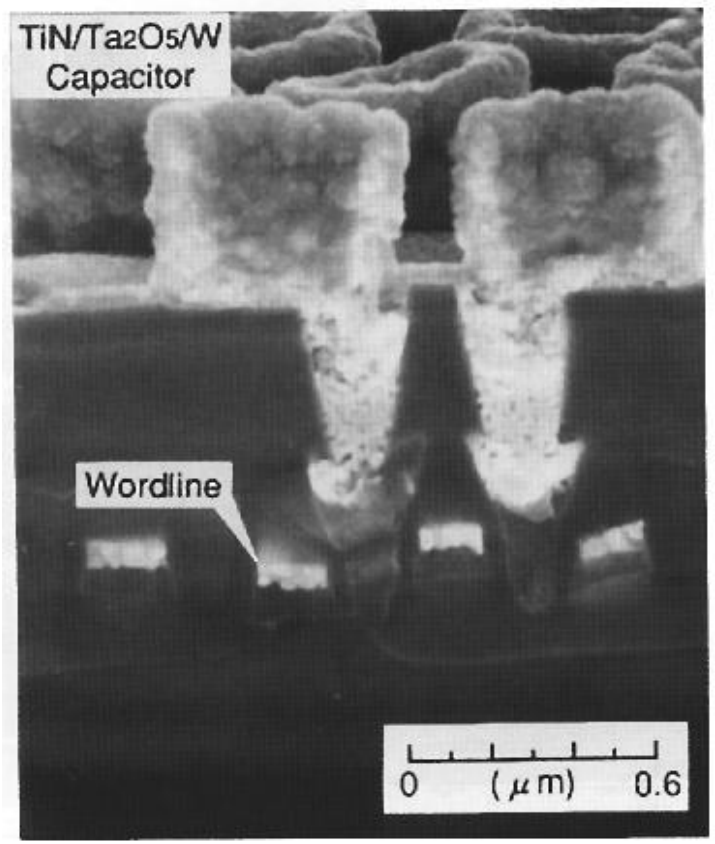
Why is bigger slower?

- Physics slows us down
- Racing the speed of light?
  - take recent Intel chip (Haswell-E 8C)
  - how far can I go in a clock cycle @ 3 GHz?
    (3.0x10^8 m/s) / (3x10^9 cycles/s) = 0.1m/cycle
  - for comparison: Haswell-E 8C is about 19mm = .019m across
  - ➔ speed of light doesn’t directly limit speed, but its in ballpark
- Capacitance:
  - long wires have more capacitance
  - either more powerful (bigger) transistors required, or slower
  - signal propagation speed proportional to capacitance
  - going “off chip” has an order of magnitude more capacitance

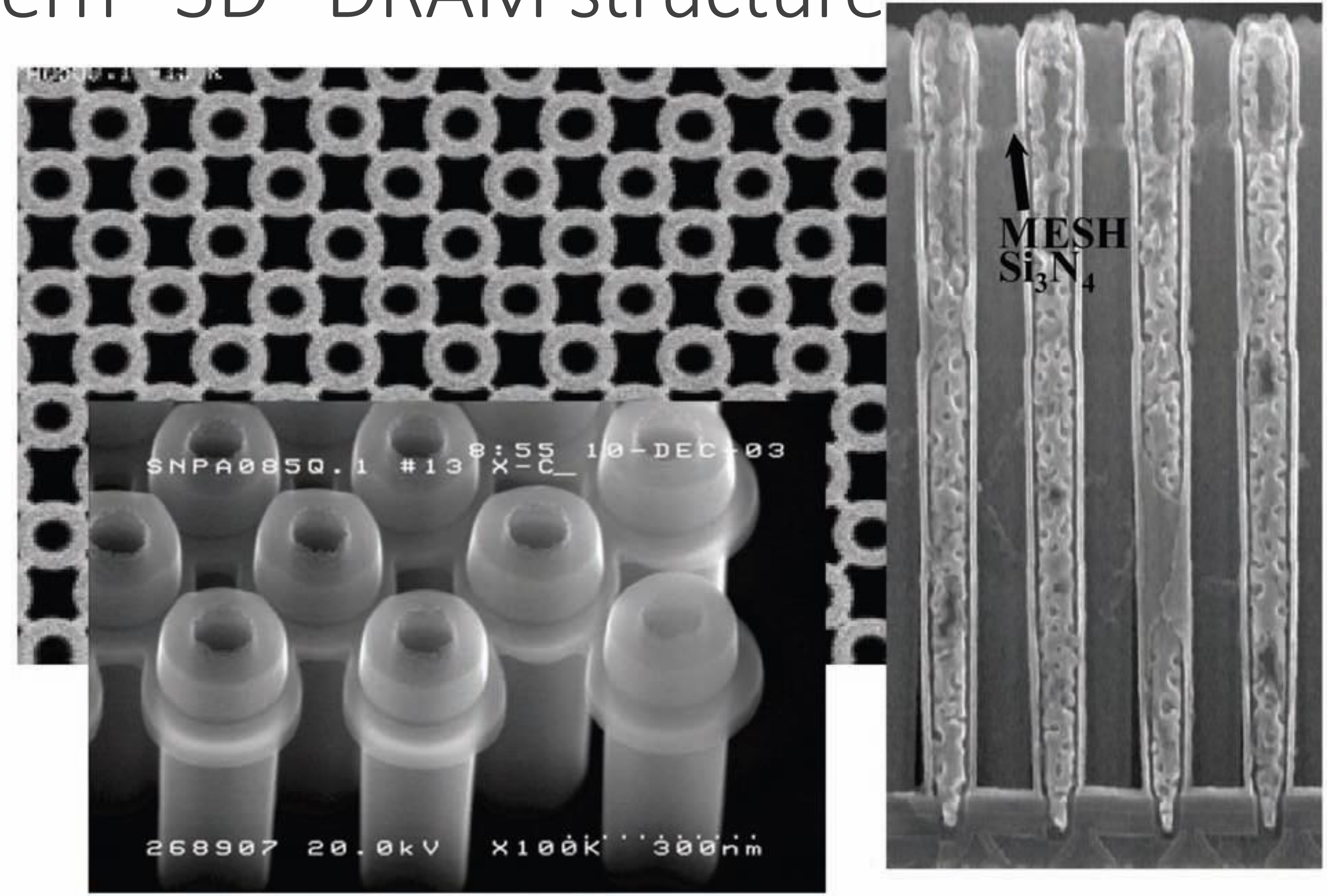
Single-transistor (1T) DRAM cell (one bit)





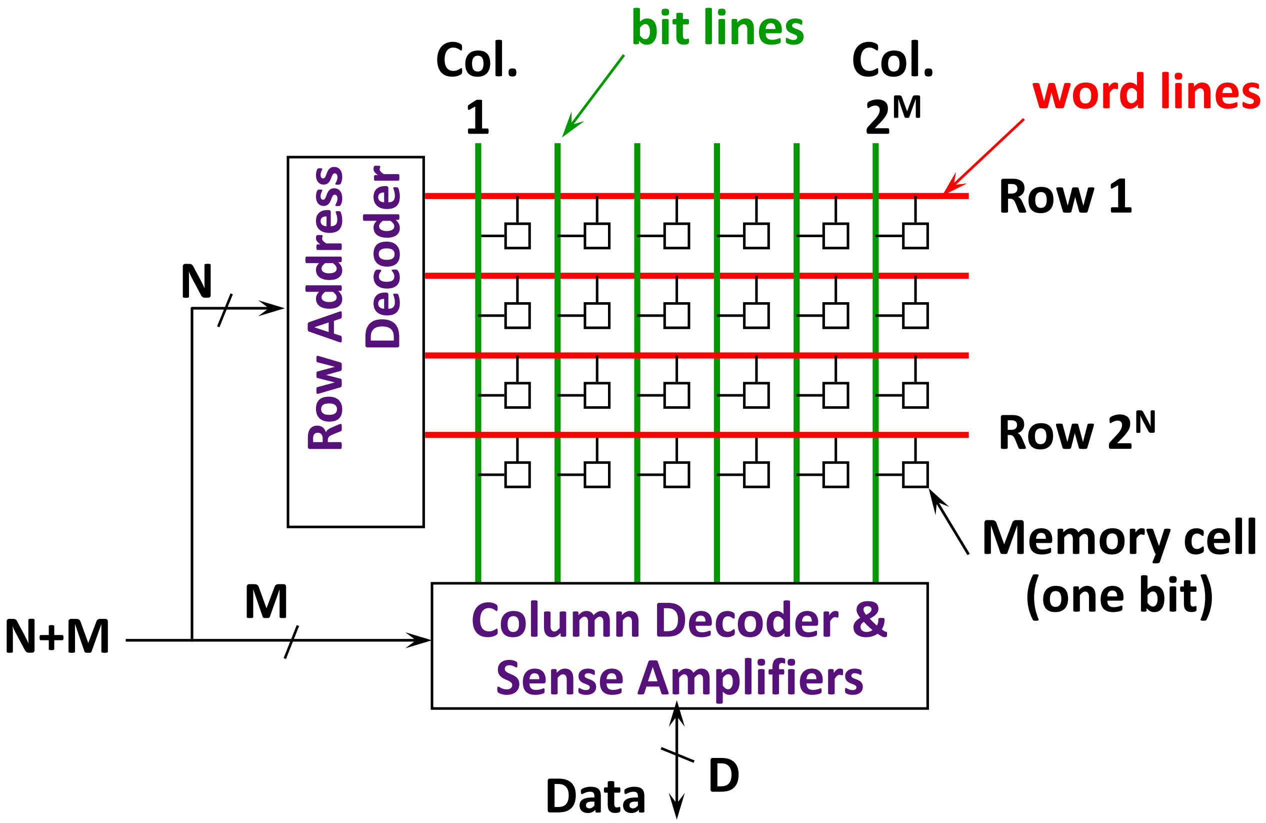


Modern “3D” DRAM structure



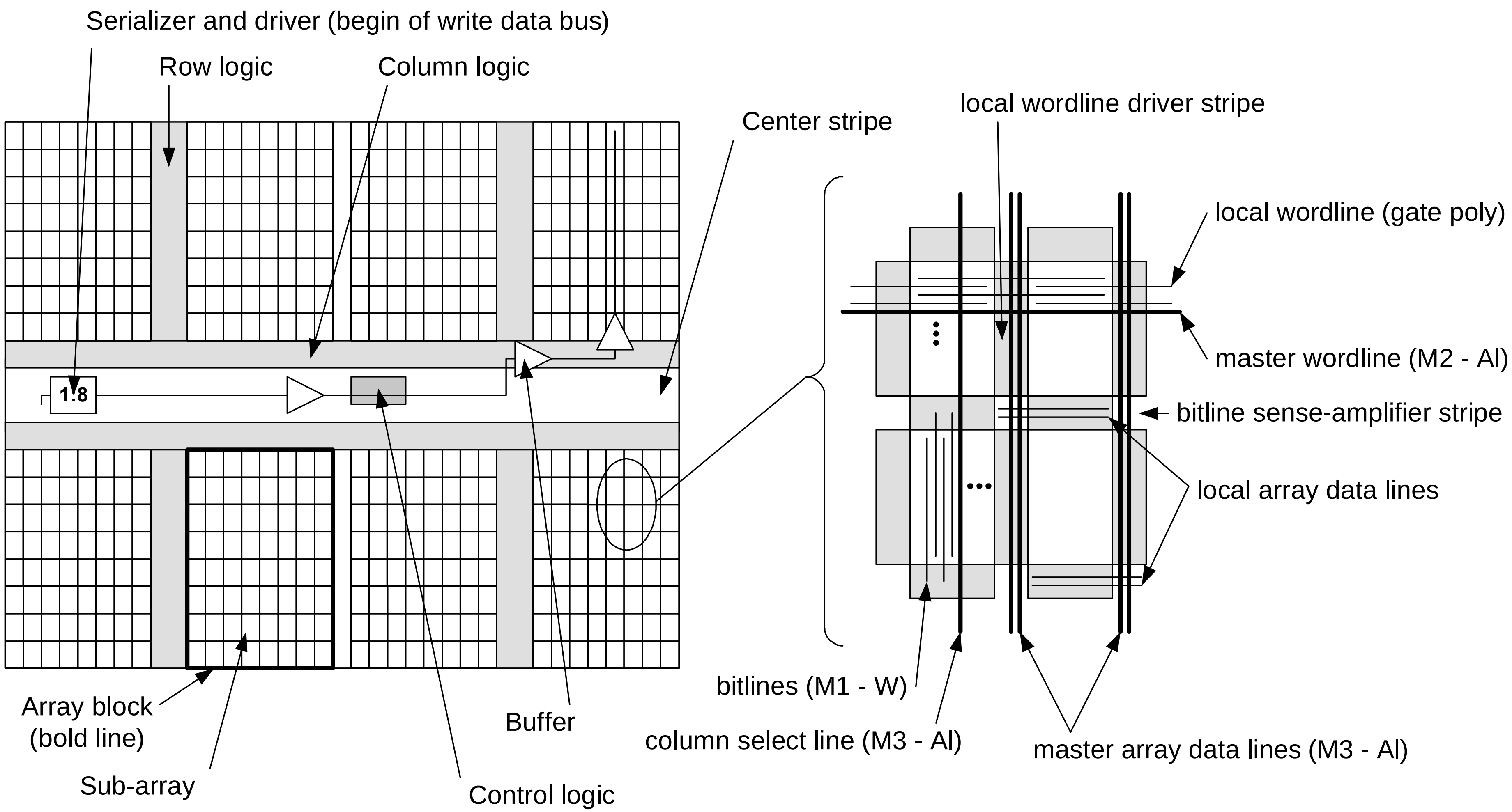
[Samsung, sub-70nm DRAM, 2004]

