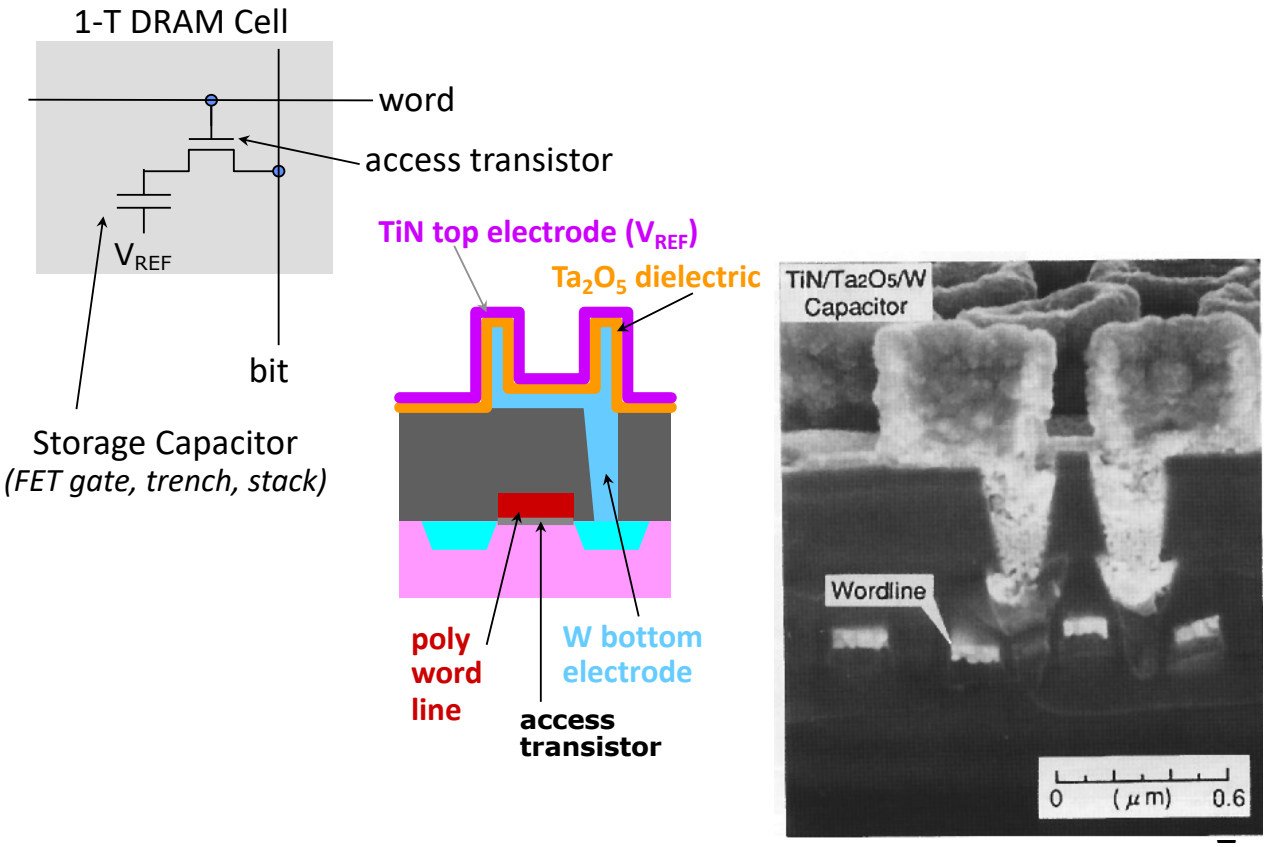
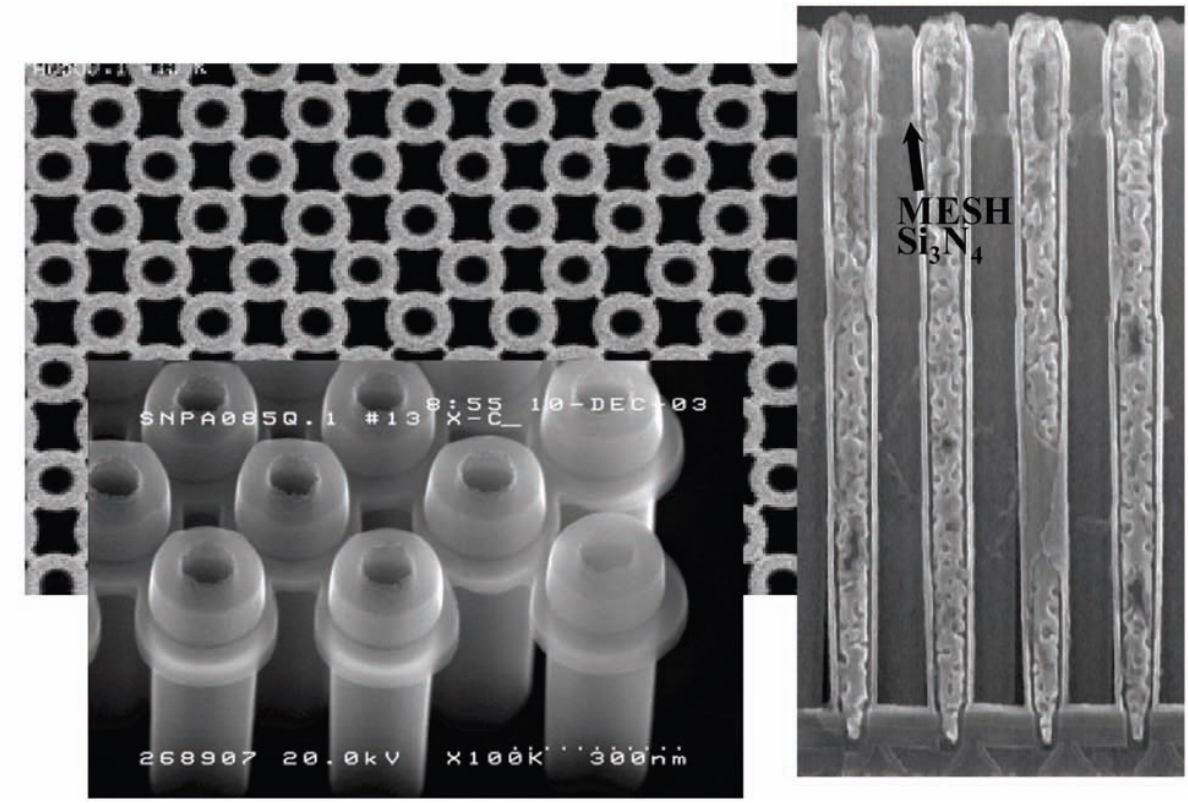
CSC 631: High-Performance Computer Architecture

Fall 2022
Lecture 7: Memory

One-Transistor Dynamic RAM

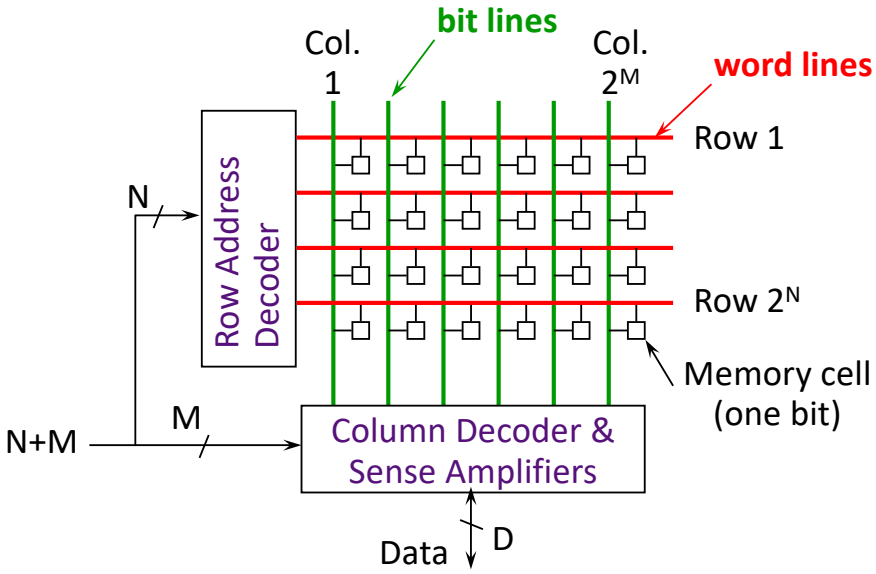


Modern DRAM Structure



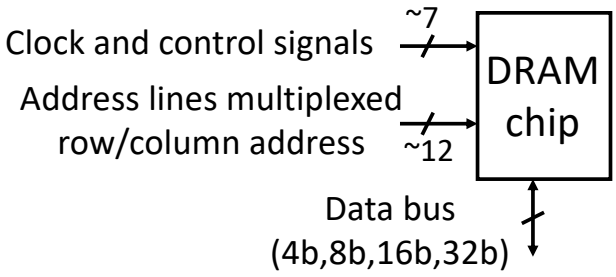
[Samsung, sub-70nm DRAM, 2004]

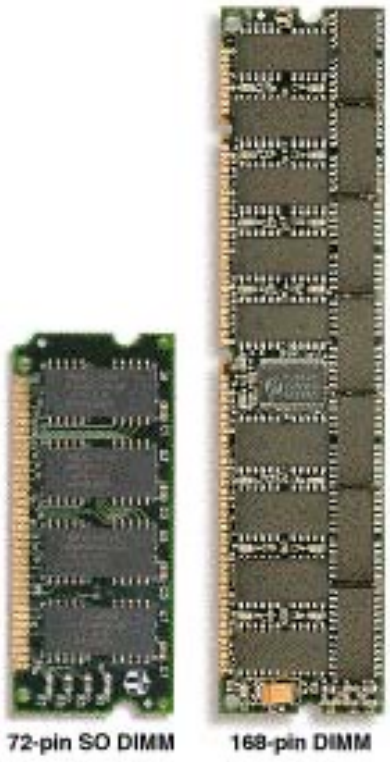
DRAM Architecture



- 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

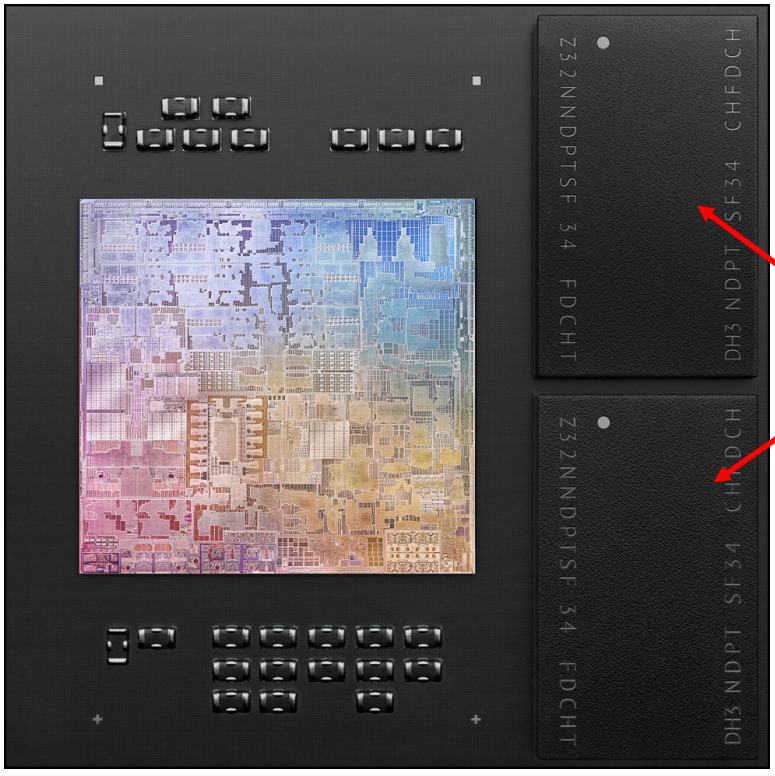
DRAM Packaging
(Laptops/Desktops/Servers)





- DIMM (Dual Inline Memory Module) contains multiple chips with clock/control/address signals connected in parallel (sometimes need buffers to drive signals to all chips)
- Data pins work together to return wide word (e.g., 64-bit data bus using 16x4-bit parts)

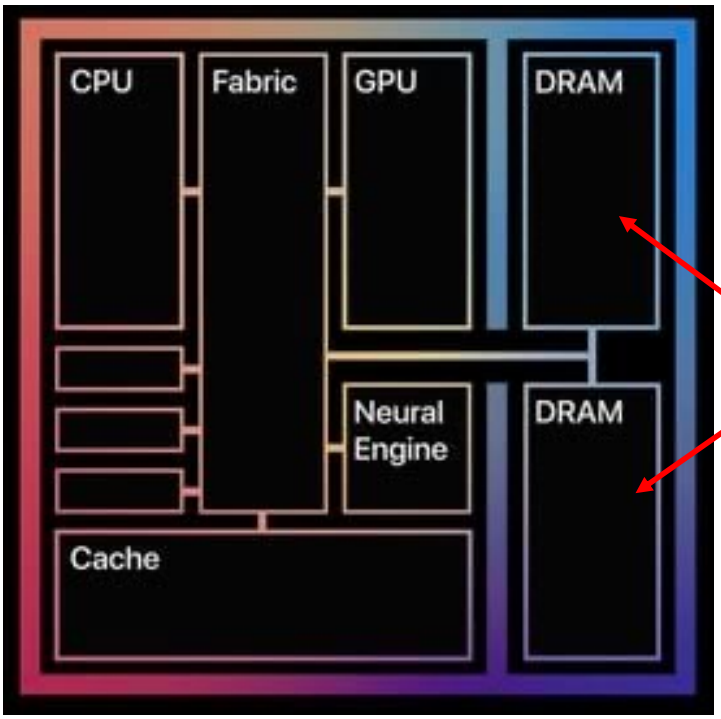
DRAM Packaging, Apple M1



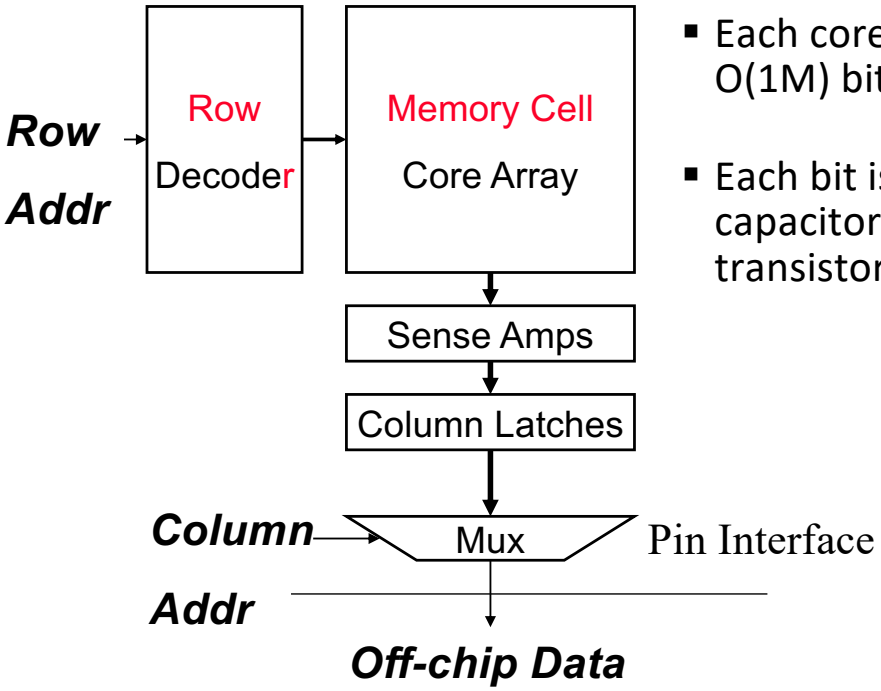
Two DRAM chips on same package as system SoC

- 128b databus, running at 4.2Gb/s
- 68GB/s bandwidth

DRAM Packaging, Apple M2



DRAM Bank Organization

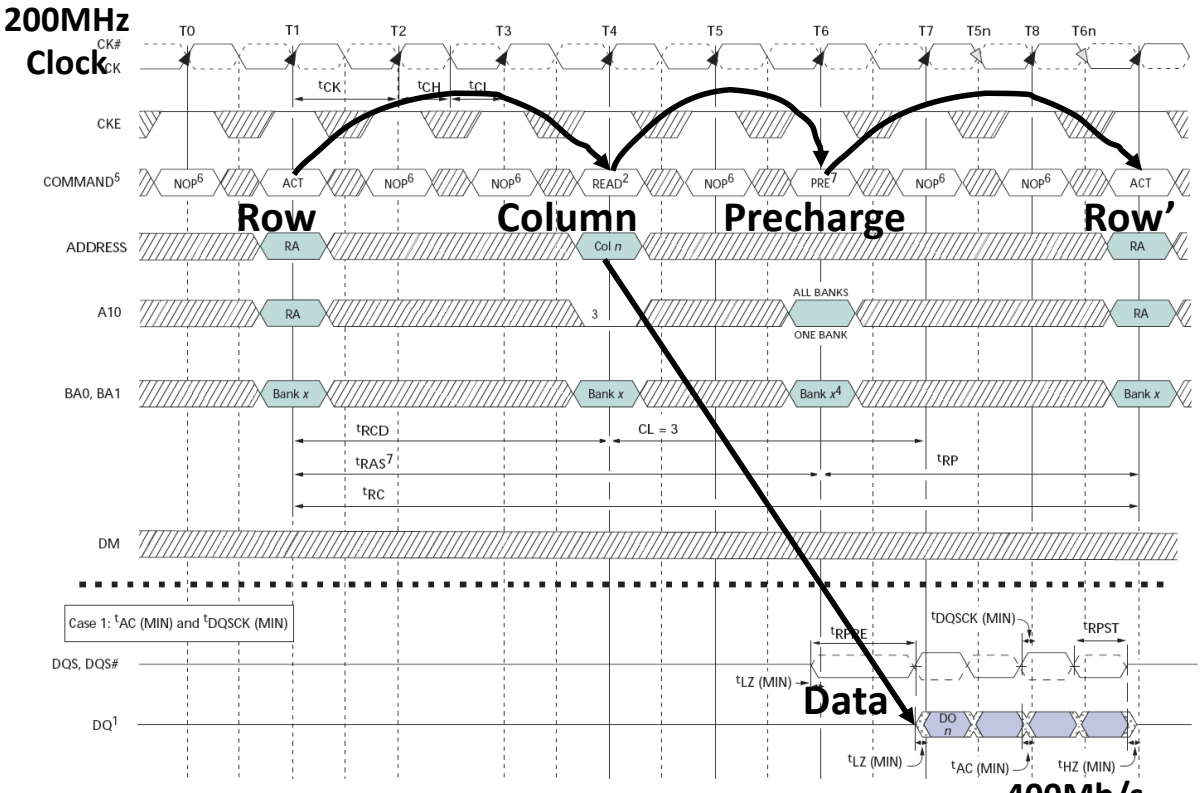


- Each core array has about O(1M) bits
- Each bit is stored in a tiny capacitor, made of one transistor

DRAM Operation

- Three steps in read/write access to a given bank
- 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)
  - 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)
- Precharge
  - charges bit lines to known value, required before next row access
- Each step has a latency of around 15-20ns 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

Double-Data Rate (DDR2) DRAM



[Micron, 256Mb DDR2 SDRAM datasheet]

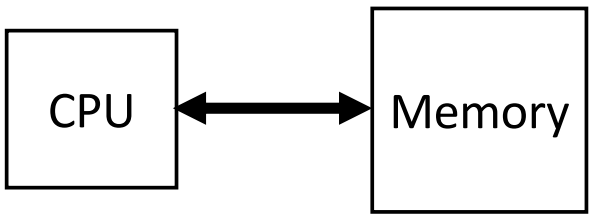
Global Memory (DRAM) Bandwidth

Ideal

Reality

11

CPU-Memory Bottleneck



Performance of high-speed computers is usually limited by memory bandwidth & latency

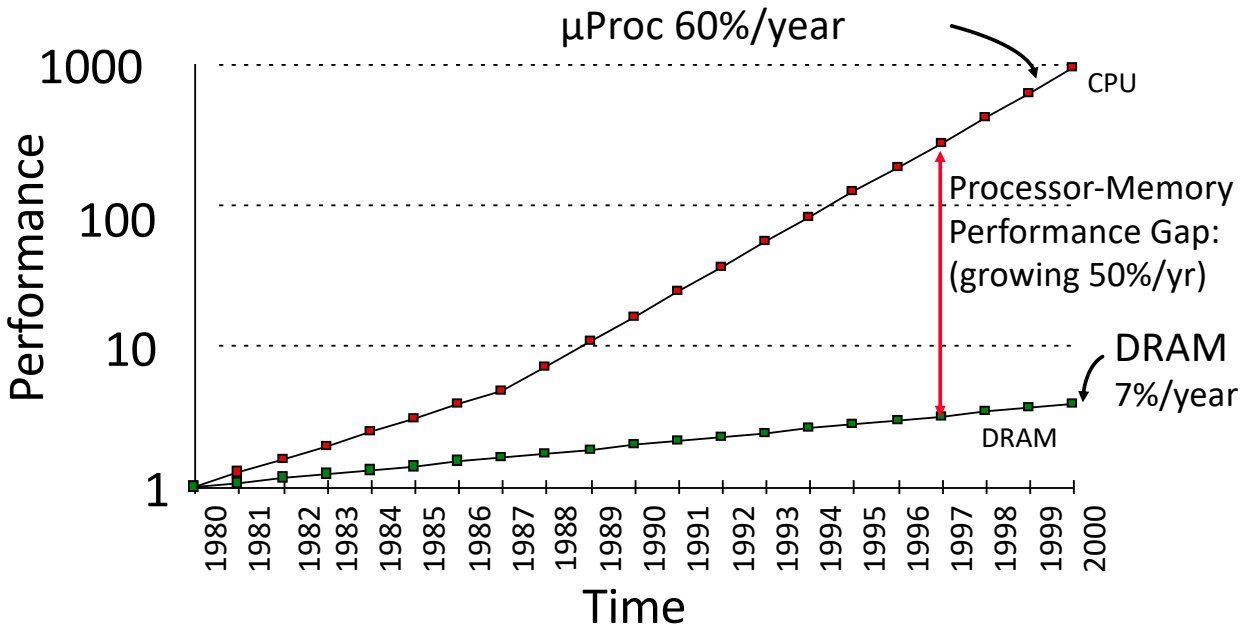
- Latency (time for a single access)
  - Memory access time >> Processor cycle time
- Bandwidth (number of accesses per unit time)

  if fraction m of instructions access memory

  ⇒ 1+m memory references / instruction

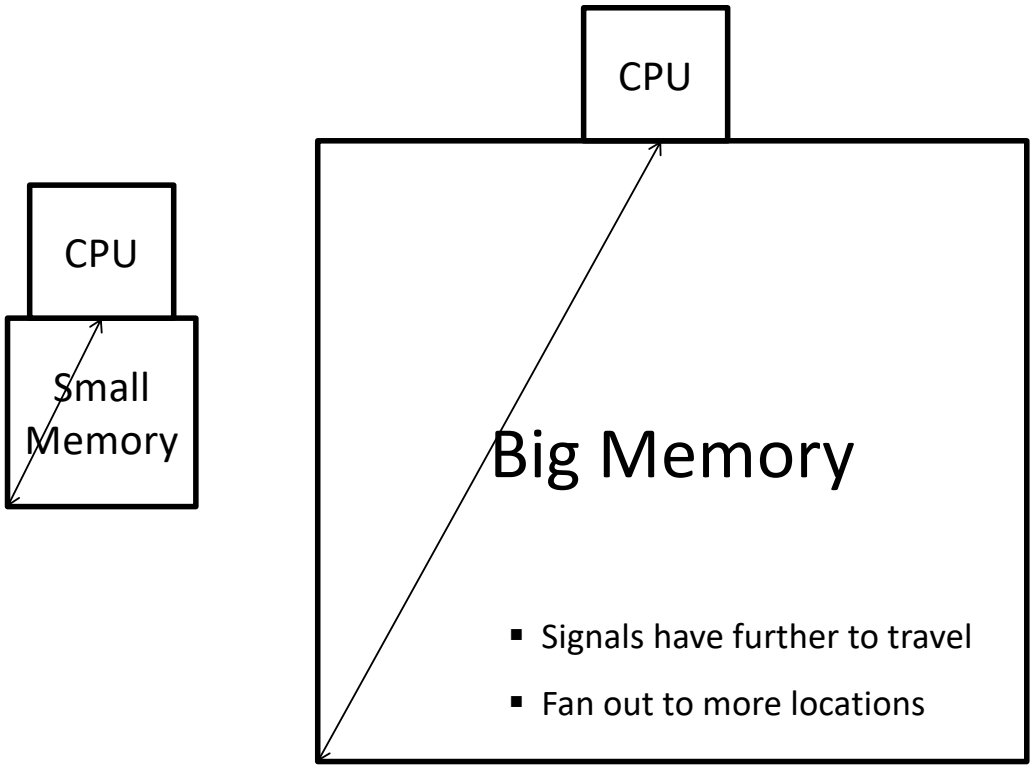
  ⇒ CPI = 1 requires 1+m memory refs / cycle (assuming RISC-V ISA)
- *Also, Occupancy (time a memory bank is busy with one request)*

Processor-DRAM Gap (latency)

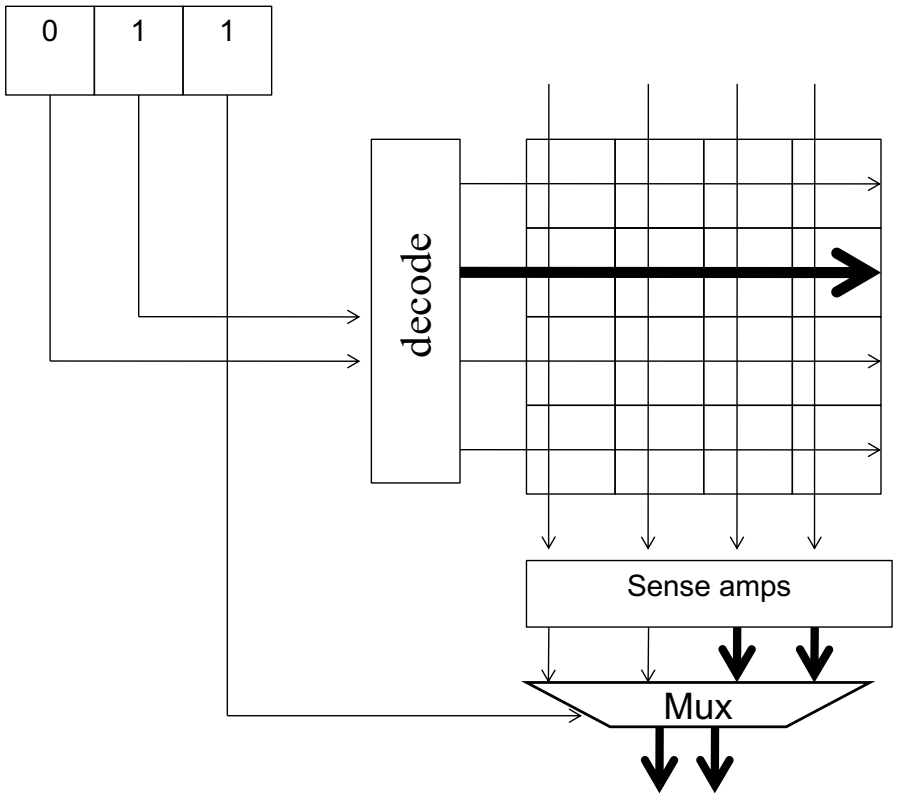


Four-issue 3GHz superscalar accessing 100ns DRAM could execute 1,200 instructions during time for one memory access!

Physical Size Affects Latency

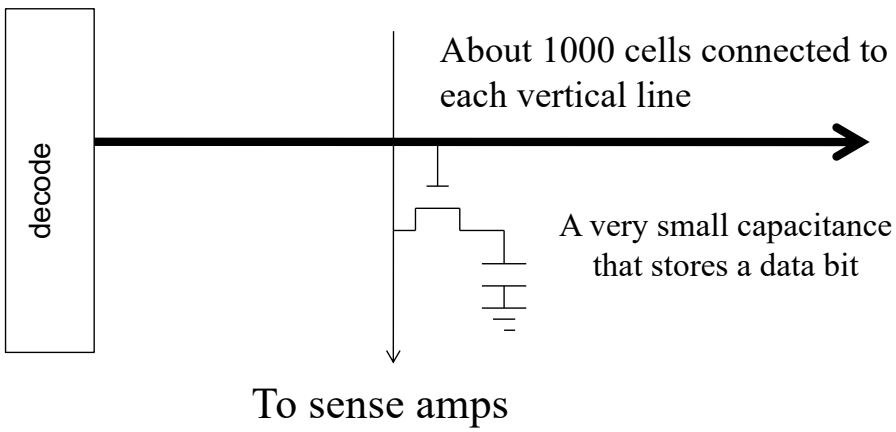


A very small (8x2 bit) DRAM Bank

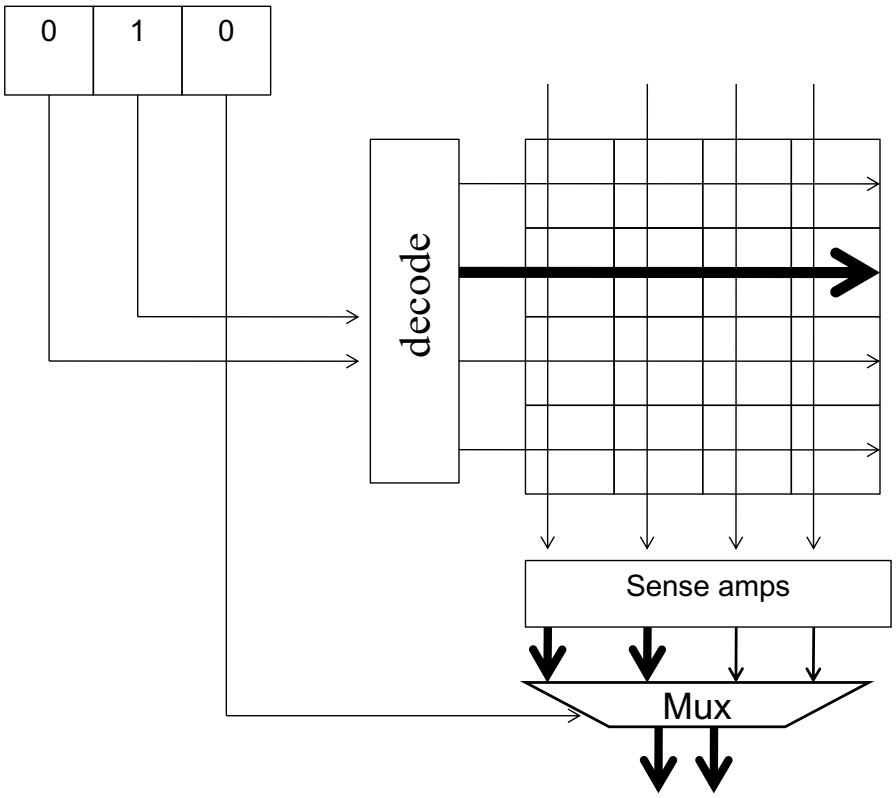


DRAM core arrays are slow.