DRAM architecture



- Bits stored in 2-dimensional arrays on chip
- Modern chips have around 4-8 logical banks on each chip
  - each logical bank physically implemented as many smaller arrays

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

- 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

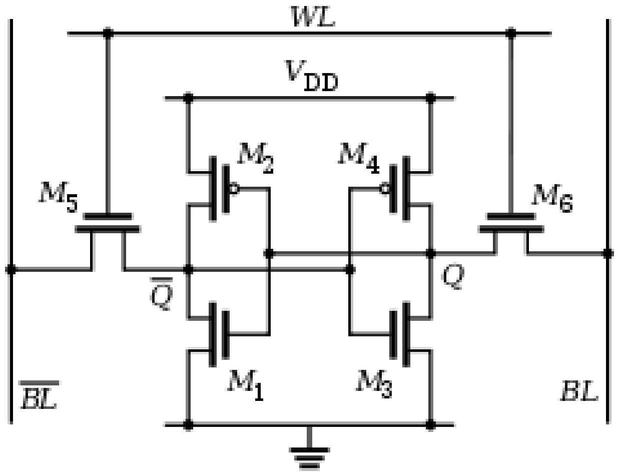
- Precharge
- Row access (RAS)

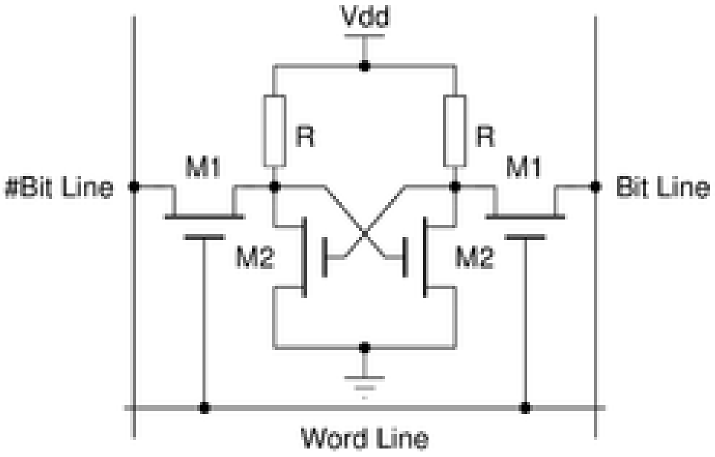
- Column access (CAS)
  - decode column address to select small number of sense amplifier latches (4, 8, 16, or 32 bits depending on DRAM package)
  - on read, send latched bits out to chip pins
  - on write, change sense amplifier latches which then charge storage cells to required value
  - **can perform 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

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)

Energy

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

SRAM vs DRAM

SRAM is simpler

- Non-destructive reads

SRAM is faster

DRAM is denser

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

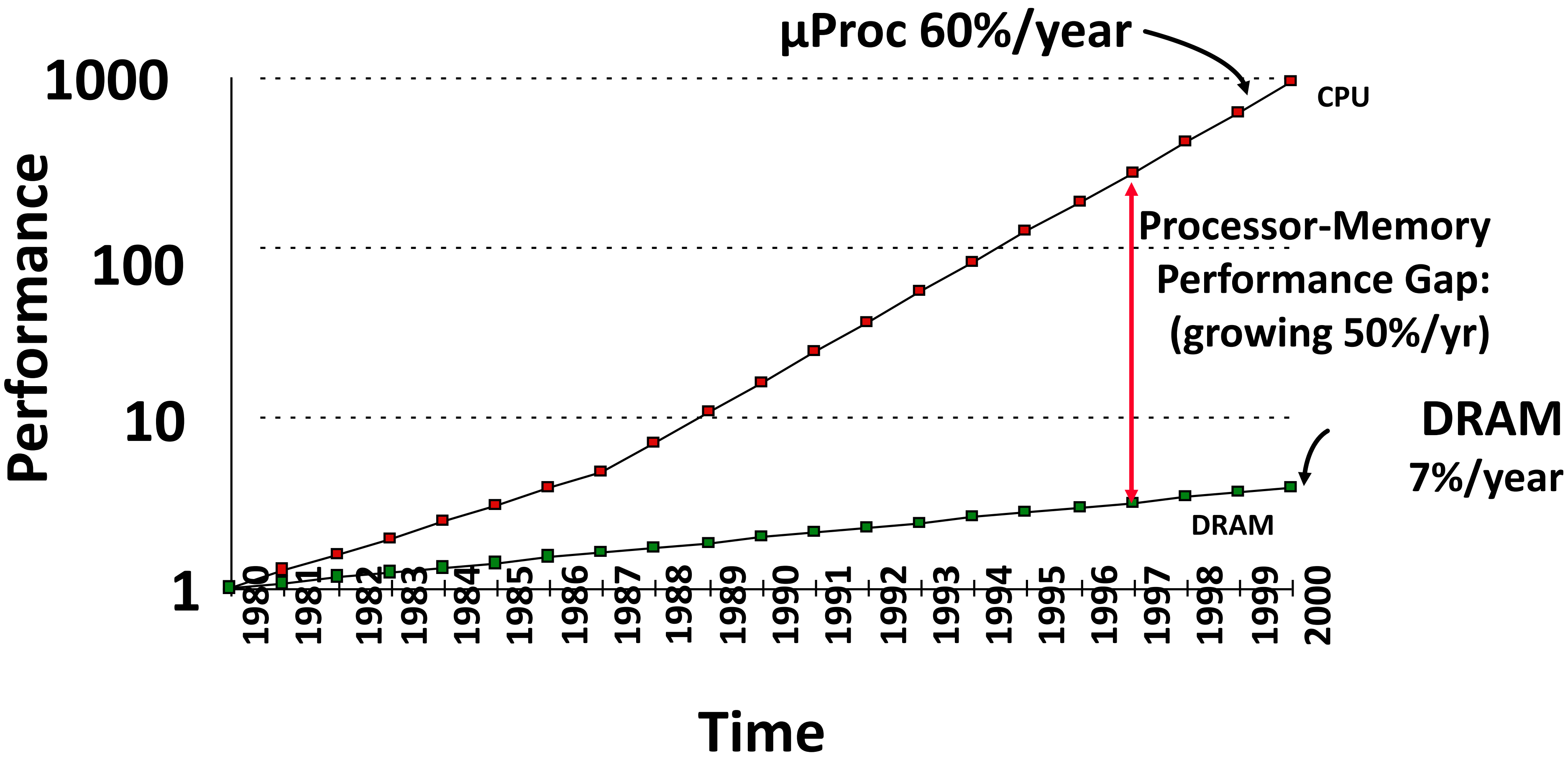
SRAM used for on-chip caches, register file

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

Processor-DRAM gap (latency)



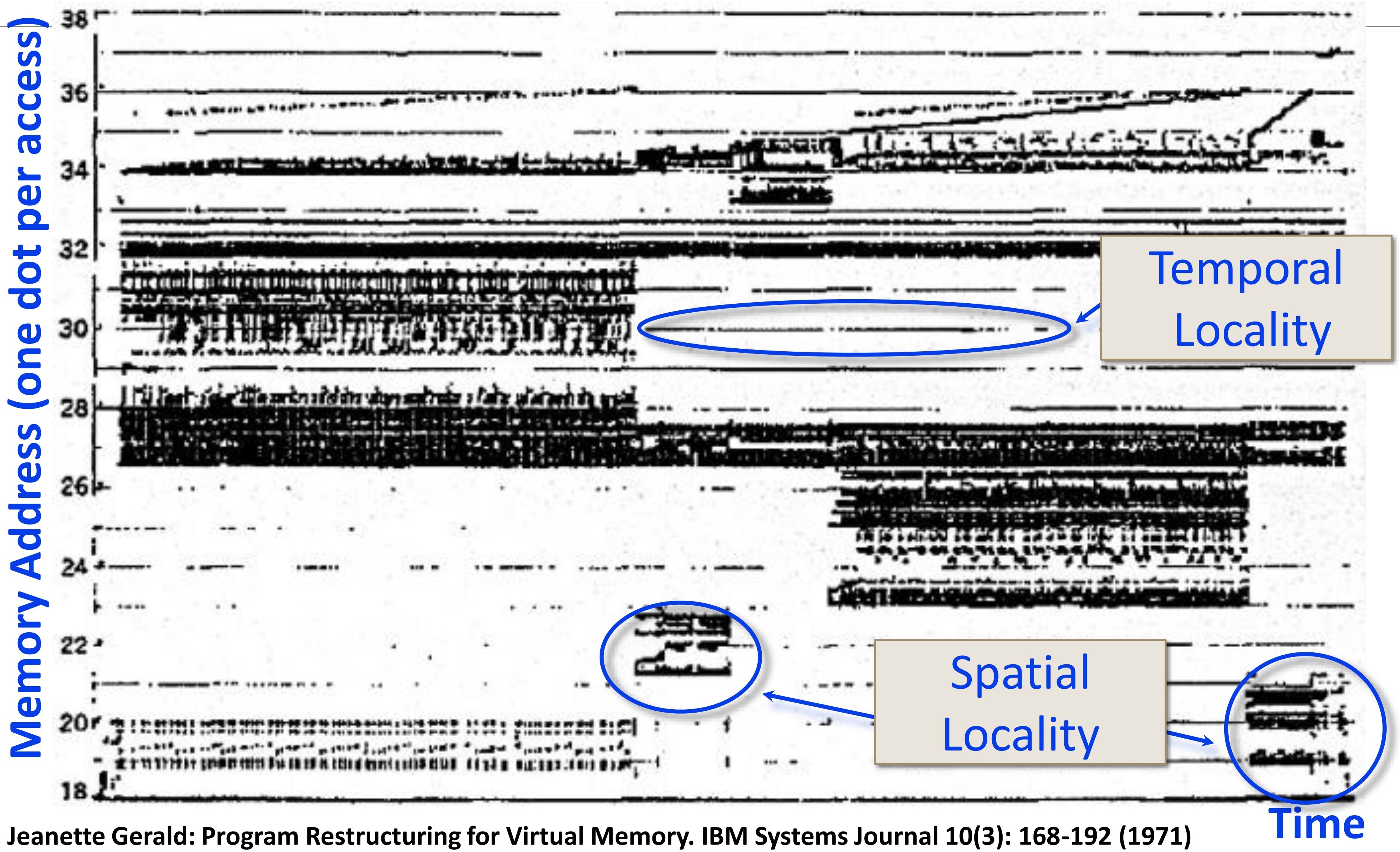
Four-issue 3GHz superscalar accessing 100ns DRAM could execute 1,200 instructions during time for one memory access!

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



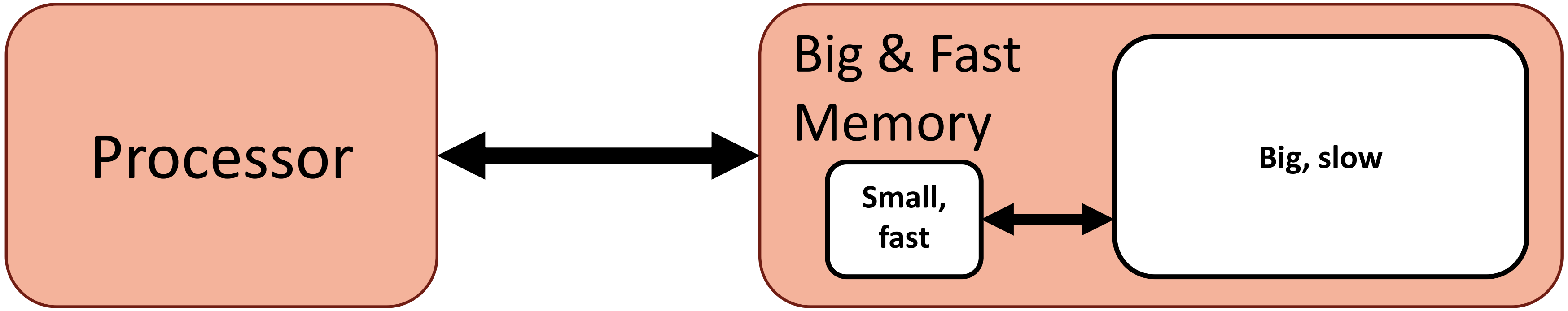
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

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 (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: von Neumann, but…split instruction/data caches; protection bits prevent execution of data; different optimizations, etc.

Instructions vs data in real systems

Where to store instructions & data?