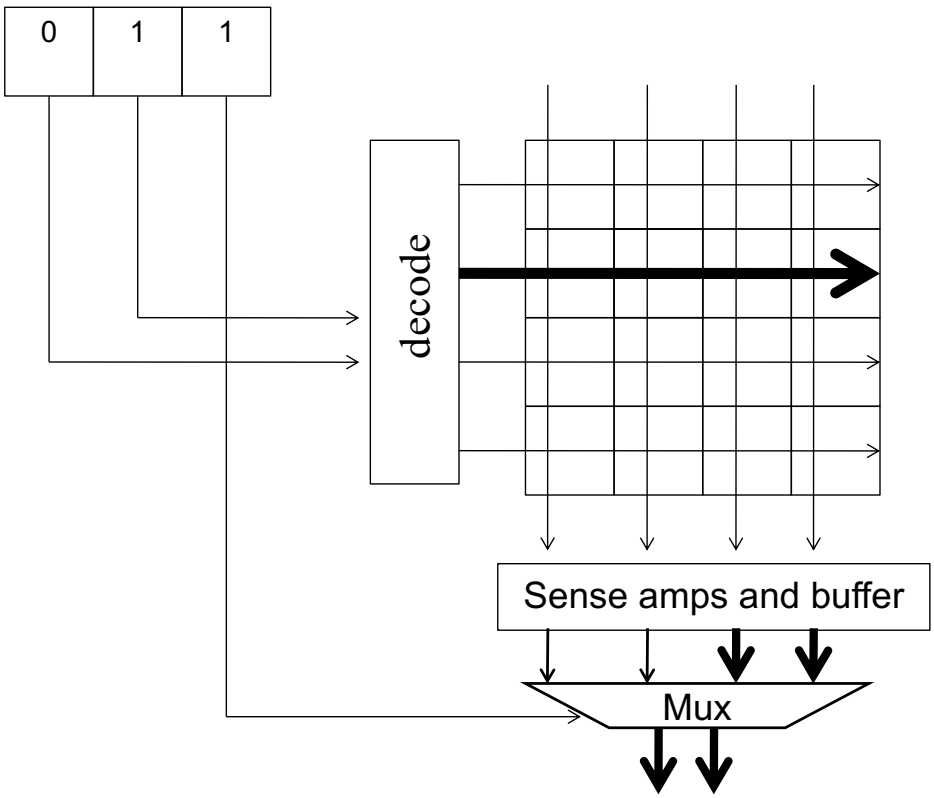
- Reading from a cell in the core array is a very slow process
  - DDR: Core speed = ½ interface speed
  - DDR2/GDDR3: Core speed = ¼ interface speed
  - DDR3/GDDR4: Core speed = ⅛ interface speed
  - … likely to be worse in the future



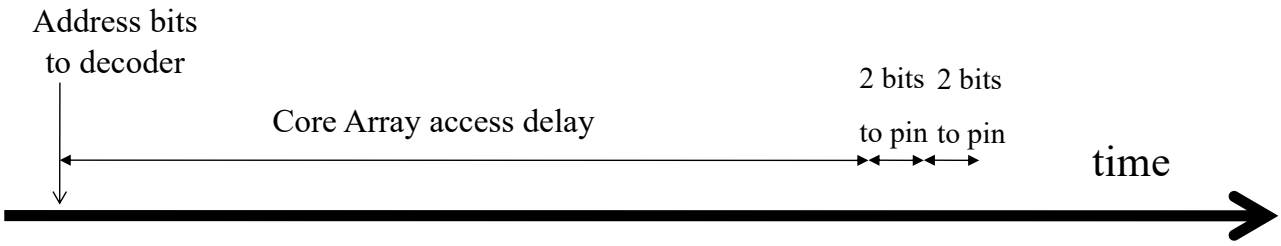
DRAM Bursting (burst size = 4 bits)



DRAM Bursting (cont.)
second part of the burst



DRAM Bursting for the 8x2 Bank



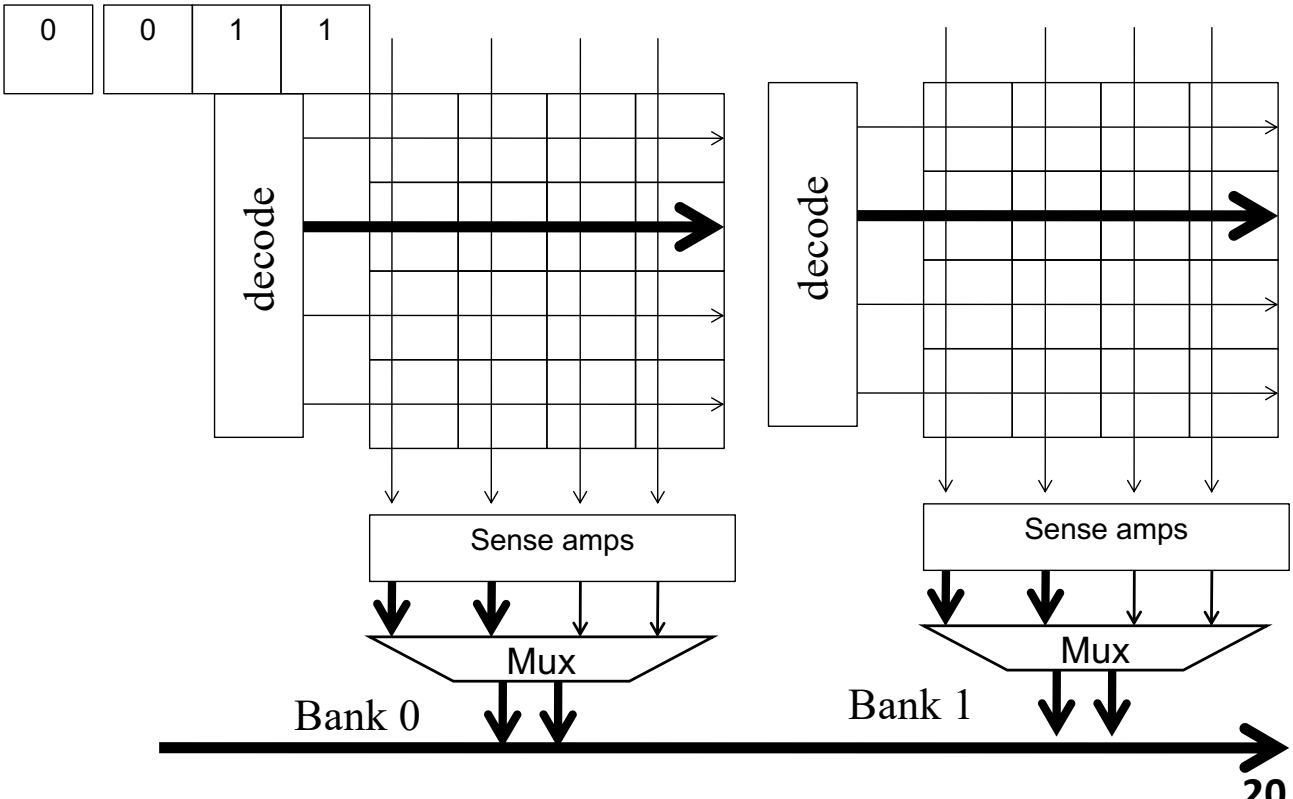
Non-burst timing

Burst timing

Modern DRAM systems are designed to be always accessed in burst mode. Burst bytes are transferred but discarded when accesses are not to sequential locations.

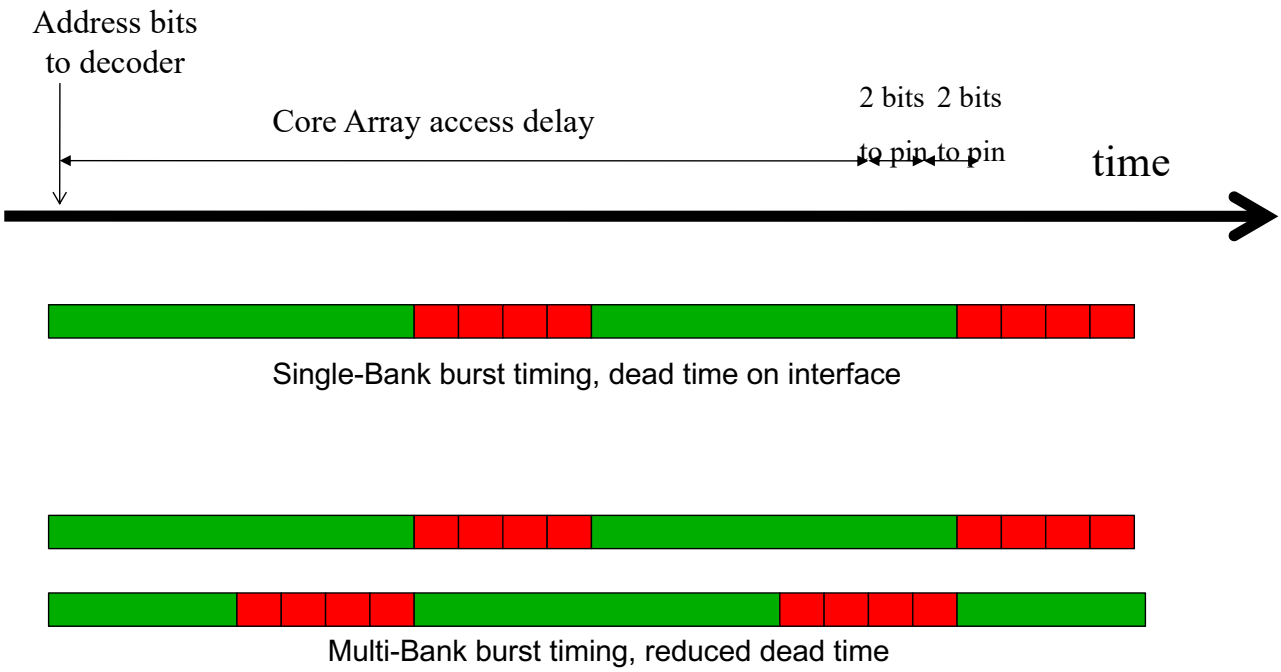
19

Multiple DRAM Banks

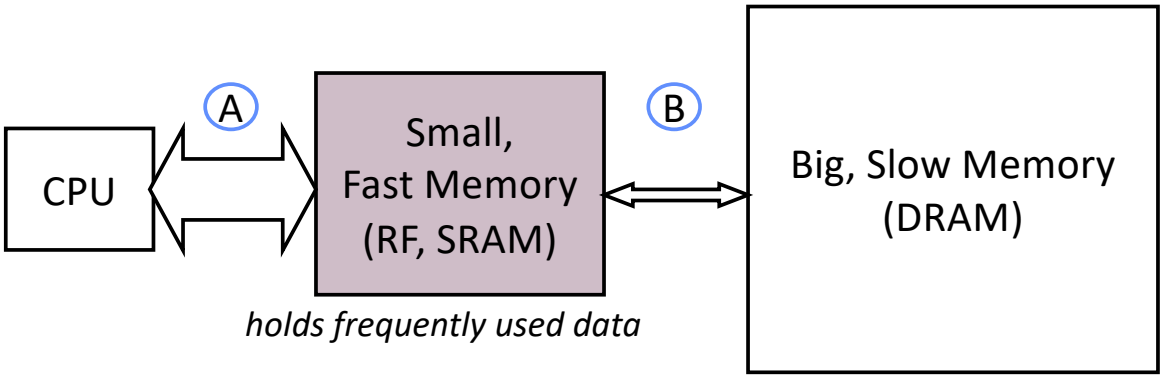


20

DRAM Bursting for the 8x2 Bank



Memory Hierarchy



- *capacity*: Register << SRAM << DRAM
- *latency*: Register << SRAM << DRAM
- *bandwidth:* on-chip >> off-chip

On a data access:

if data $\in$ fast memory $\Rightarrow$ low latency access *(SRAM)*

if data $\notin$ fast memory $\Rightarrow$ high latency access *(DRAM)*

Management of Memory Hierarchy

- 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., don't cache or prefetch*

Two predictable properties of memory references:

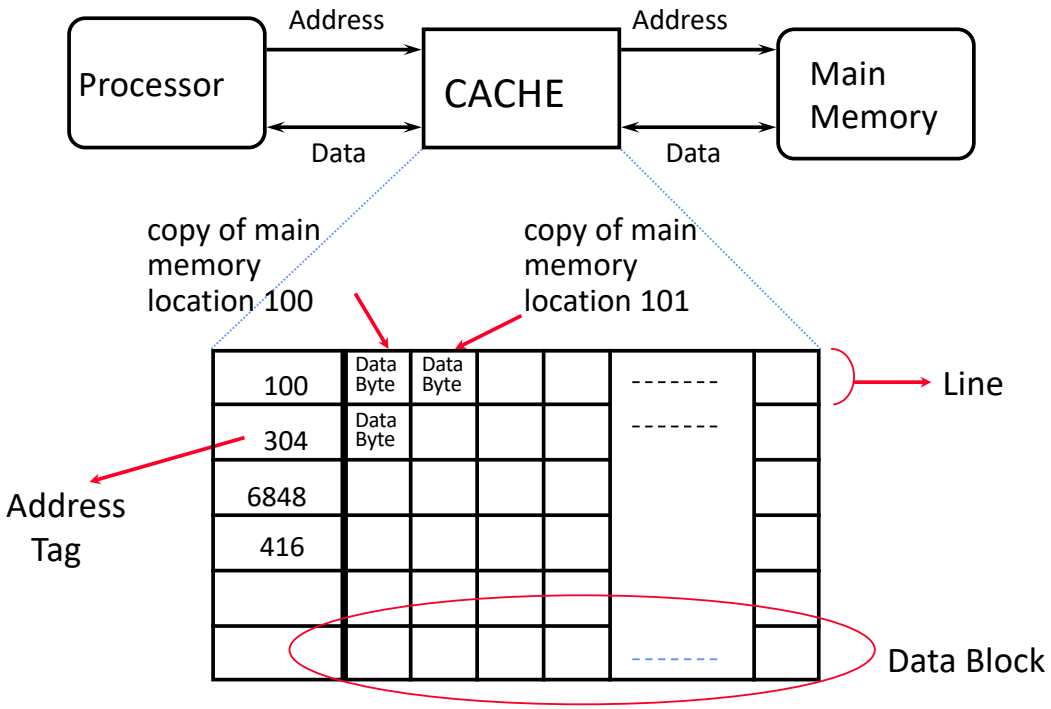
- **Temporal Locality**: If a location is referenced it is likely to be referenced again in the near future.