Harvard architecture:

Princeton/von Neumann architecture:

Modern architecture: **von Neumann –** but …

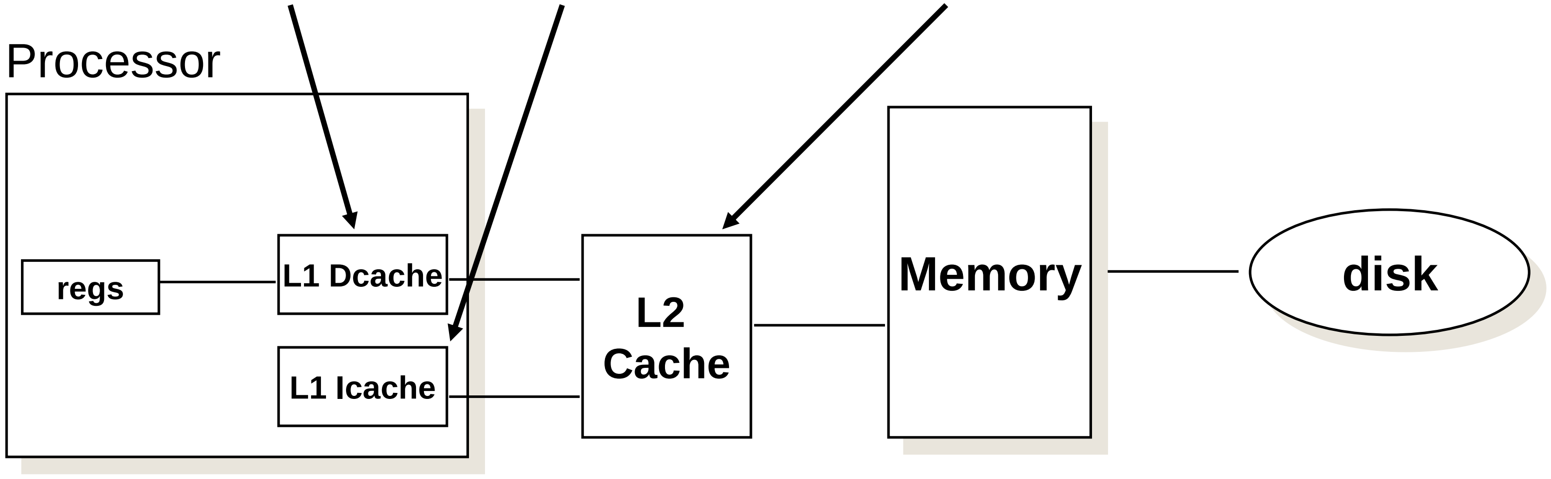
- Split instruction/data caches
- Protection bits prevent execution of data
- Different optimizations in instruction vs data caches (e.g., prefetching)
- Etc.

Lesson: Real systems inevitably compromise and try to get best of both worlds!

Split vs. unified caches

Why?

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



How does this affect self modifying code?

Alpha 21164

Microprocessor
Report 9/12/94

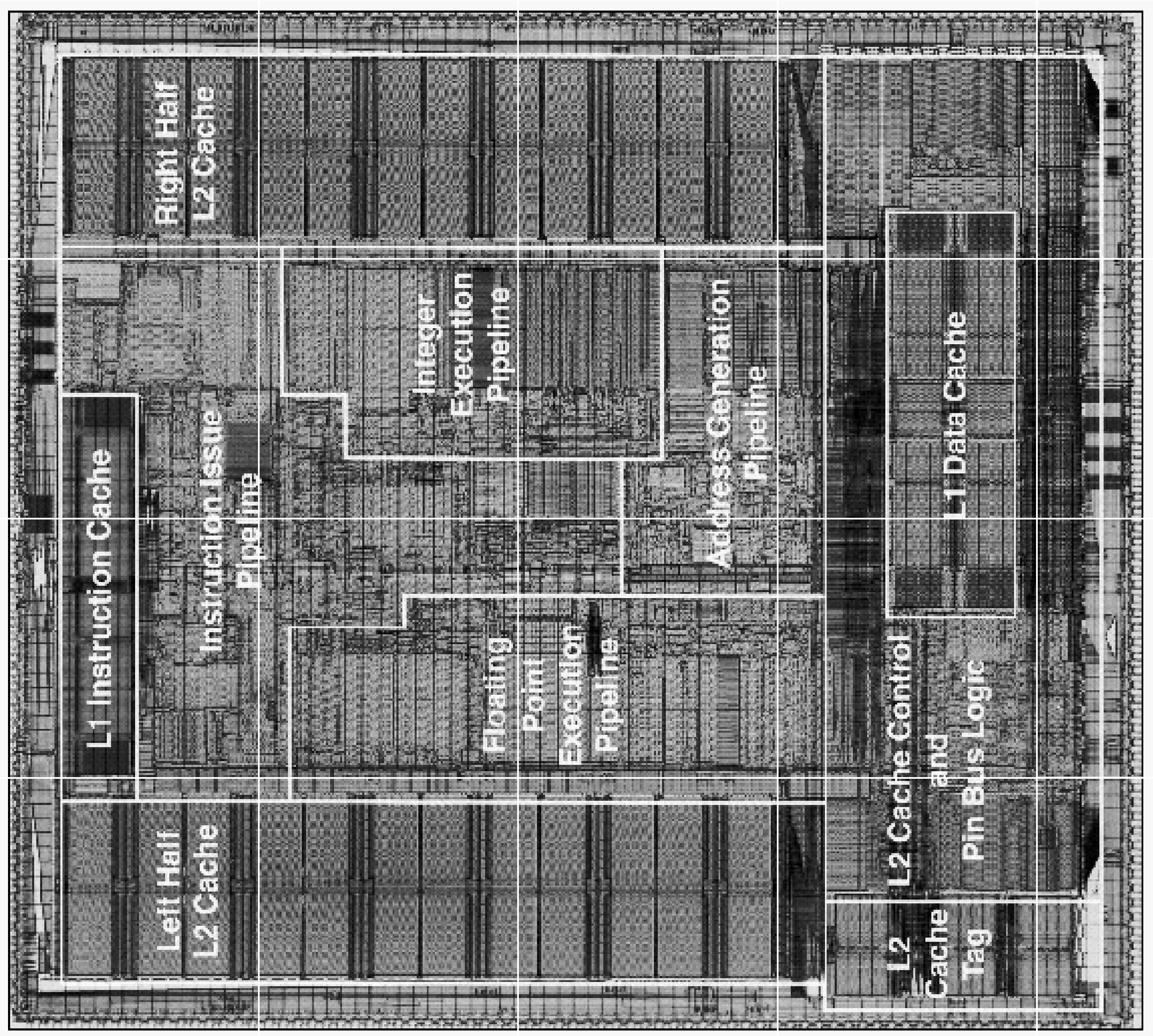
Caches:

L1 data

L1 instruction

L2 unified

+ L3 **off-chip**

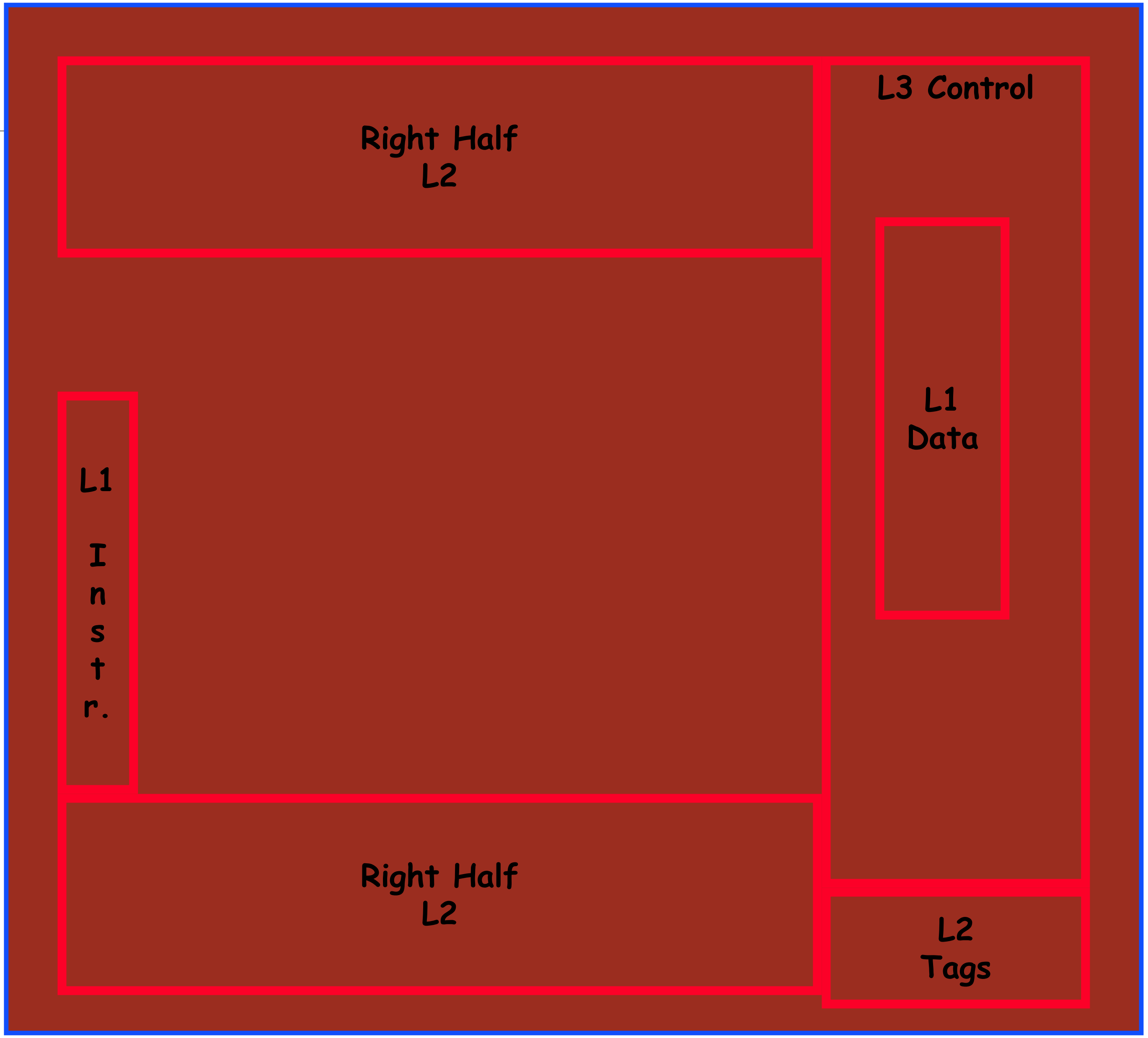

Right Half L2 Cache
Integer Execution Pipeline
Address Generation Pipeline
L1 Data Cache
L1 Instruction Cache
Instruction Issue Pipeline
Floating Point Execution Pipeline
L2 Cache Control and Pin Bus Logic
Left Half L2 Cache
L2 Cache Tag

Alpha 21164

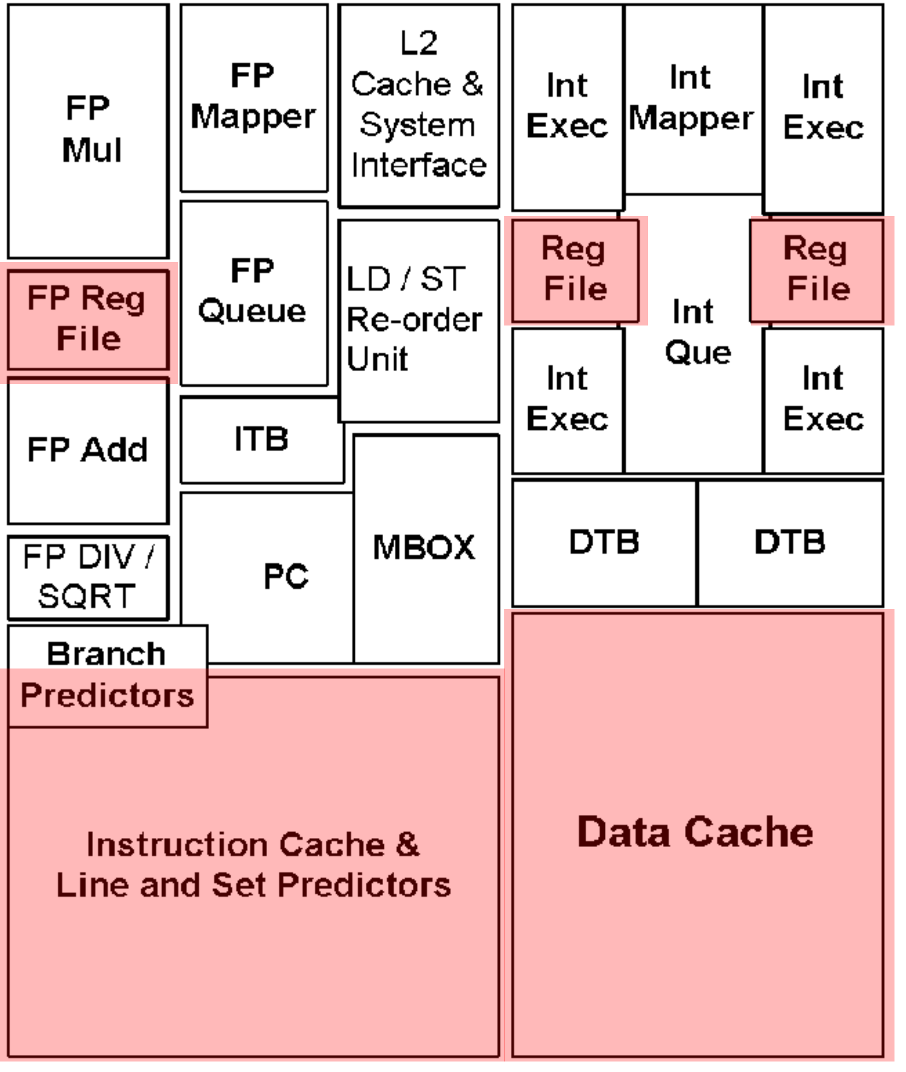
Microprocessor Report 9/12/94

Caches:

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



Alpha 21264



- 21264 Floorplan
- Register files in middle of execution units
- 64k instr cache
- 64k data cache
- Caches take up a large fraction of the die
  - ≈**30-50%** in recent chips

(Figure from Jim Keller, Compaq Corp.)