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

Caches exploit both types of predictability:

- Exploit temporal locality by remembering the contents of recently accessed locations.
- Exploit spatial locality by fetching blocks of data around recently accessed locations.

Inside a Cache



A Couple of Notes First…

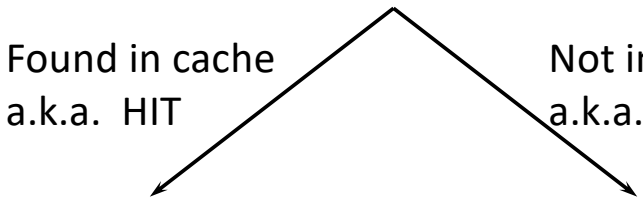
- Data is stored in a cache based on a mapping process
  - Could be *i mod* 2^k where *i* is the memory address and 2^k is the number of blocks
  - Could be least significant bits

- How do we find the data
  - Use the address in the cache plus a tag
  - Concatenate the block tag with the block index

- Use a valid bit for each block to indicate whether the data in the block is valid

Cache Algorithm (Read)

Look at Processor Address, search cache tags to find match. Then either



Return copy of data from cache

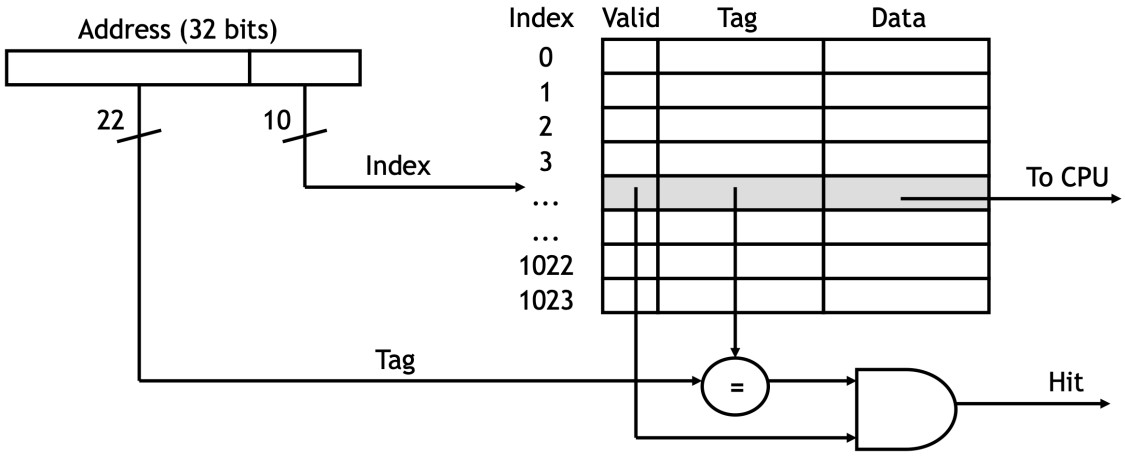
Read block of data from Main Memory

Wait …

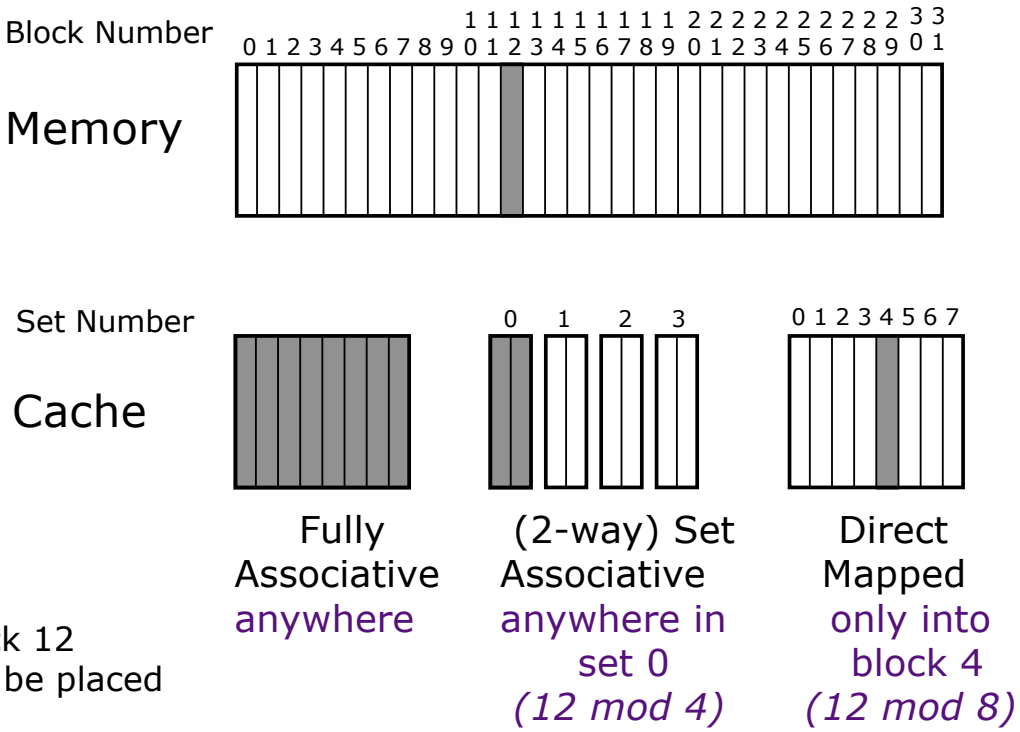
Return data to processor and update cache

Q: Which line do we replace?

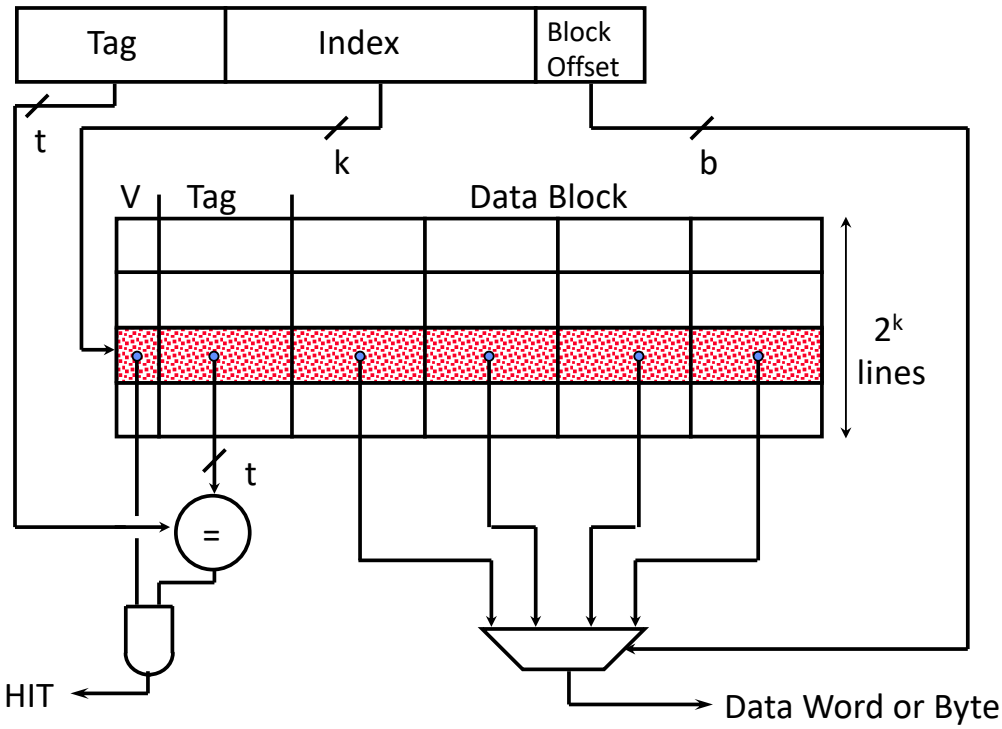
Cache Hit



Placement Policy

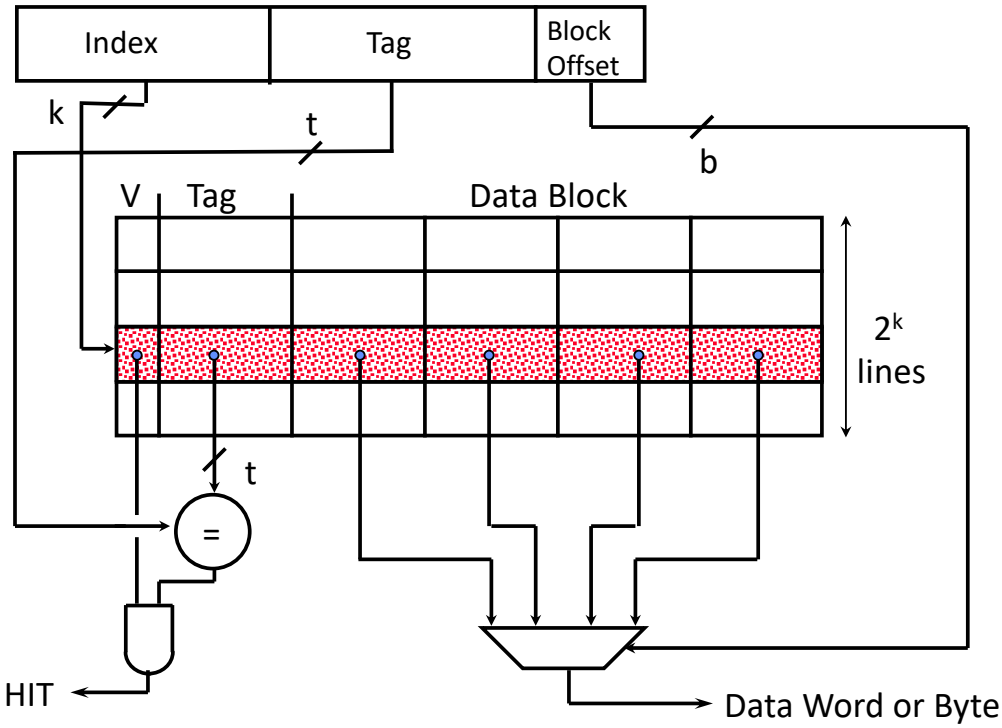


Direct-Mapped Cache

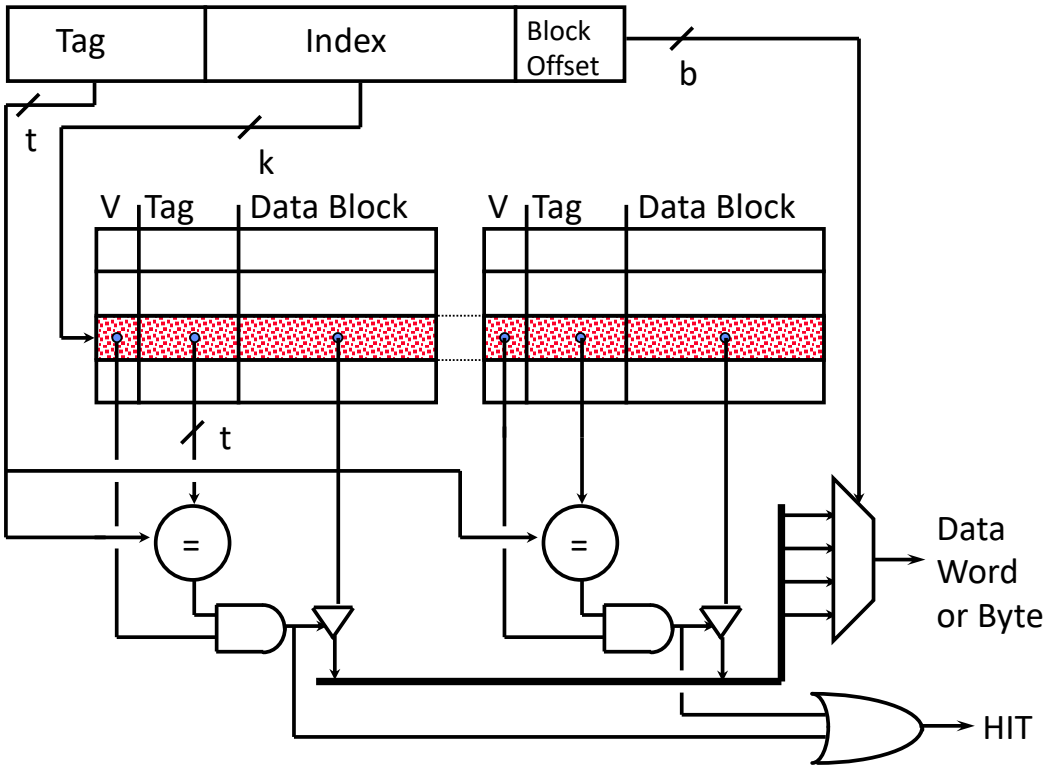


Direct Map Address Selection

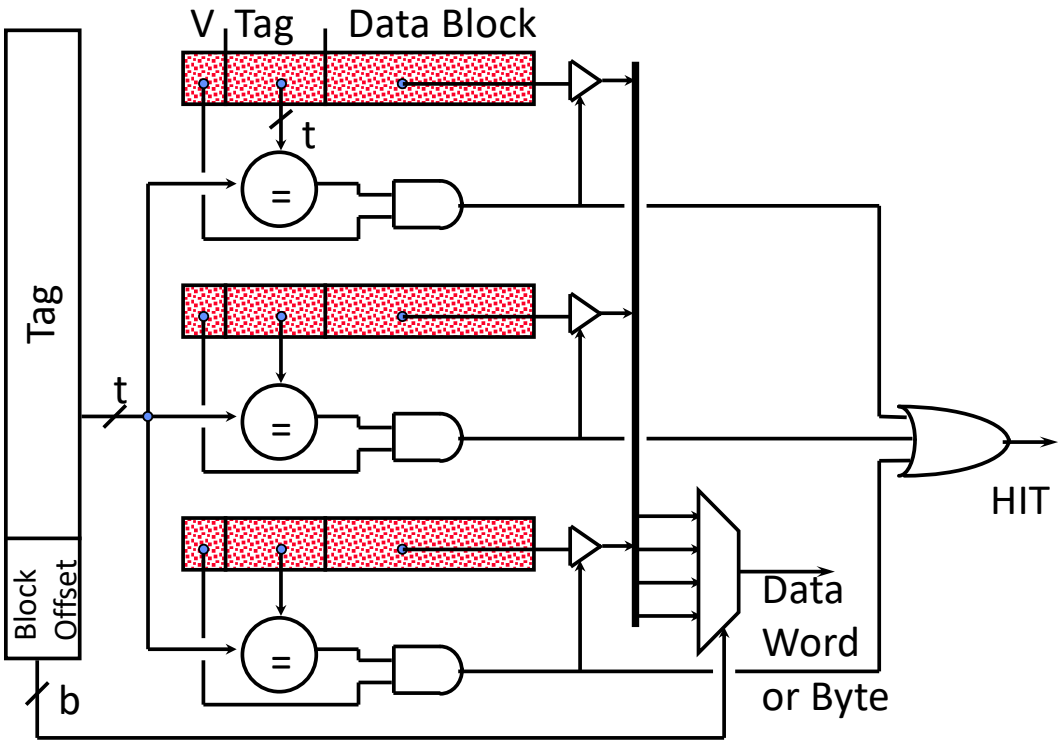
higher-order vs. lower-order address bits



2-Way Set-Associative Cache



Fully Associative Cache



Replacement Policy

In an associative cache, which block from a set should be evicted when the set becomes full?

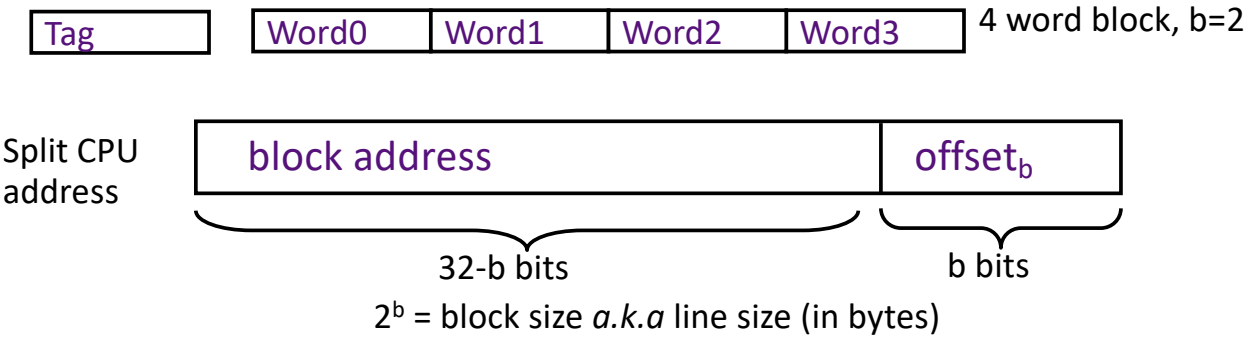
- Random
- Least-Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree often used for 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches
- Not-Most-Recently Used (NMRU)
  - FIFO with exception for most-recently used block or blocks

This is a second-order effect. Why?

Replacement only happens on misses

Block Size and Spatial Locality

Block is unit of transfer between the cache and memory



Larger block size has distinct hardware advantages

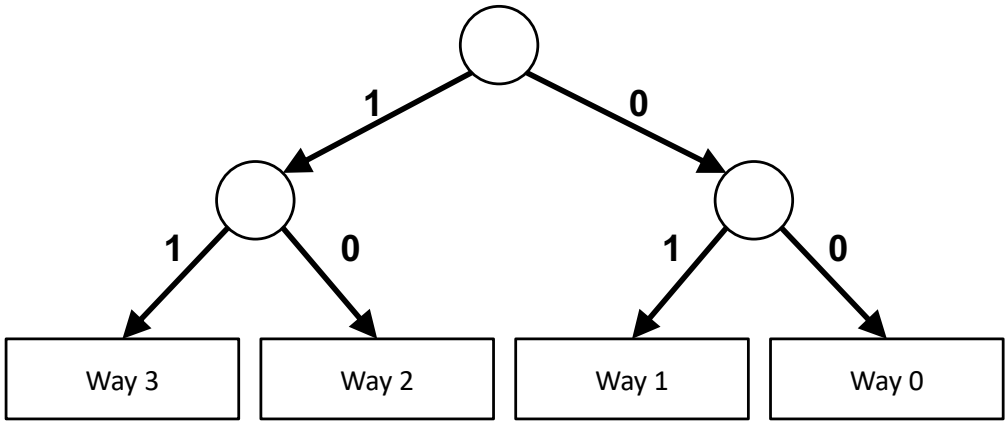
- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

What are the disadvantages of increasing block size?

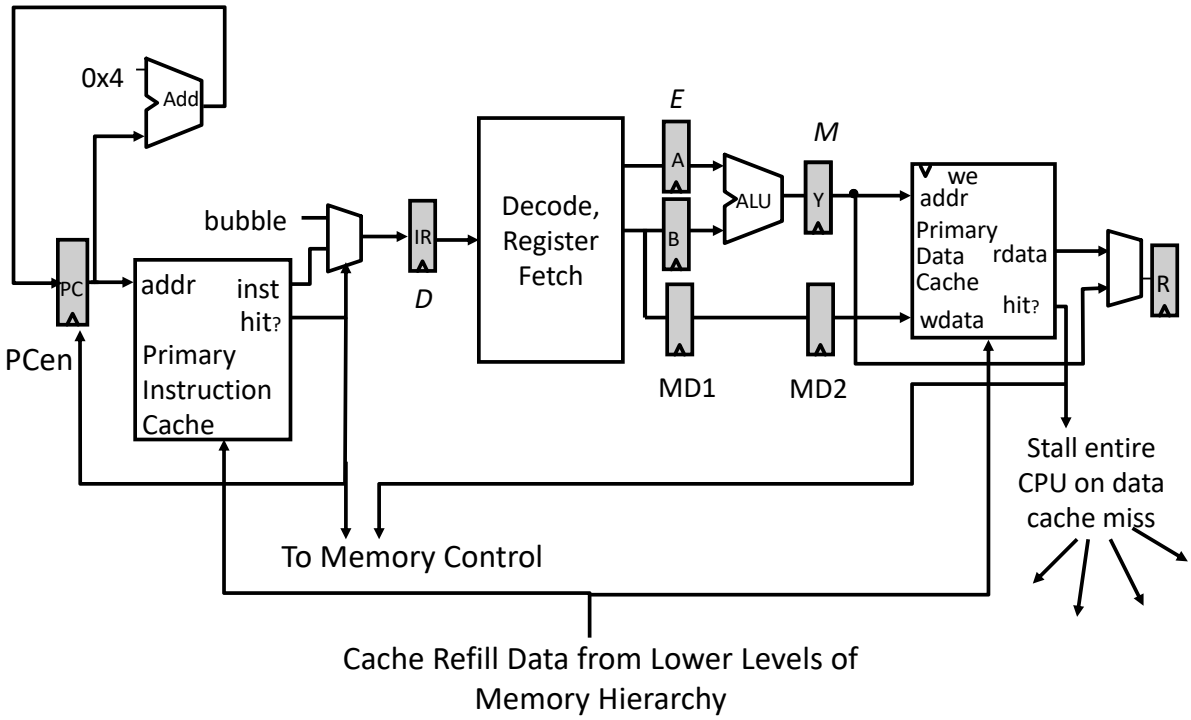
Fewer blocks => more conflicts. Can waste bandwidth.

Pseudo-LRU Binary Tree

- For 2-way cache, on a hit, single LRU bit is set to point to other way
- For 4-way cache, need 3 bits of state. On cache hit, on path down tree, set all bits to point to other half. On miss, bits say which way to replace



CPU-Cache Interaction
(5-stage pipeline)



Improving Cache Performance

Average memory access time (AMAT) =
Hit time + Miss rate x Miss penalty

To improve performance:

- reduce the hit time
- reduce the miss rate
- reduce the miss penalty

What is best cache design for 5-stage pipeline?

Biggest cache that doesn't increase hit time past 1 cycle (approx 8-32KB in modern technology)

[design issues more complex with deeper pipelines and/or out-of-order superscalar processors]

Cache Misses

- The first request to a cache block is called a *compulsory miss*, because the block must be read from memory regardless of the cache design
- *Capacity misses* occur when the cache is too small to hold all concurrently used data.
- *Conflict misses* are caused when several addresses map to the same set and evict blocks that are still needed.

Cache misses can be reduced by changing capacity, block size, and/or associativity

Causes of Cache Misses: The 3 C's

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

– *misses that would occur even with infinite cache*

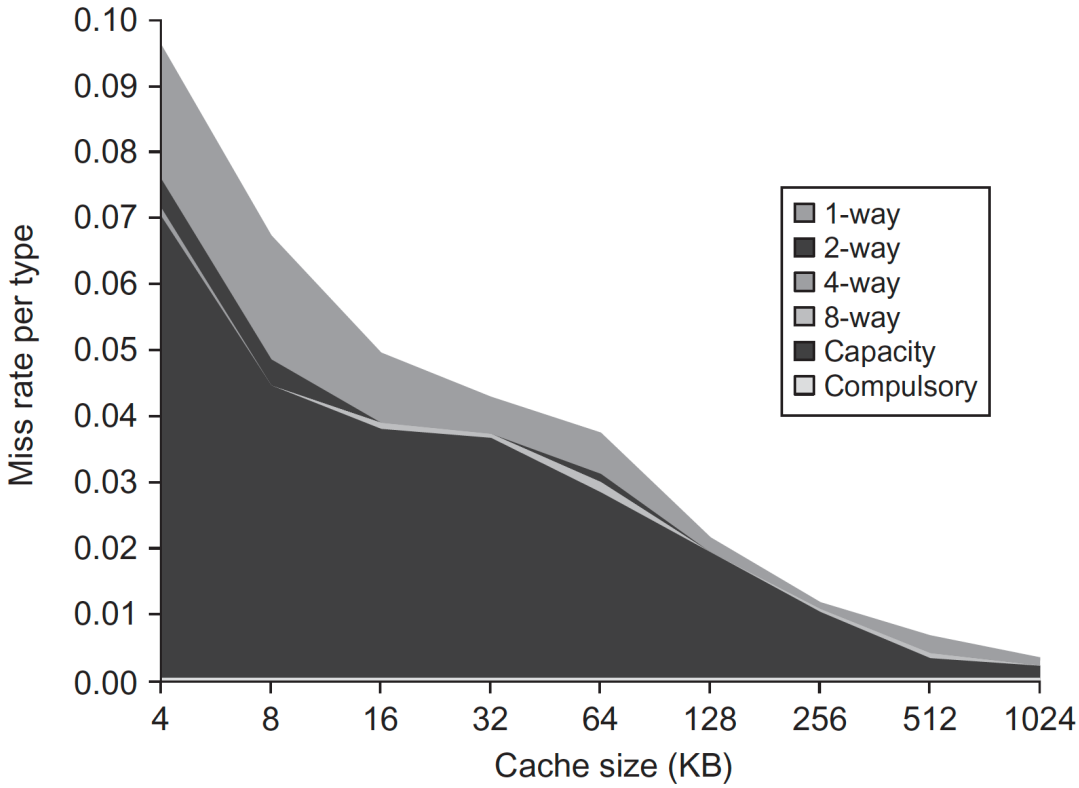
Capacity: cache is too small to hold all data needed by the program

– *misses that would occur even under perfect replacement policy*

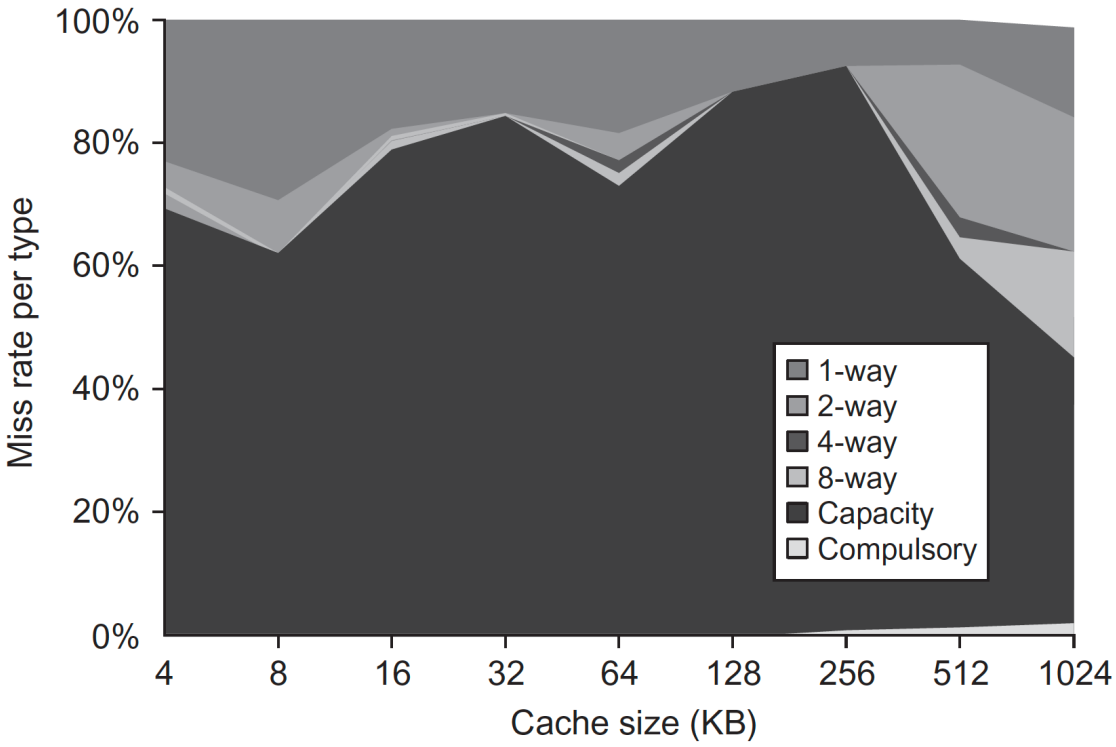
Conflict: misses that occur because of collisions due to line-placement strategy

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

Total miss rate for each size cache: actual data cache miss rates



Total miss rate for each size cache: percentage per category



Effect of Cache Parameters on Performance

- Larger cache size
  + reduces capacity and conflict misses
  - hit time will increase

- Higher associativity
  + reduces conflict misses
  - may increase hit time

- Larger line size
  + reduces compulsory and capacity (reload) misses
  - increases conflict misses and miss penalty

Effect of Cache Parameters on Performance

- Fully associative placement avoids all conflict misses but expensive in hardware and may slow the processor clock rate leading to lower overall performance

- The three C's also ignore replacement policy

Definition: thrashing is when a program spends a significant percentage of the time in moving data between two levels in the hierarchy

Cache Optimization

- Larger Block Size to Reduce Miss Rate
  - Reduce Miss Rate by taking advantage of spatial locality
  - Reduce compulsory misses
  - Reduce the number of blocks in the cache and thus increasing the miss penalty

- Larger Caches to Reduce Miss Rate
  - Longer hit time and higher cost and power

- Higher Associativity to Reduce Miss Rate
  - 8-way set associative is for practical purposes as effective in reducing misses as fully associative
  - 2:1 cache rule of thumb: a direct-mapped cache of size N has about the same miss rate as a two-way set associative cache of size N/2