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*
- *Prefetchers* speculate about next accesses and fetch them into cache

Note: Cache contents are not “architectural”!

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 variable (temporal, allocated to register)
- **Instructions**
  - – Reference instructions in sequence (spatial)
  - – Cycle through loop repeatedly (temporal)

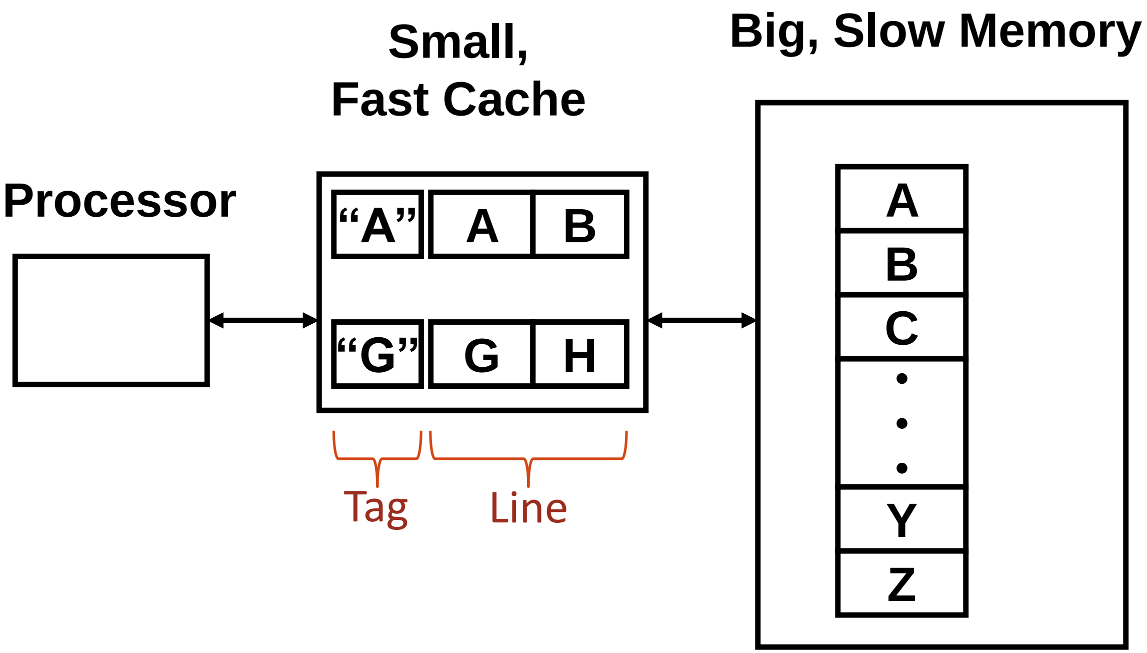
Caching: The basic idea

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** of multiple words
  - Exploit spatial locality
  - Amortize overheads



Access

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

MPKI and AMAT

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

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

Three ways to improve memory performance:

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

There’s a tension between these

MPKI and AMAT in parallel programs

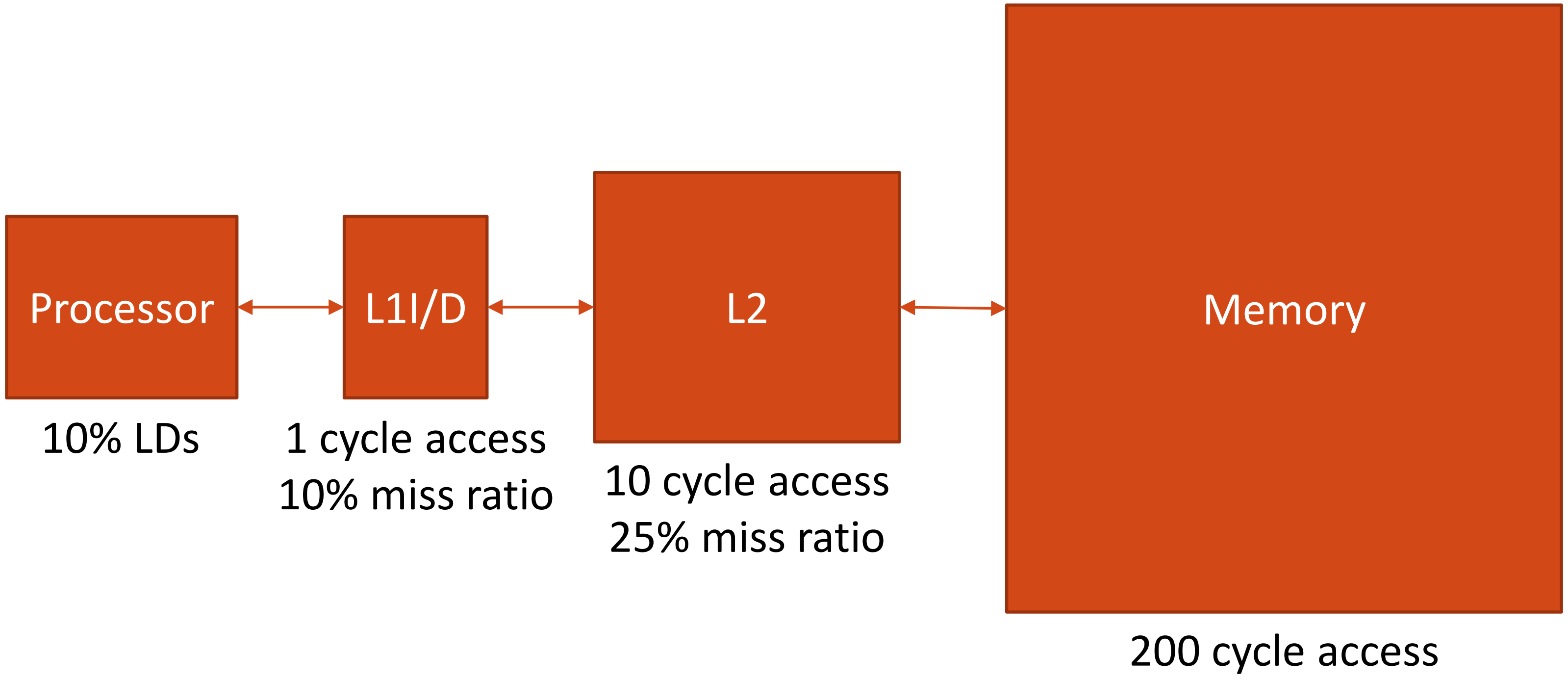
$$\text{MPKI} = \frac{\text{Misses}}{1000\ \text{Instructions}} = \frac{\text{Miss ratio} \times \text{Memory accesses}}{1000\ \text{Instructions}} = \text{Miss ratio} \times \frac{\text{Memory accesses}}{1000\ \text{Instructions}}$$

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

Note that speculative and multithreaded processors may execute other instructions during a miss

- Reduces performance impact of misses
- Memory-level parallelism (MLP) overlaps miss latency

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?

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)

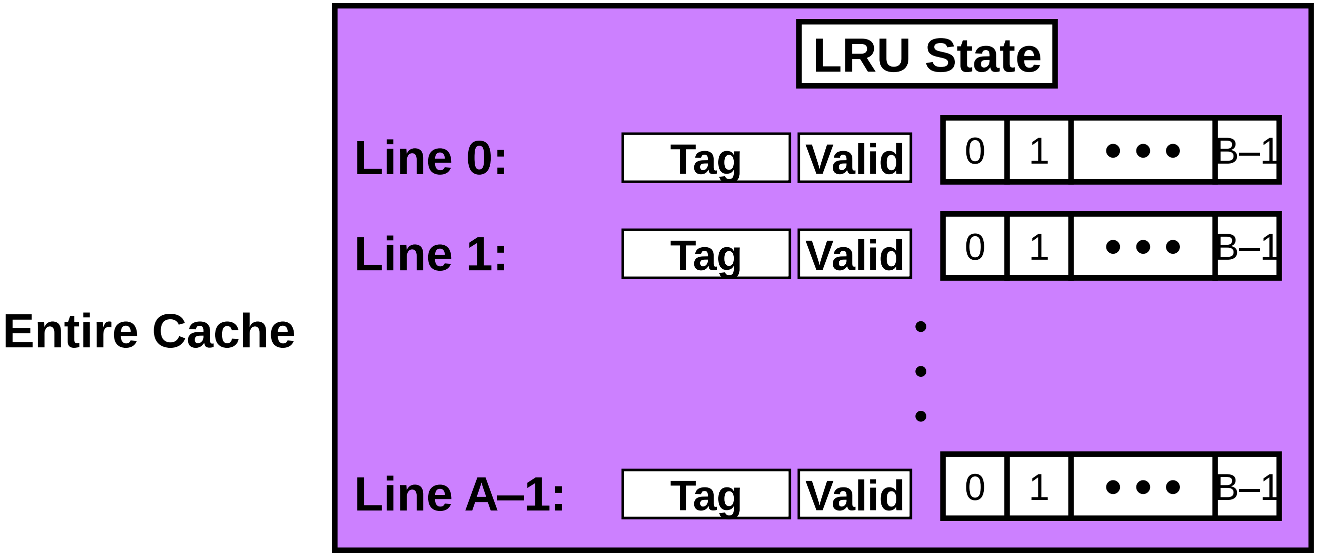
Constraints:

- 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

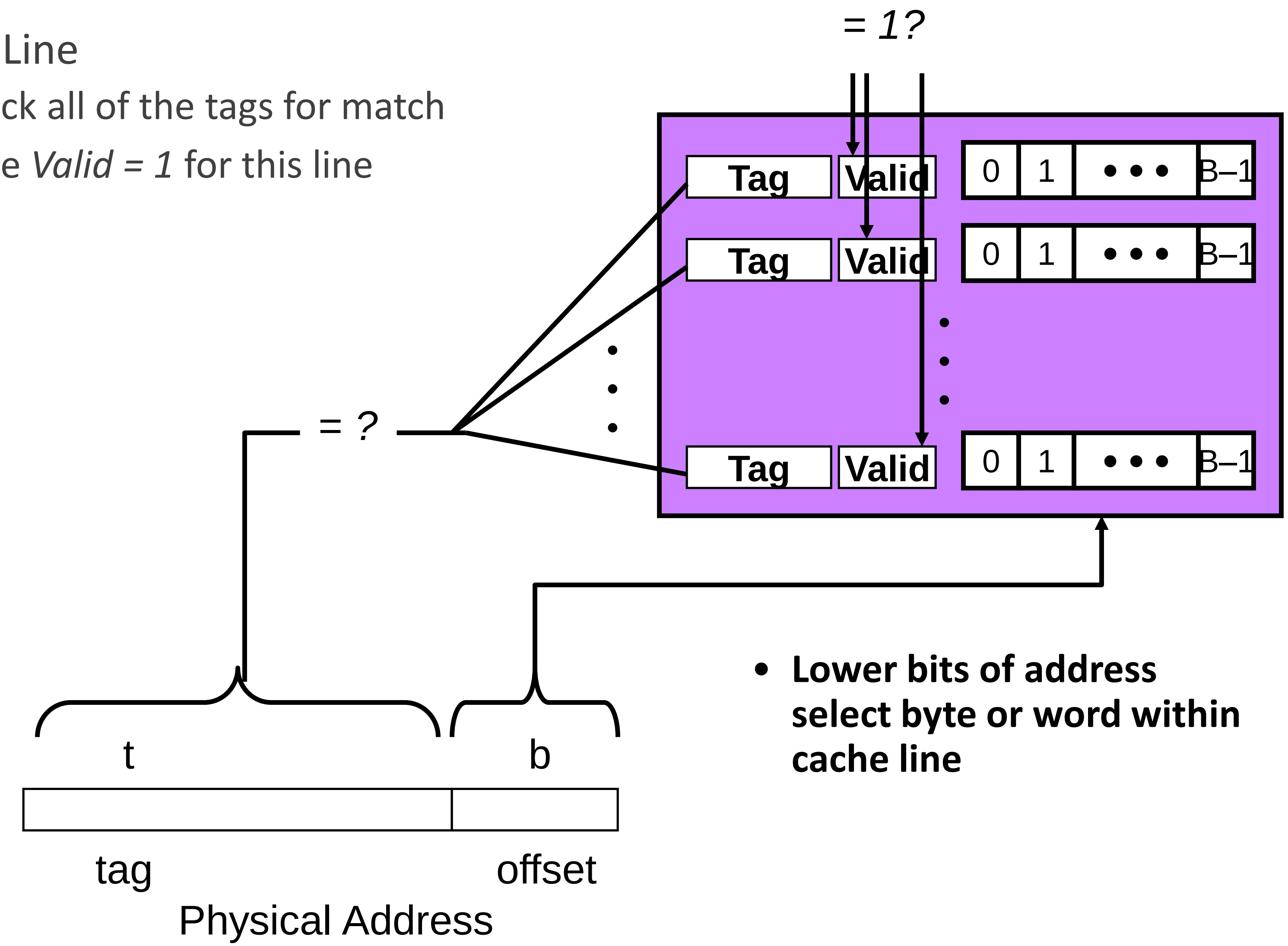
- Cache consists of single set holding A=S lines
- Given memory line can map to any line in set
- Only practical for small caches
- Useful for analysis and simulation
- Common in software caches



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

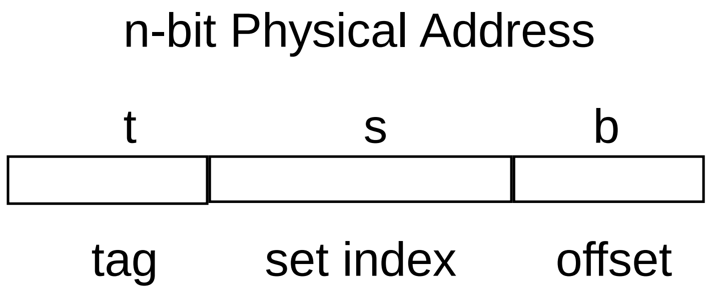
- Each memory line has a **unique** cache location

Parameters

- Line (aka block) size $B = 2^b$
    - Number of bytes in each line
    - Typically 2X–8X word size
- Number of sets $S = 2^s$
    - Number of lines cache can hold
- Total Cache Size $= B*S = 2^{b+s}$

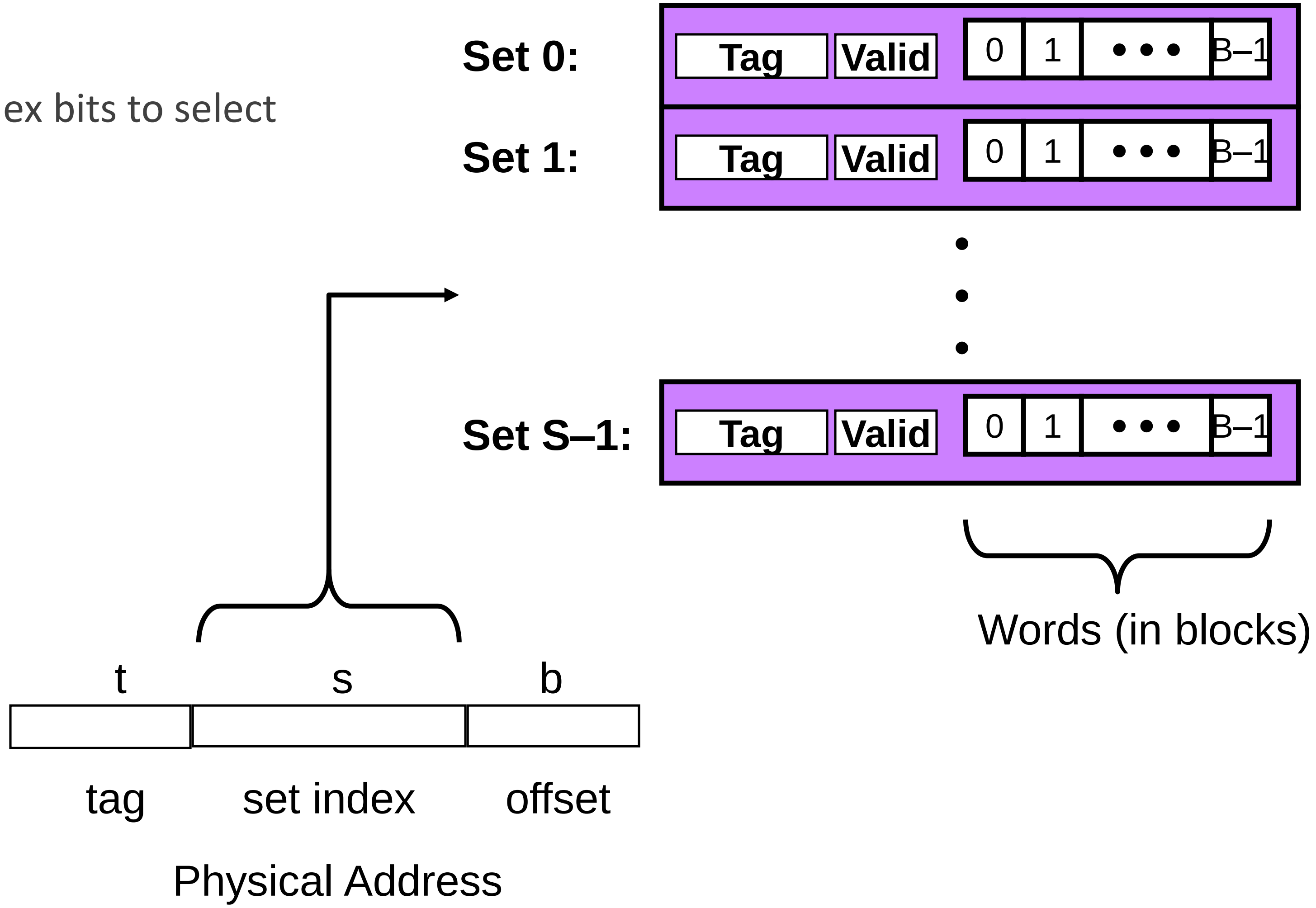
Physical Address

- Address used to reference main memory
- *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

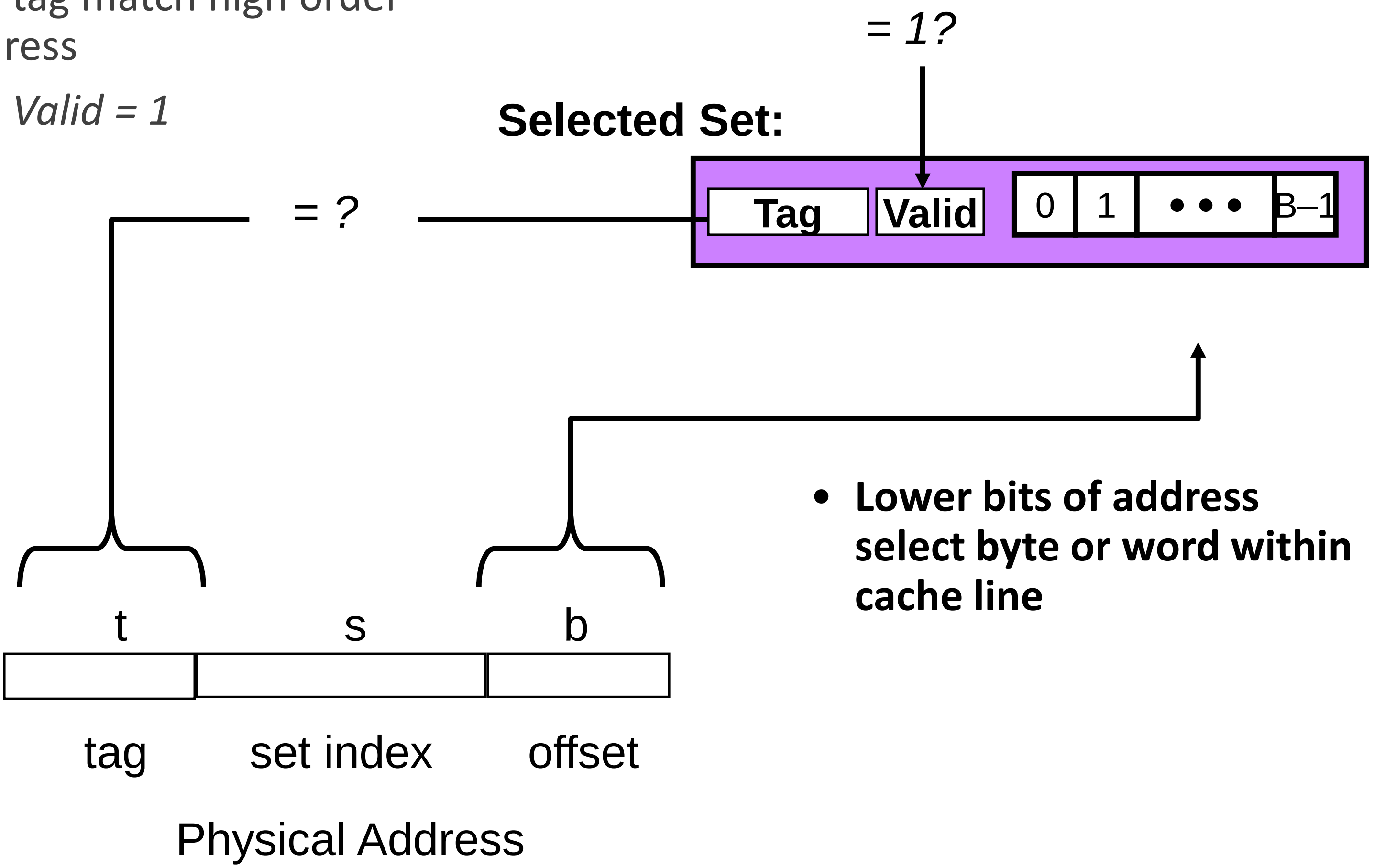
- Use set index bits to select cache set



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**

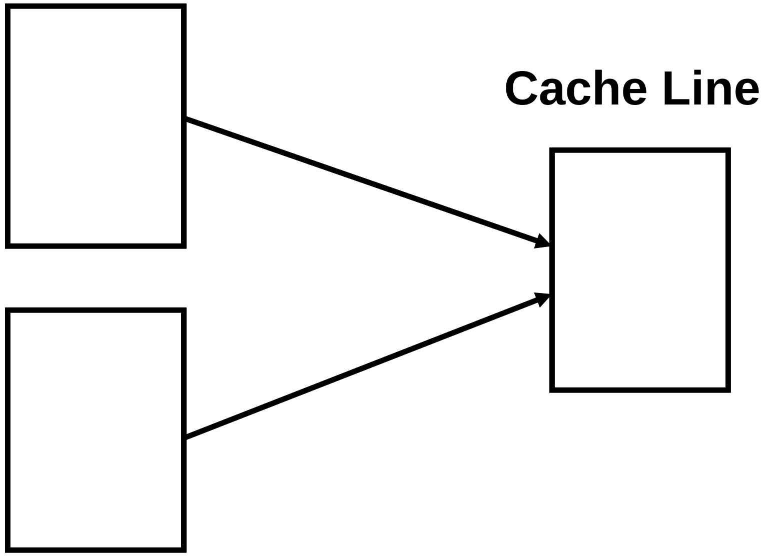
Tradeoffs of direct-mapped caches

Strength

- Minimal control hardware overhead
- Simple design
- (Relatively) easy to make fast