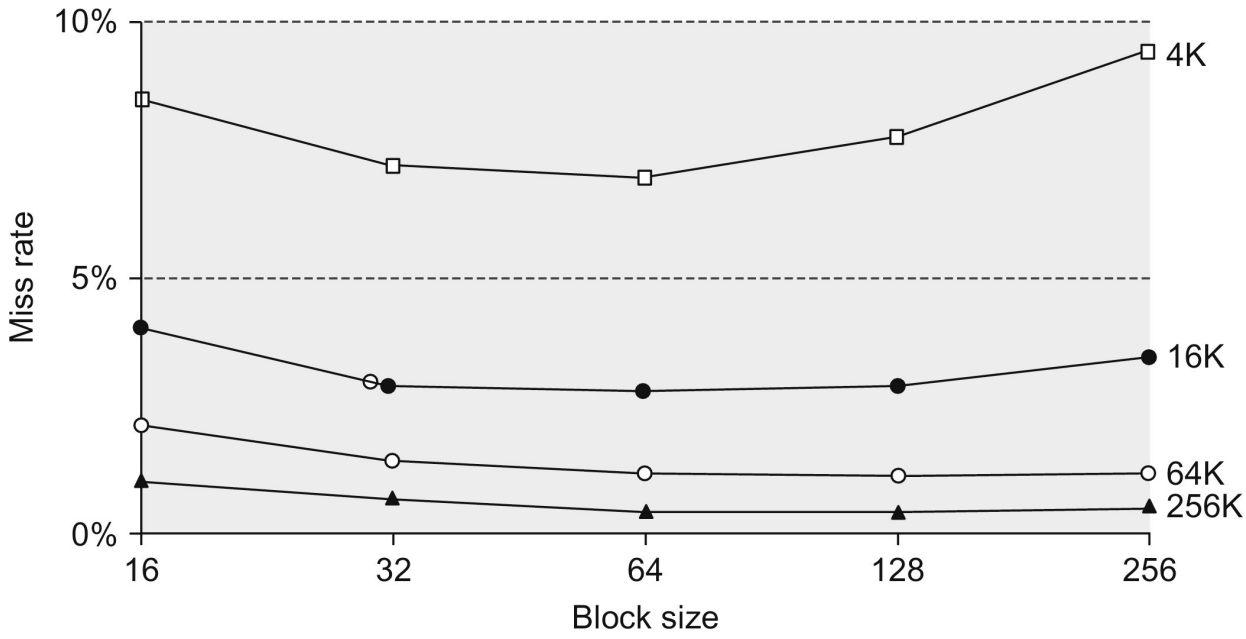
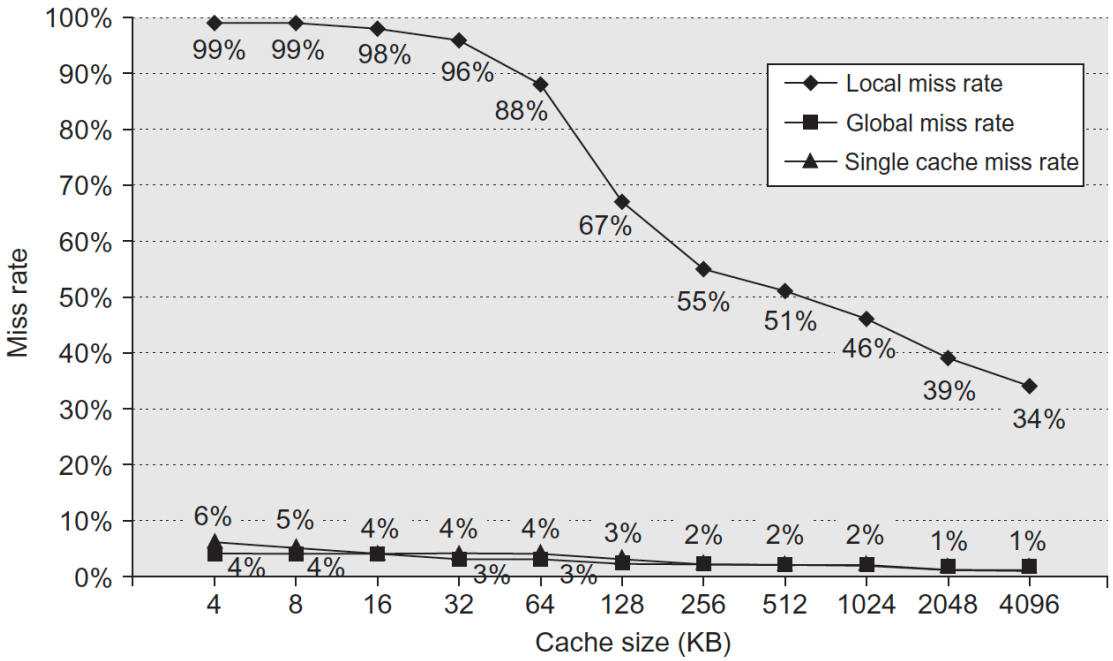


Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 (Gee et al. 1993).

Cache Optimization

- Multilevel Caches to Reduce Miss Penalty
  - First-level cache is small enough to match the clock cycle time of the processor
  - The second-level is large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty
  - Impacts AMAT and cost of miss penalty
  - Two new parameters:
    - Local miss rate
    - Global miss rate

Cache Optimization



Cache Optimization

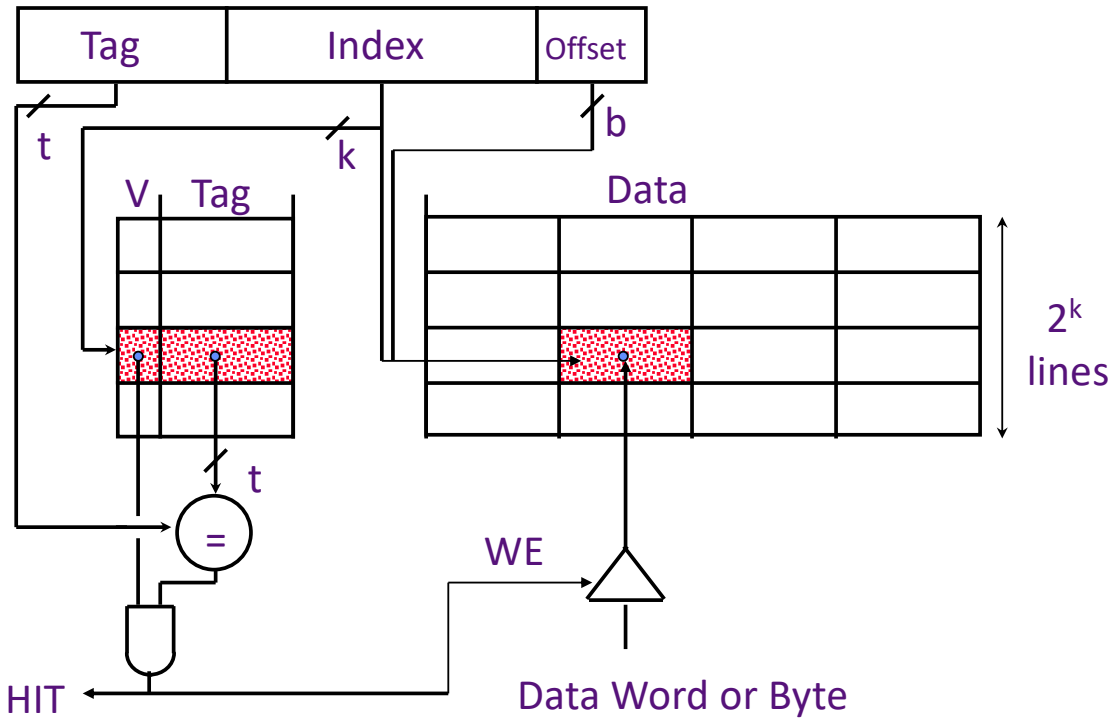
- Give priority to *Read Misses* over *Writes* to reduce *Miss Penalty*
  - Serves reads before writes have been completed
  - Use a Write Buffer
    - Complicates memory accesses because they might hold the updated value of a location needed on a read miss

- Avoid address translation during indexing of the cache to reduce Hit Time
  - Problem with page protection, process switching, and and user programs using two different virtual addresses for the same physical address

Write Policy Choices

- Cache hit:
  - ***write-through***: write both cache & memory
    - Generally higher traffic but simpler pipeline & cache design
  - ***write-back***: write cache only, memory is written only when the entry is evicted
    - A dirty bit per line further reduces write-back traffic
    - Must handle 0, 1, or 2 accesses to memory for each load/store
- Cache miss:
  - ***no-write-allocate***: only write to main memory
  - ***write-allocate*** (aka fetch-on-write): fetch into cache
- Common combinations:
  - write-through and no-write-allocate
  - write-back with write-allocate

Write Performance

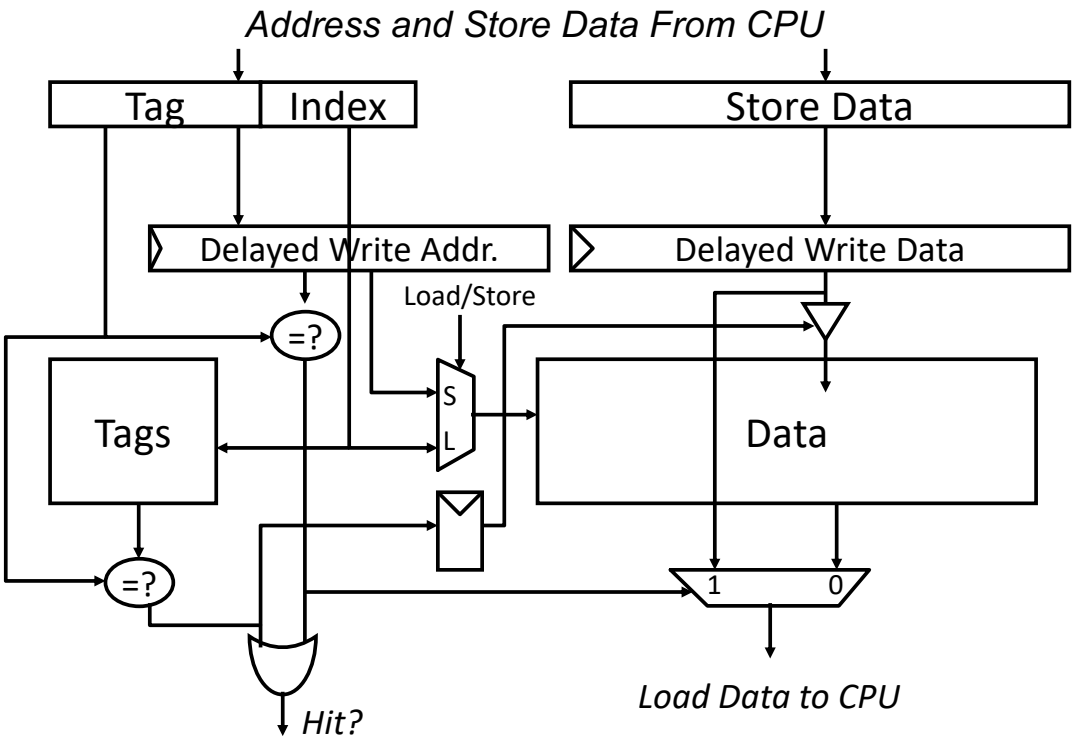


Reducing Write Hit Time

- **Problem**
  - Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

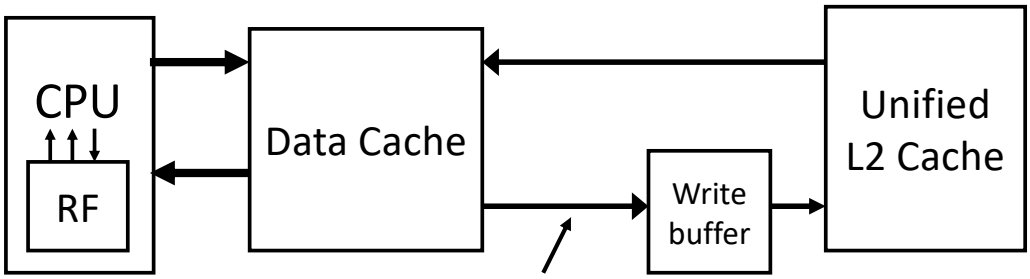
- **Solutions:**
  - Design data RAM that can perform read and write in one cycle, restore old value after tag miss
  - Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check
  - Fully-associative (CAM Tag) caches: Word line only enabled if hit

Pipelining Cache Writes



Data from a store hit is written into data portion of cache during tag access of subsequent store

Write Buffer to Reduce Read Miss Penalty



Evicted dirty lines for write-back cache
OR
All writes in write-through cache

- Processor is not stalled on writes, and read misses can go ahead of write to main memory

- **Problem**
  - Write buffer may hold updated value of location needed by a read miss
- **Simple solution:**
  - on a read miss, wait for the write buffer to go empty
- **Faster solution:**
  - Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

Reducing Tag Overhead with Sub-Blocks

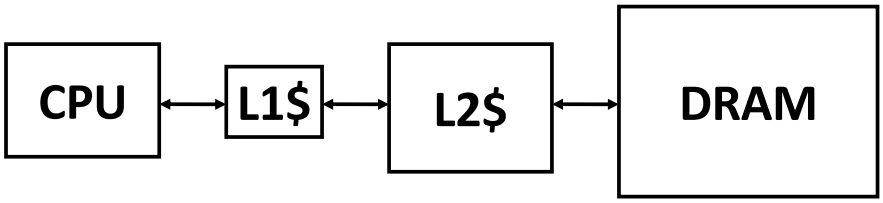
- **Problem**: Tags are too large, i.e., too much overhead
  - Simple solution: Larger lines, but miss penalty could be large.
- **Solution**: Sub-block placement (aka sector cache)
  - A valid bit added to units smaller than full line, called sub-blocks
  - Only read a sub-block on a miss
  - *If a tag matches, is the word in the cache?*

100	*1*		*1*		*1*		*1*	
300	*1*		*1*		*0*		*0*	
204	*0*		*1*		*0*		*1*	

Multilevel Caches

Problem: A memory cannot be large and fast

Solution: Increasing sizes of cache at each level



Local miss rate = misses in cache / accesses to cache

Global miss rate = misses in cache / CPU memory accesses

Misses per instruction = misses in cache / number of instructions

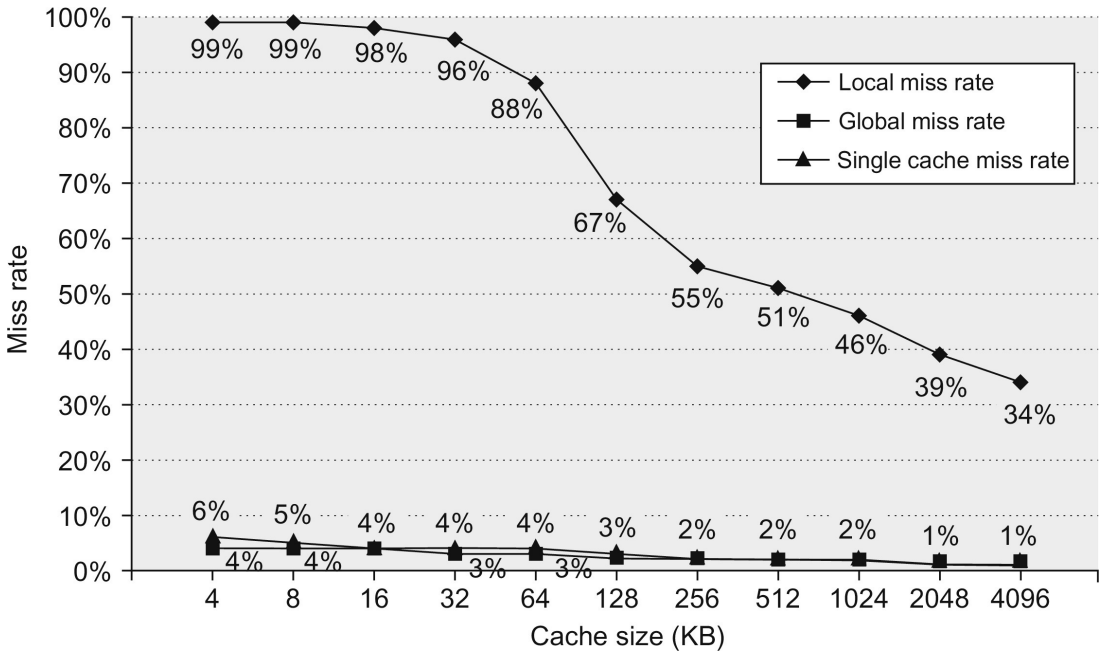


Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64 KiB first-level caches make little sense, as reflected in the high miss rates. After 256 KiB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32 KiB first-level cache. The L2 caches (unified) were two-way set associative with replacement. Each had split L1 instruction and data caches that were 64 KiB two-way set associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data were collected as in Figure B.4.

Presence of L2 influences L1 design

- Use smaller L1 if there is also L2
  - Trade increased L1 miss rate for reduced L1 hit time
  - Backup L2 reduces L1 miss penalty
  - Reduces average access energy

- Use simpler write-through L1 with on-chip L2
  - Write-back L2 cache absorbs write traffic, doesn't go off-chip
  - At most one L1 miss request per L1 access (no dirty victim write back) simplifies pipeline control
  - Simplifies coherence issues
  - Simplifies error recovery in L1 (can use just parity bits in L1 and reload from L2 when parity error detected on L1 read)

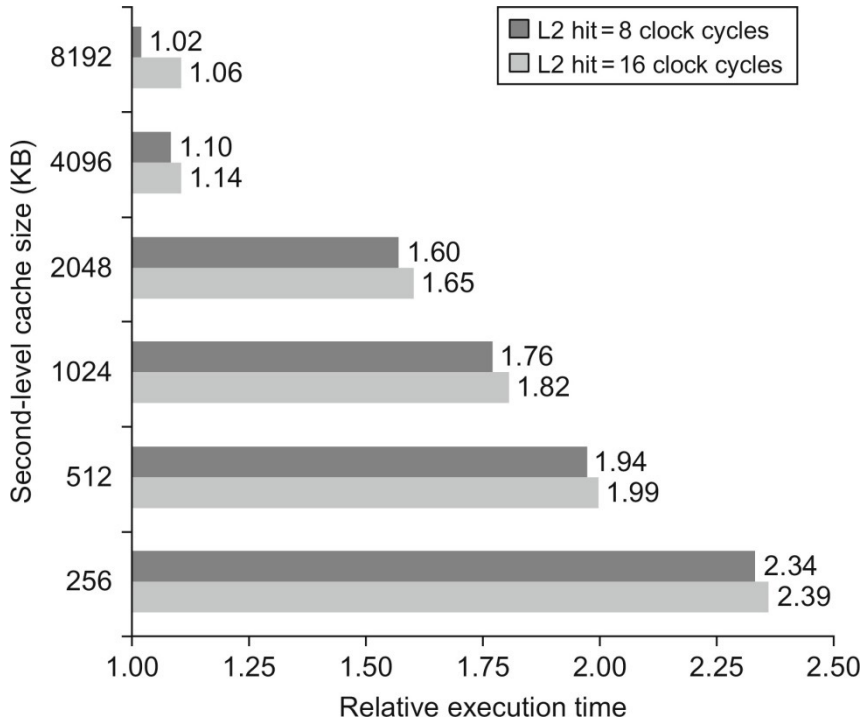
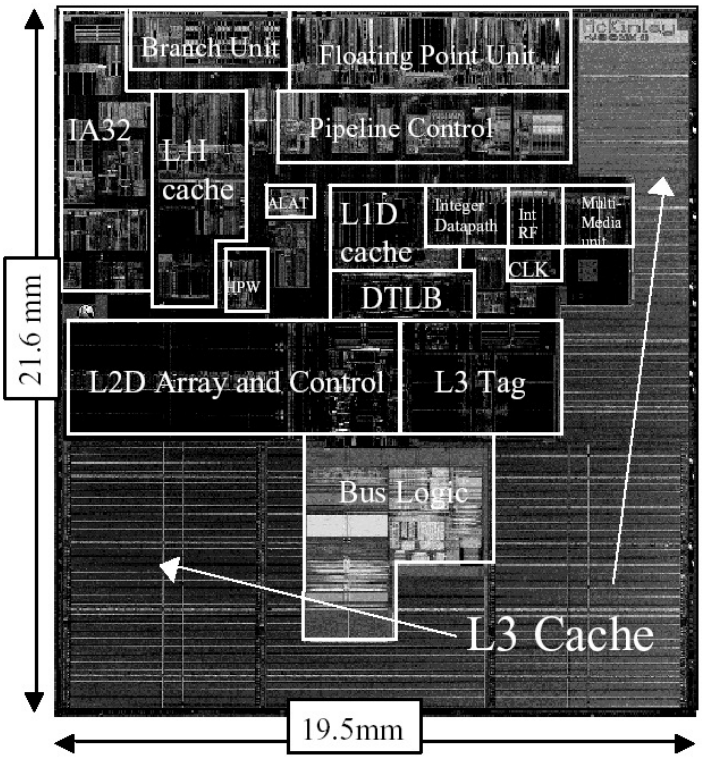


Figure B.15 Relative execution time by second-level cache size. The two bars are for different clock cycles for an L2 cache hit. The reference execution time of 1.00 is for an 8192 KiB second-level cache with a 1-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure B.14, using a simulator to imitate the Alpha 21264.

Inclusion Policy

- Inclusive multilevel cache:
  - Inner cache can only hold lines also present in outer cache
  - External coherence snoop access need only check outer cache
- *Exclusive* multilevel caches:
  - Inner cache may hold lines not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache

Itanium-2 On-Chip Caches
(Intel/HP, 2002)



Level 1: 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency

Level 2: 256KB, 4-way s.a, 128B line, quad-port (4 load or 4 store), five cycle latency

Level 3: 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency

Power 7 On-Chip Caches [IBM 2009]

32KB L1 I$/core

32KB L1 D$/core

3-cycle latency

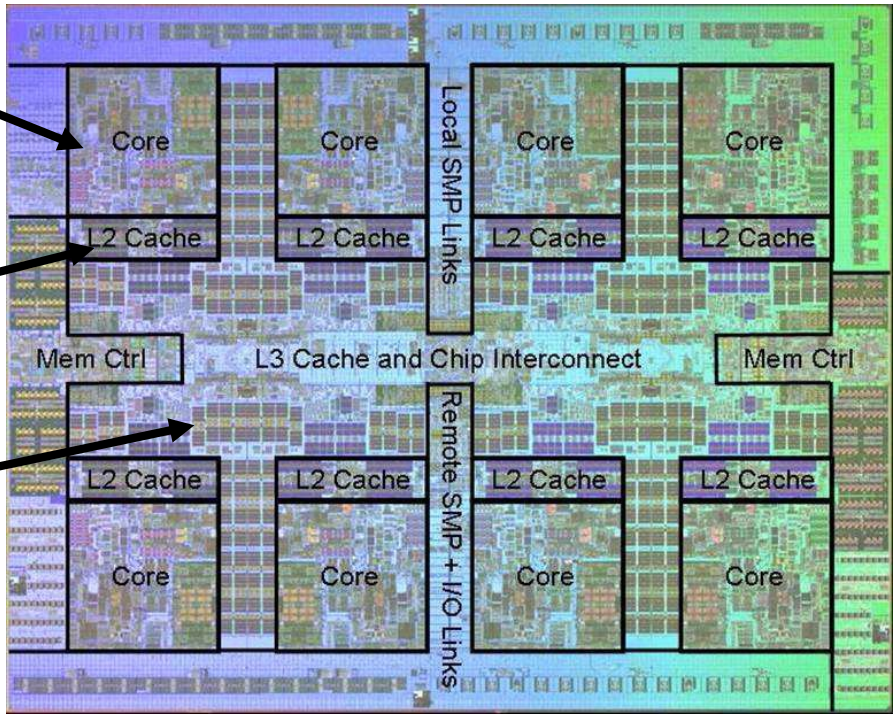
256KB Unified L2$/core

8-cycle latency

32MB Unified Shared L3$

Embedded DRAM (eDRAM)

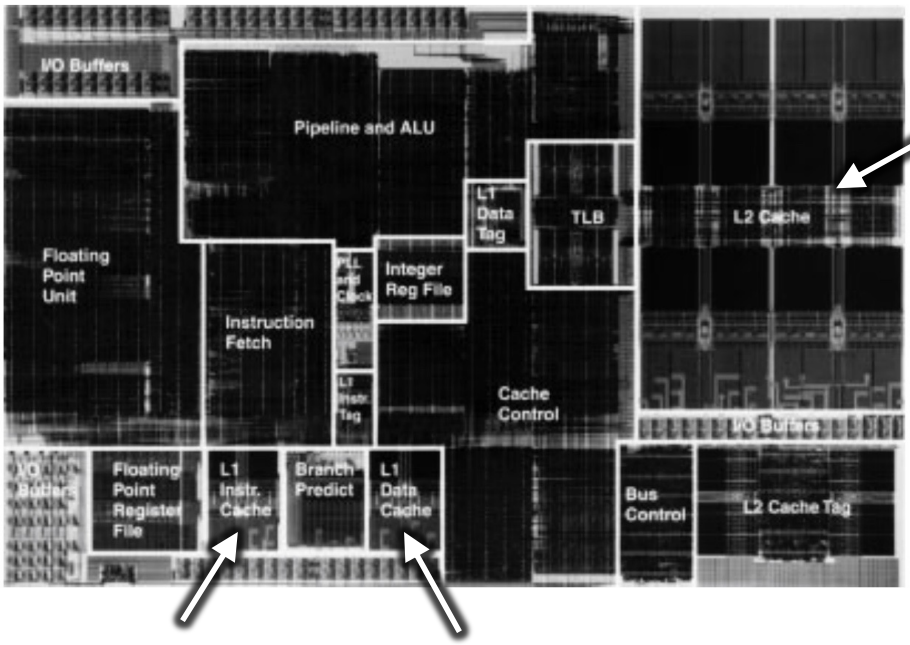
25-cycle latency to local slice



IBM z196 Mainframe Caches 2010

- 96 cores (4 cores/chip, 24 chips/system)
  - Out-of-order, 3-way superscalar @ 5.2GHz
- L1: 64KB I-$/core + 128KB D-$/core
- L2: 1.5MB private/core (144MB total)
- L3: 24MB shared/chip (eDRAM) (576MB total)
- L4: 768MB shared/system (eDRAM)

Exponential X704 PowerPC Processor (1997)



32KB L2 8-way Set-Associative Write-Back Unified Cache

2KB L1 Direct-Mapped Instruction Cache

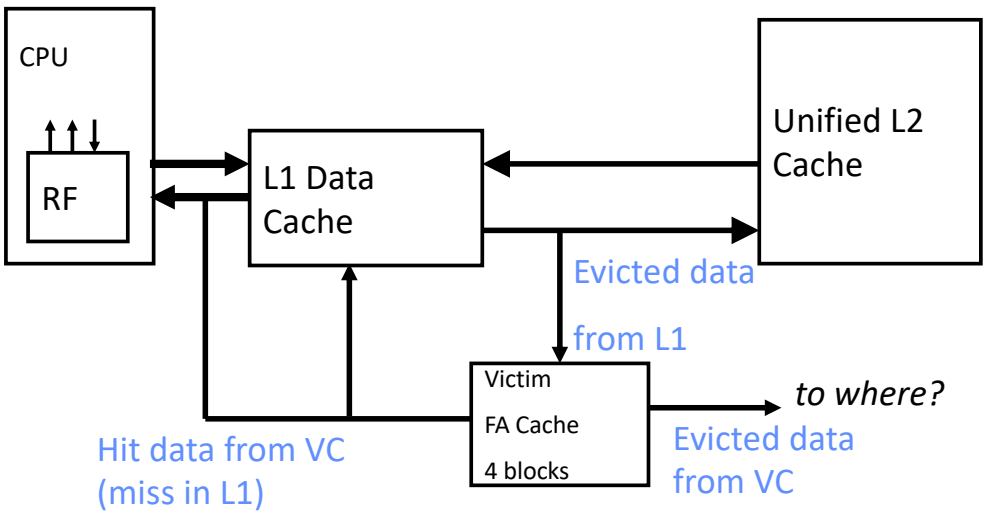
2KB L1 Direct-Mapped Write-Through Data Cache

0.5µm BiCMOS

Ran at 410-533MHz when other PC processors were much lower clock rate

Project delayed – missed market window for Apple

Victim Caches (HP 7200)



Victim cache is a small associative backup cache, added to a direct-mapped cache, which holds recently evicted lines

- First look up in direct-mapped cache
- If miss, look in victim cache
- If hit in victim cache, swap hit line with line now evicted from L1
- If miss in victim cache, L1 victim -> VC, VC victim->?