Weakness

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



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;
}
```

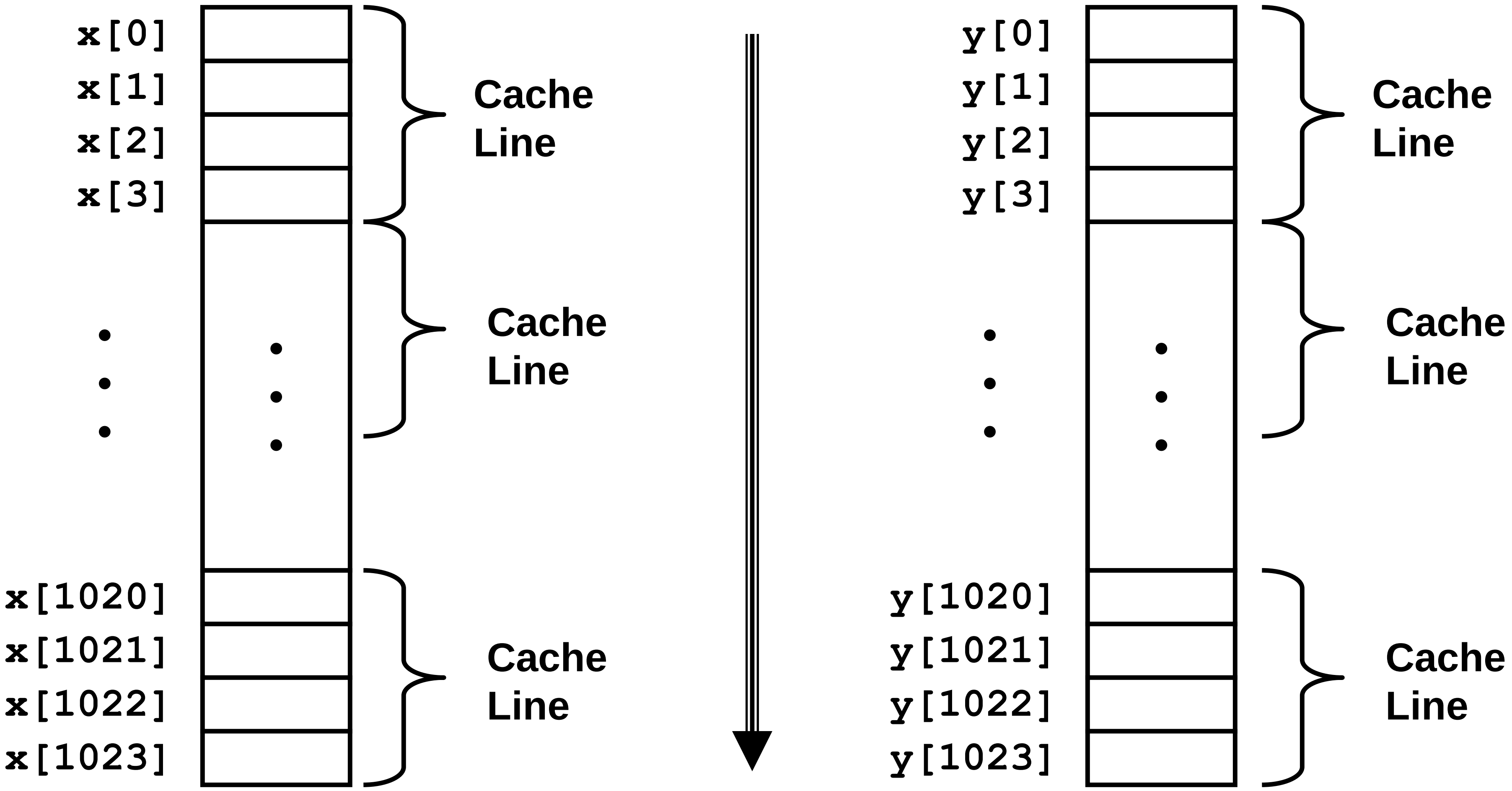
Machine

- DECStation 5000
- MIPS Processor with 64KB direct-mapped cache, 16 B line size

Performance

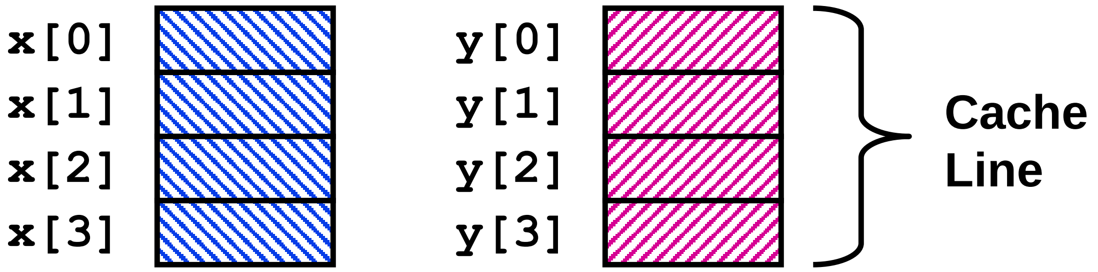
- Good case: 24 cycles / element
- Bad case: 66 cycles / element

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

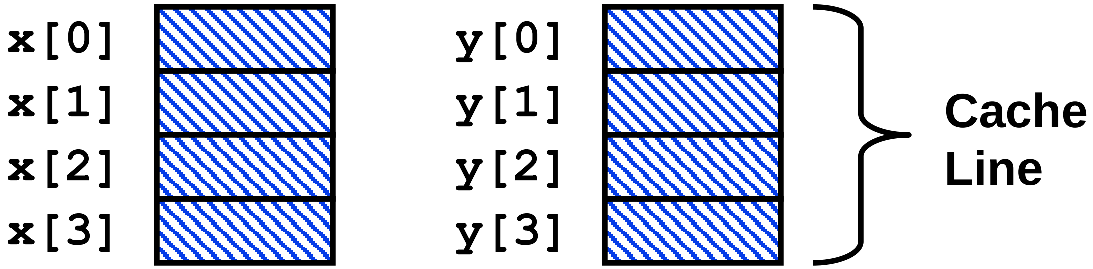
Analysis

- 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

- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration =
  10 + 0.25 * 2 * 28

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

- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration =
  10 + 1.0 * 2 * 28

Impact of increasing block size

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

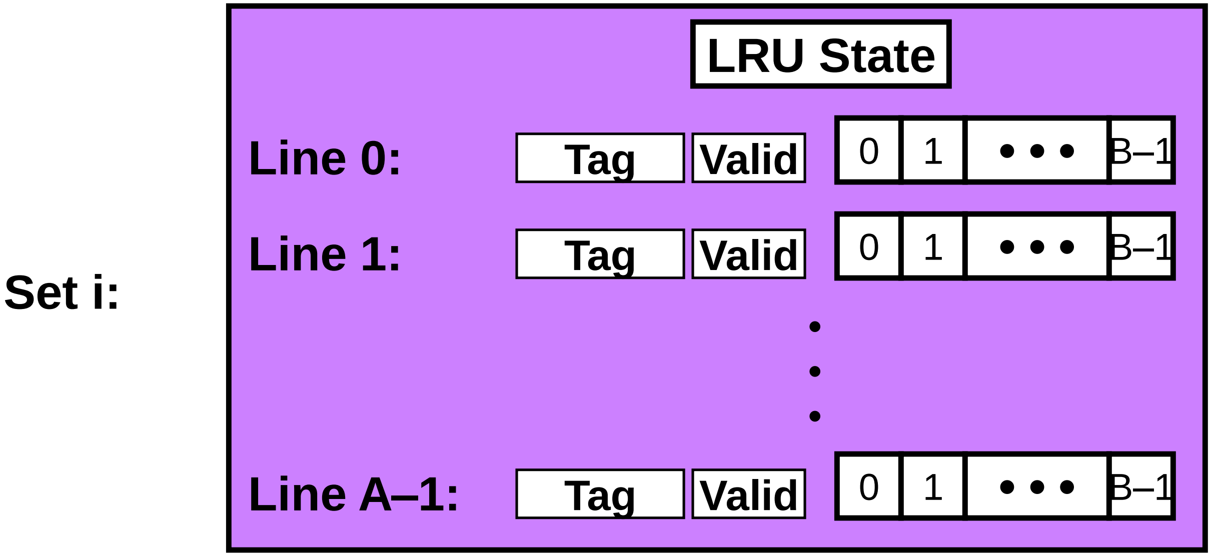
Set-associative cache

Mapping of Memory Lines

- Each set can hold A lines (usually A=2-8 for L1, A=8-32 for L3)
- Given memory line can map to any entry within its given set

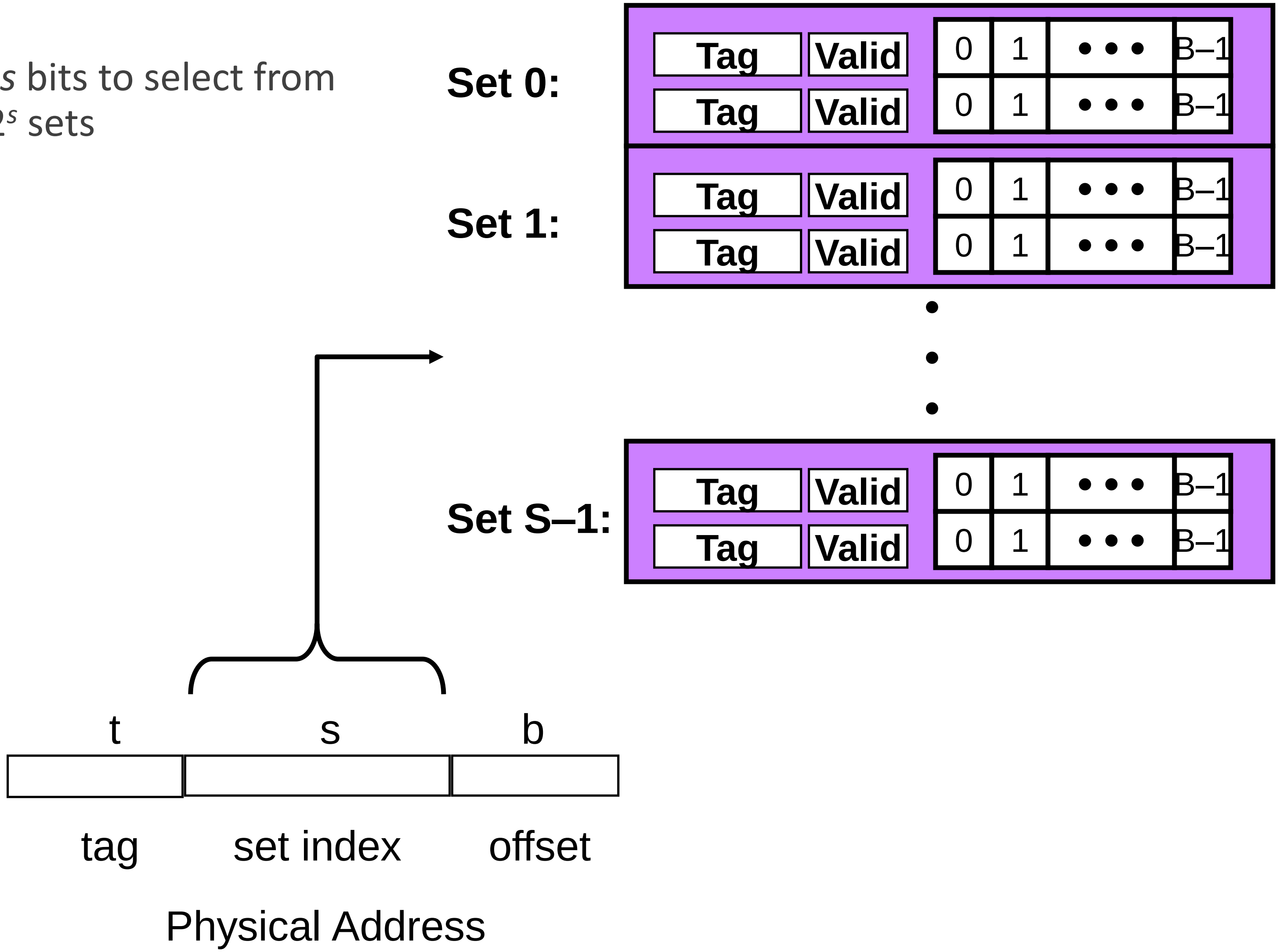
Tradeoffs

- Fewer conflict misses
- Forced by virtual memory
- Longer access latency
- More complex to implement



Indexing a 2-way set-associative cache

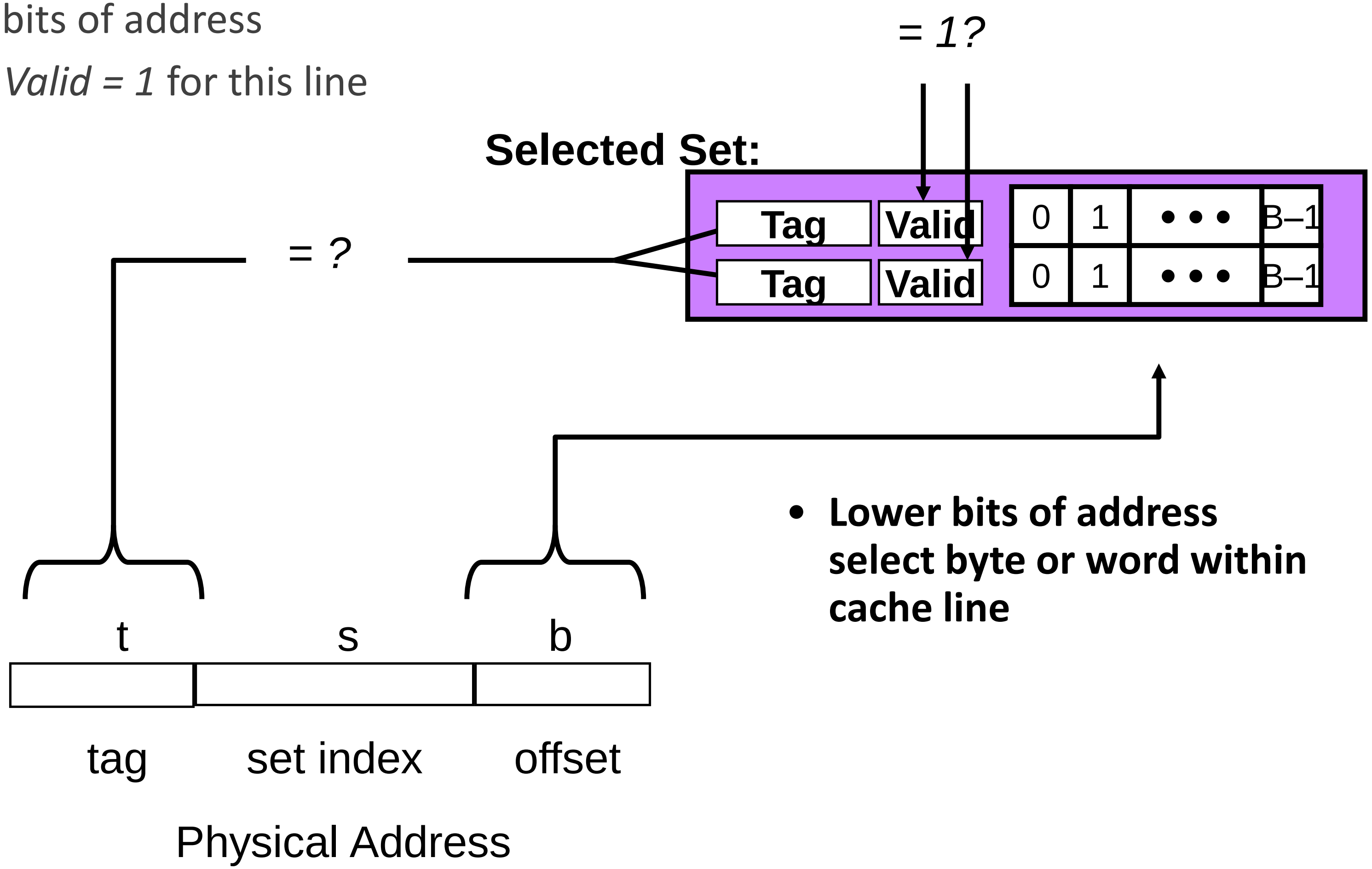
- Use middle *s* bits to select from among S = 2^s sets



Set-associative tag matching

Identifying Line

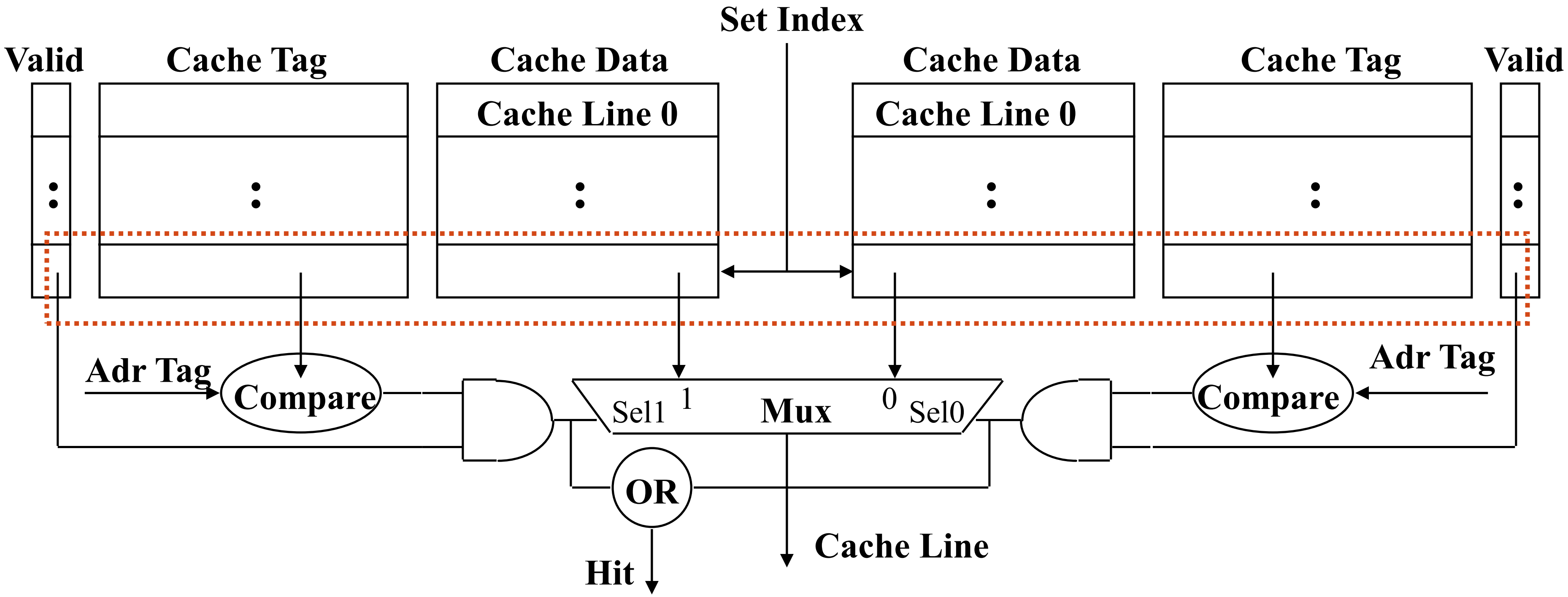
- Must have one of the tags match high order bits of address
- Must have *Valid = 1* for this line



- **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

What is associativity?

Simple answer: number of replacement candidates

More associativity ➔ better hit rates

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

What is associativity?

What about…

Victim caches

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

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

What is associativity?

What about…

Hashing

- Hash address to compute set
- Reduce conflict misses
- Add latency + tag size + complexity

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

What is associativity?

What about…

Skew-associative caches [Seznec, ISCA‘93]

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

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

Associativity through the lens of probability

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

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
- **Must know the future** (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)

Implementing replacement algorithm

- FIFO: Keep per-set counter, replace block at counter offset + increment
- Random: Like FIFO, but a global counter instead
- Naïve LRU: encode ordering within set (n log n bits) + state machine
- Simple LRU: track time in # accesses, each candidate stores timestamp it was last accessed
  - Tradeoff?
  - Efficiency vs complexity
  - Coarsened ages (eg, high bits of timestamp) save space with ~no performance loss

Eviction algorithms are active research area

- Fix pathologies in, eg, LRU [Qureshi, ISCA'07]
  - E.g.: ???
- Shared caches ("thread-aware" variants, cache partitioning) [Qureshi, MICRO'06]
  - Throughput vs fairness vs latency targets [Kasture, ASPLOS'14]
- Different object sizes
  - E.g., compressed caches, software caches [Pekhimenko, HPCA'15]
- How to predict future reuse?
  - PC of referencing instruction ← (turns out to be an excellent predictor) [Jain, ISCA'16]
- Perceptron (i.e., neural network) predictors [Teran, MICRO'16][Jiminez, MICRO'17]
- Guaranteeing theoretical properties
  - E.g., convex miss curves [Beckmann, HPCA'15]
- Ways to think about things more rigorously? [Beckmann, HPCA'17][Beckmann, NSDI'18]

Categorizing misses: The 3 C's++

Compulsory misses - unchanged

Capacity Misses – cache not big enough

- Capacity misses = fully associative misses with optimal replacement – 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 – replacement misses

Impact of Replacement Policy

Improving replacement policy
(eg, random ➔ LRU)

Effect on cache area (tags+data)?

Hit time?

Miss rate?

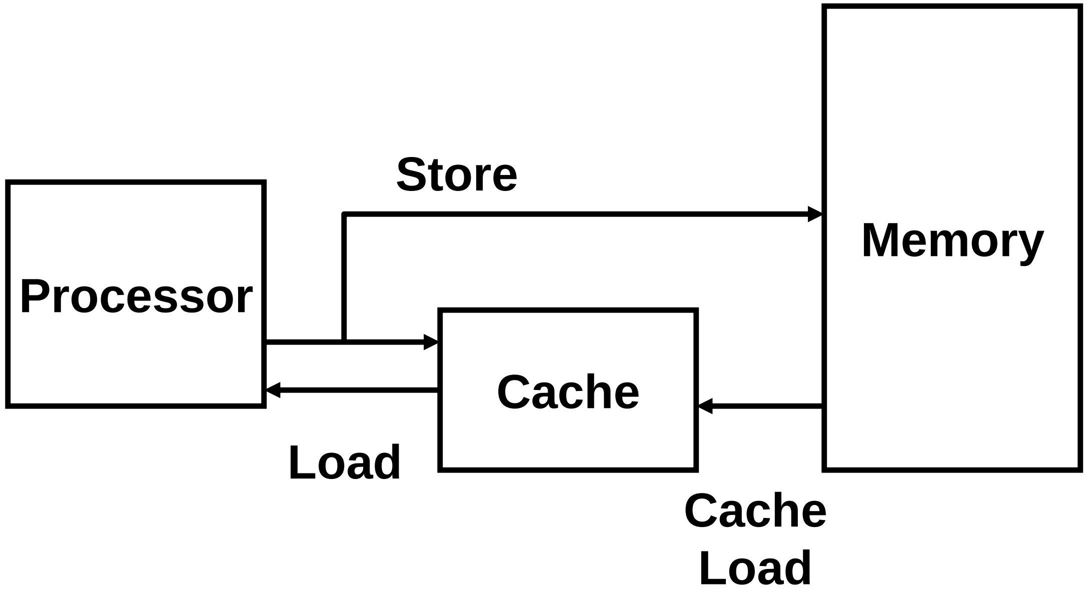
Miss penalty?

Write policy

- What happens when processor writes to the cache?
- Should memory be updated as well?

Write Through:

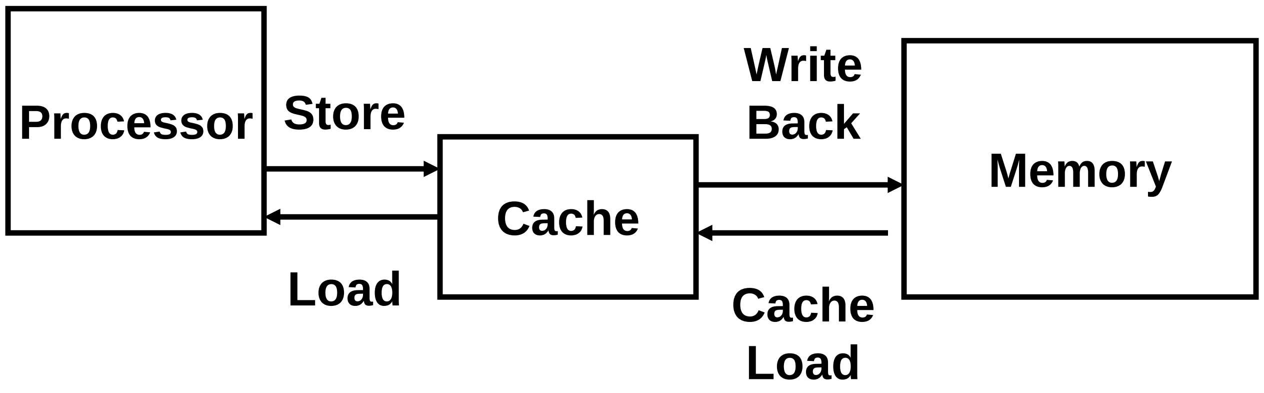
- Store by processor updates cache *and* memory
- Memory always consistent with cache
- Never need to store from cache to memory
- ~2X more loads than stores



Write policy (cont'd)

Write Back:

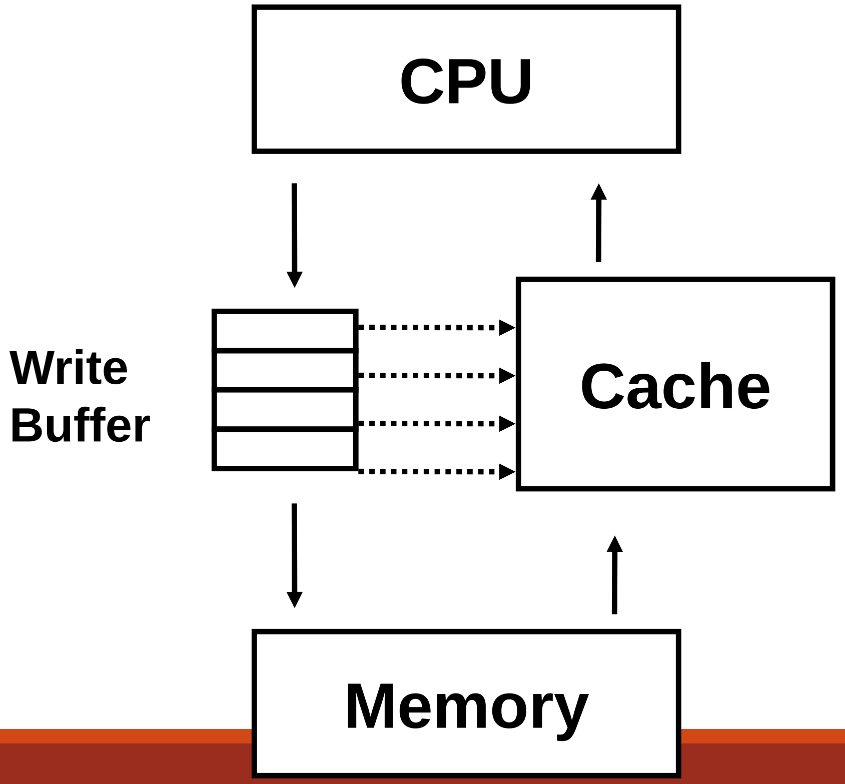
- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
  - Requires "dirty bit" for each line
    - Set when line in cache is modified
    - Indicates that line in memory is stale
- Memory not always consistent with cache



Write buffering

Write Buffer

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

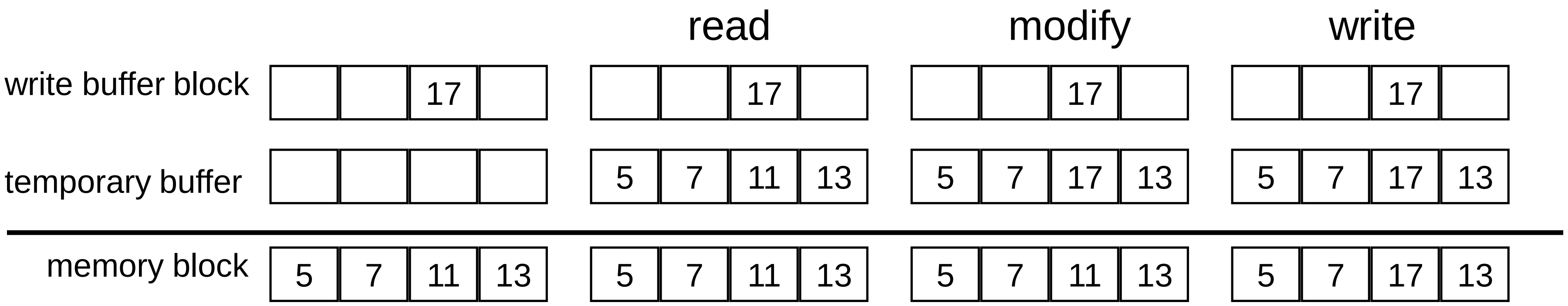


Allocation strategies

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

Write Allocate:

- Block is loaded into cache on a write miss.
- Usually used with write back
- Otherwise, write back 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?

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
- Usually used with write through
    - Memory system directly handles word-level writes

Impact of write policy

Writeback vs write-through

Effect on cache area (tags+data)?

Hit time?

Miss rate?

Miss penalty?

Example: Matrix multiply

Interactions Between Program & Cache

Major Cache Effects to Consider

- Total cache size
  - Try to keep heavily used data in highest level cache
- Block size (sometimes referred to “line size”)
  - Exploit spatial locality

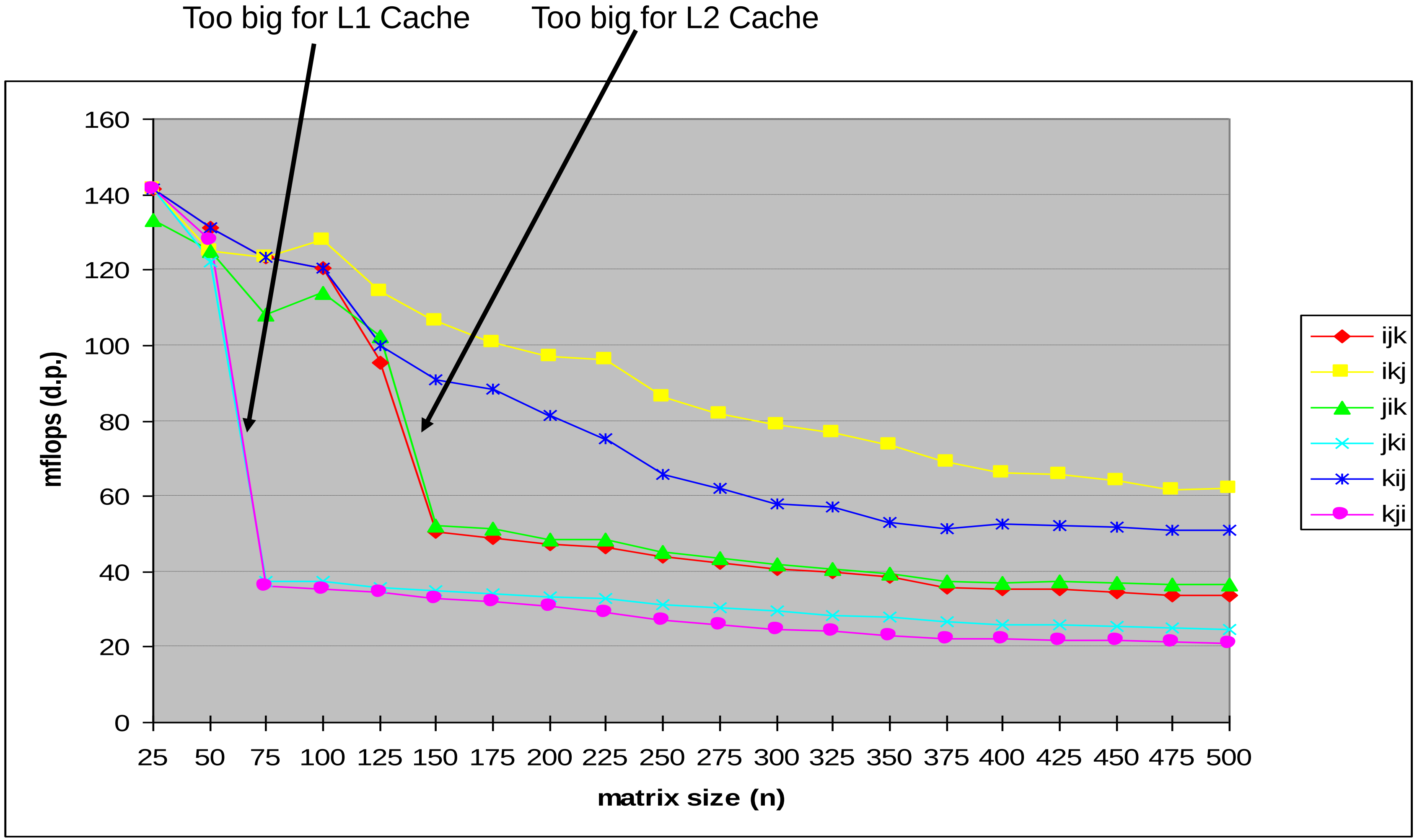
Example Application

- Multiply $n \times n$ matrices
- $O(n^3)$ total operations
- Accesses
  - n reads per source element
  - n values summed per destination, but may be able to hold in register

```
/* 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;
  }
}
```

Variable `sum`
held in register

Matmult Performance (Alpha 21164)



Block Matrix Multiplication

Example n=8, B = 4:

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \text{ X } \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

Key idea: Sub-blocks (i.e., A_{ij}) can be treated just like scalars.

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$ $C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$ $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Blocked Matrix Multiply (bijk)

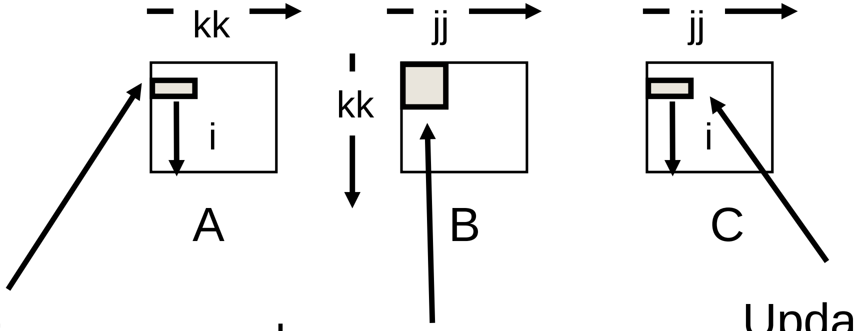
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
      c[i][j] = 0.0;
    }
  }
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
      }
    }
  }
}
```

Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies 1 X bsize sliver of A times bsize X bsize block of B and accumulates into 1 X bsize sliver of C
- Loop over i steps through n row slivers of A & C, using same B

```
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
      sum = 0.0
      for (k=kk; k < min(kk+bsize,n); k++) {
        sum += a[i][k] * b[k][j];
      }
      c[i][j] += sum;
    }
```

Innermost Loop Pair

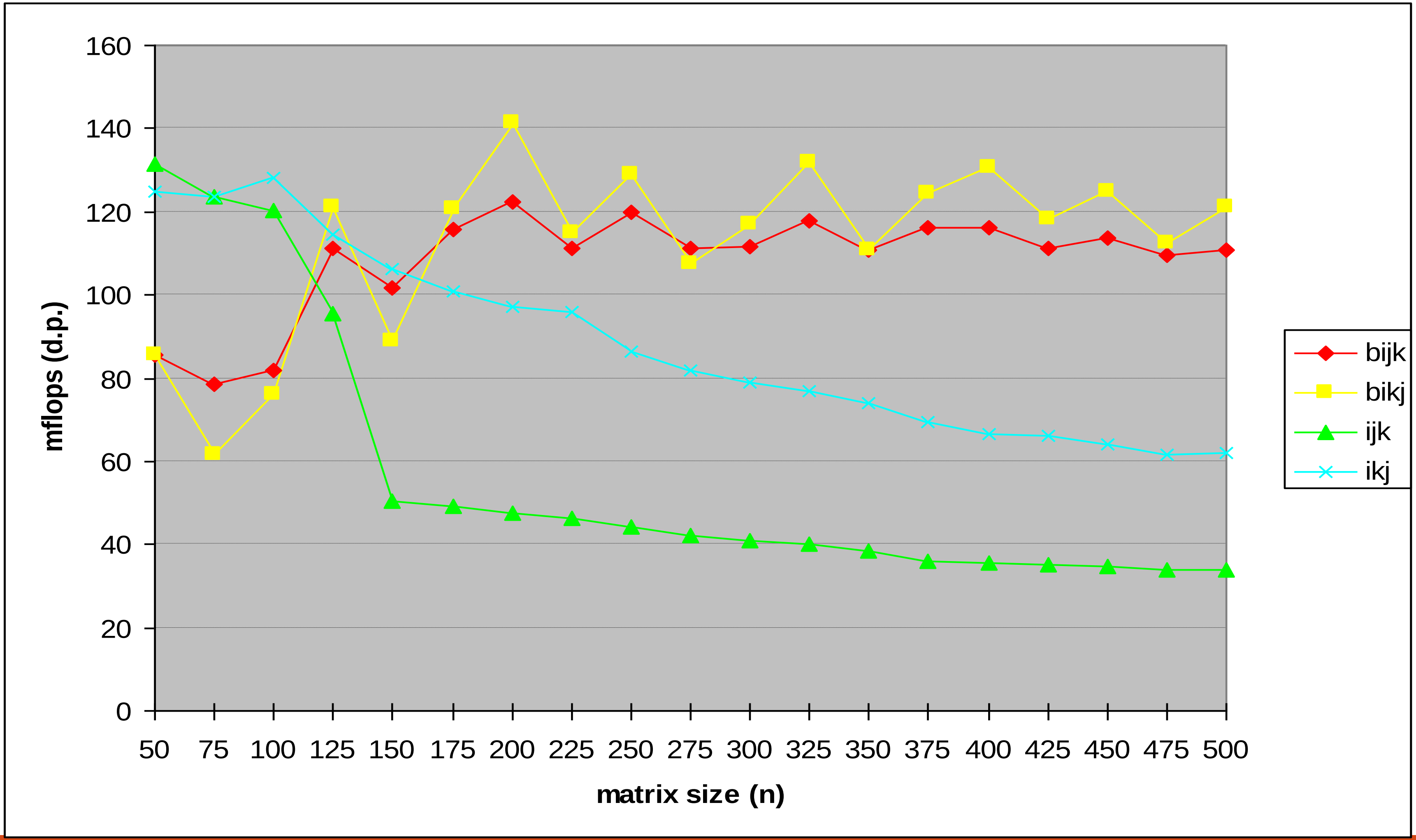


row sliver accessed bsize times

block reused n times in succession

Update successive elements of sliver

Blocked matmult perf (Alpha 21164)



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

Bandwidth matching

Challenge

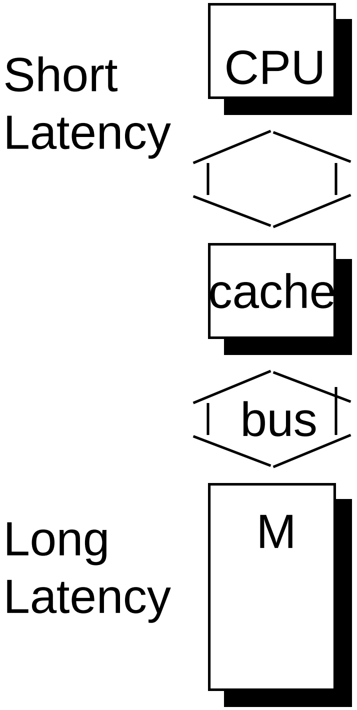
- CPU works with short cycle times
- DRAM (relatively) long cycle times
- *How can we provide enough bandwidth between processor & memory?*

Effect of Caching

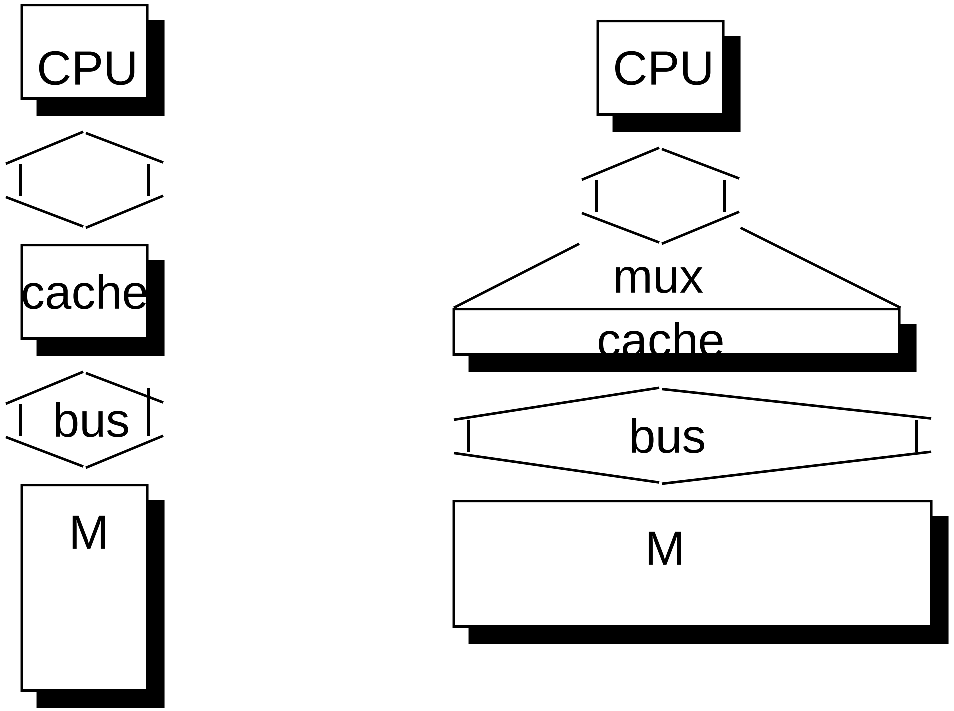
- Caching greatly reduces amount of traffic to main memory
- But, sometimes need to move large amounts of data from memory into cache

Trends

- Need for high bandwidth much greater for multimedia applications
  - Repeated operations on image data
- Recent generation machines greatly improve on predecessors



High Bandwidth Memory Systems



Solution 1
High BW DRAM

Solution 2
Wide path between memory & cache

Example:
Page Mode DRAM
RAMbus

Example: Alpha AXP 21064
256 bit wide bus, L2 cache,
and memory.