Fast hit time of direct mapped but with reduced conflict misses

MIPS R10000 Off-Chip L2 Cache (Yeager, IEEE Micro 1996)

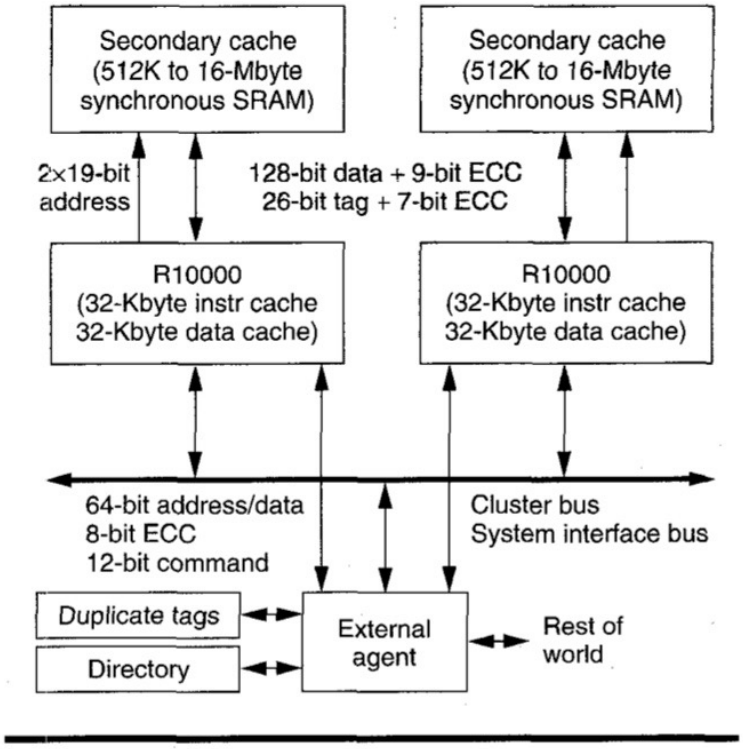
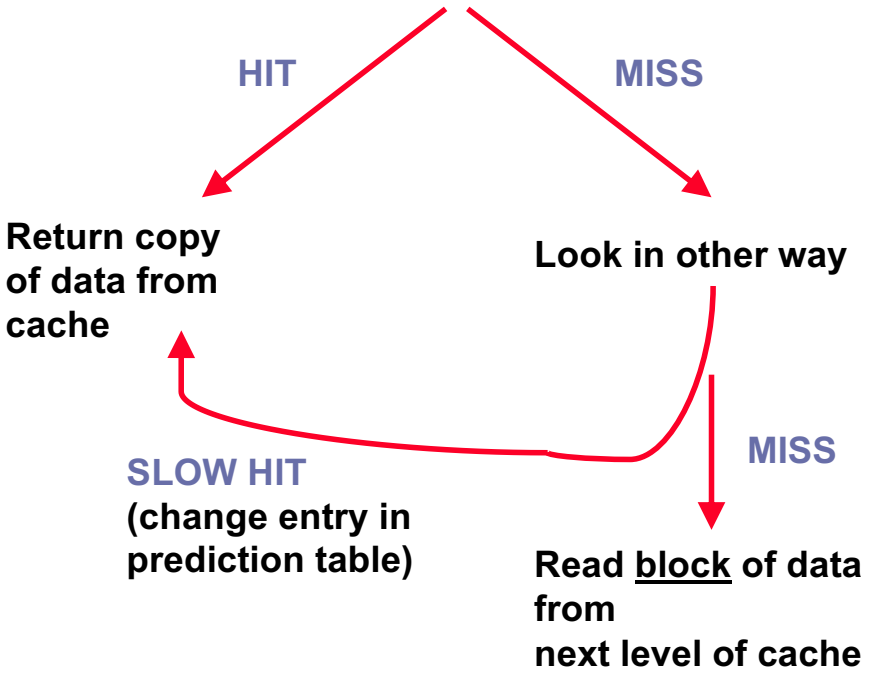


Figure 1. System configuration. The cluster bus directly connects as many as four chips.

Way-Predicting Caches (MIPS R10000 L2 cache)

- Use processor address to index into way-prediction table
- Look in predicted way at given index, then:



R10000 L2 Cache Timing Diagram

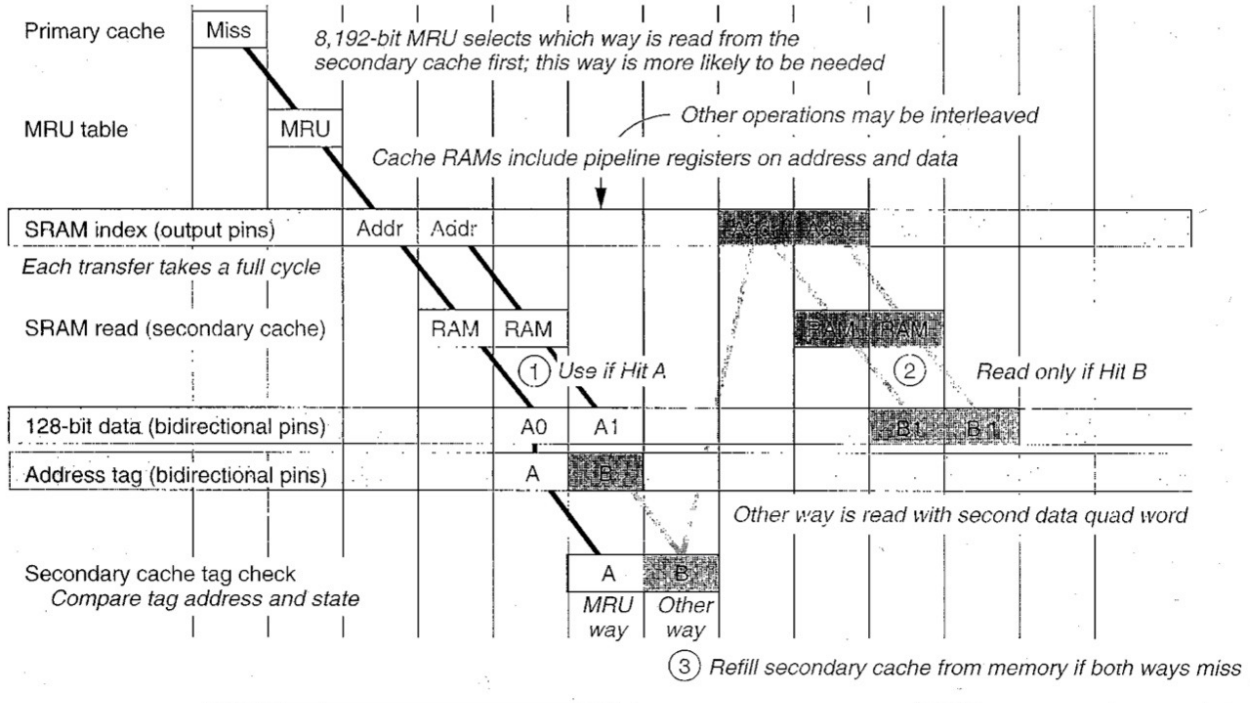
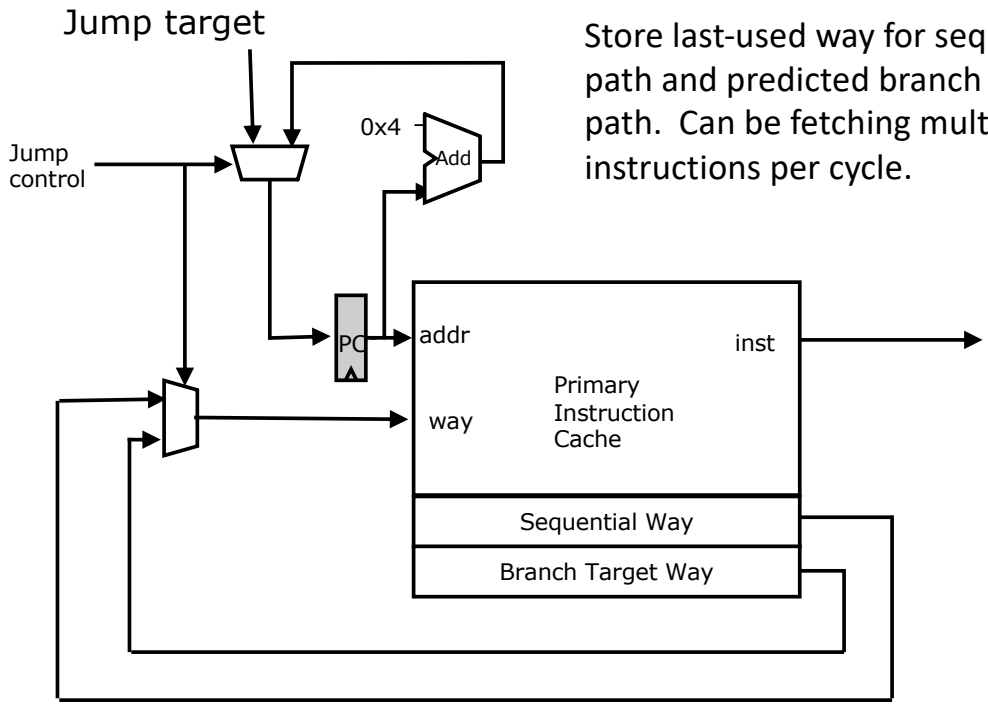


Figure 12. Refill from the set-associative secondary cache. In this example, the secondary clock equals the processor's internal pipeline clock. It may be slower.

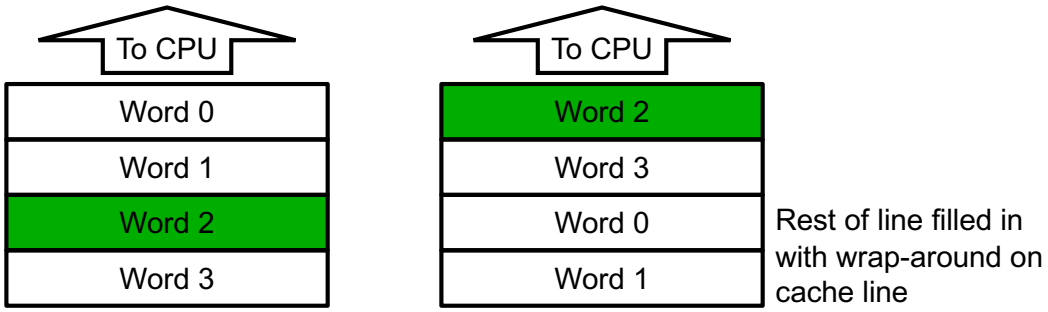
Way-Predicting Instruction Cache (Alpha 21264-like)



Store last-used way for sequential path and predicted branch taken path. Can be fetching multiple instructions per cycle.

Reduce Miss Penalty of Long Blocks: Early Restart and Critical Word First

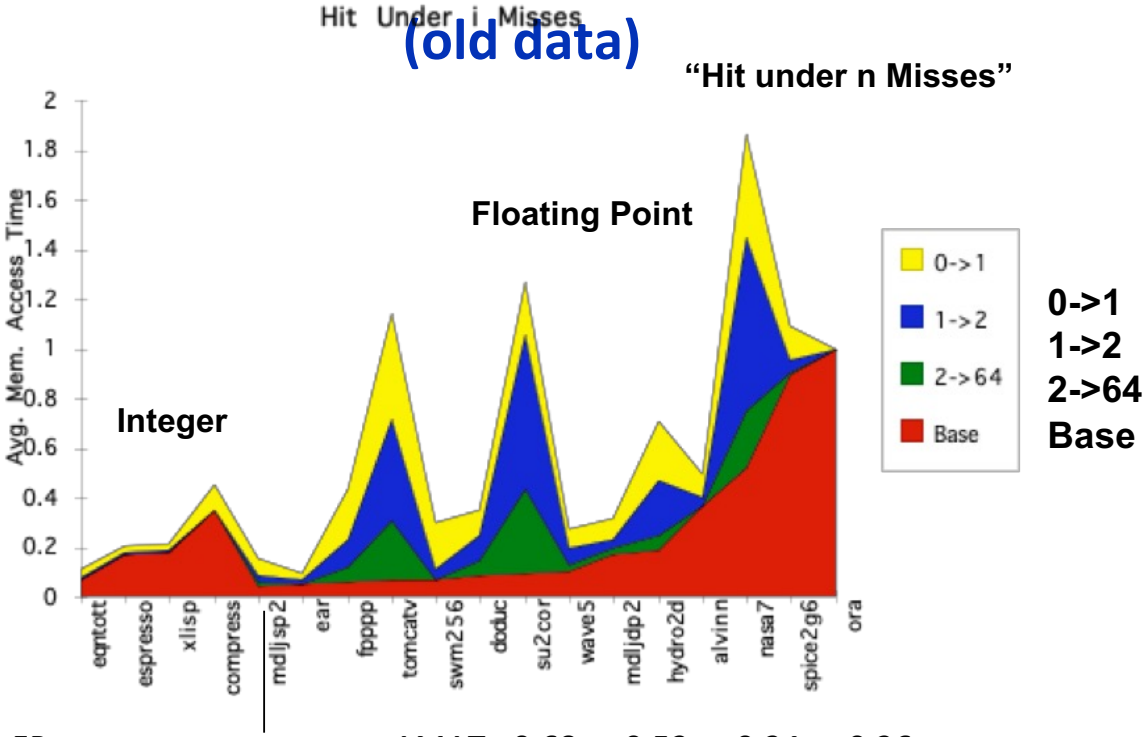
- Don't wait for full block before restarting CPU
- *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
- *Critical Word First*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
  - Long blocks more popular today ⇒ Critical Word 1st Widely used



Increasing Cache Bandwidth with Non-Blocking Caches

- *Non-blocking cache* or *lockup-free cache* allow data cache to continue to supply cache hits during a miss
  - requires Full/Empty bits on registers or out-of-order execution
- "*hit under miss*" reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses, and can get miss to line with outstanding miss (secondary miss)
  - Requires pipelined or banked memory system (otherwise cannot support multiple misses)
  - Pentium Pro allows 4 outstanding memory misses
  - Cray X1E vector supercomputer allows 2,048 outstanding memory misses

Value of Hit Under Miss for SPEC (old data)



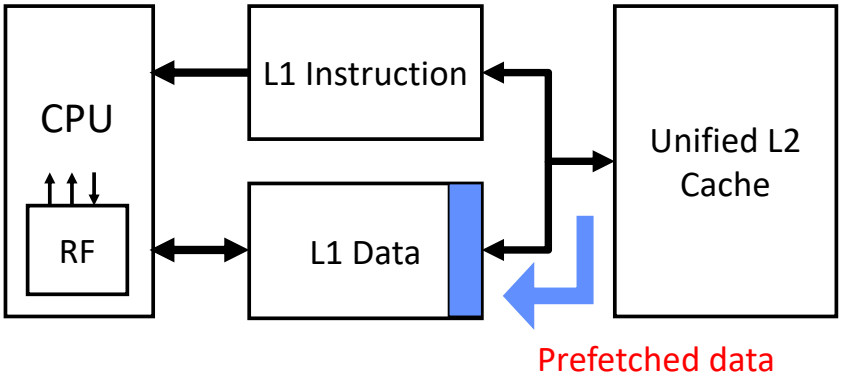
- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26
- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss, SPEC 92

Prefetching

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- What types of misses does prefetching affect?

Issues in Prefetching

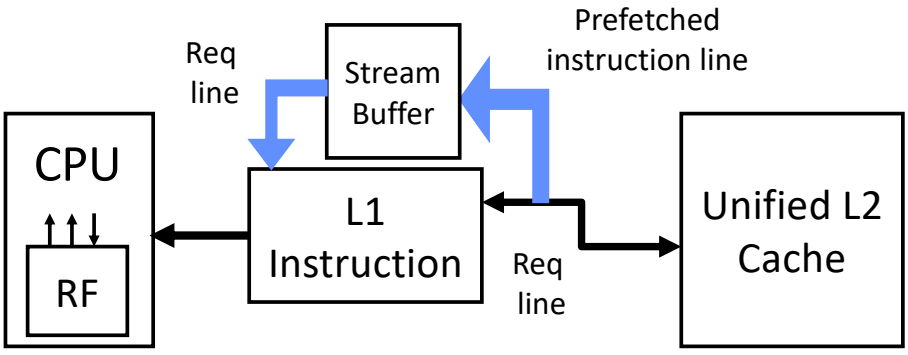
- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution



Hardware Instruction Prefetching

Instruction prefetch in Alpha AXP 21064

- Fetch two lines on a miss; the requested line (i) and the next consecutive line (i+1)
- Requested line placed in cache, and next line in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer line into cache and prefetch next line (i+2)



Hardware Data Prefetching

- Prefetch-on-miss:
  - Prefetch b + 1 upon miss on b
- One-Block Lookahead (OBL) scheme
  - Initiate prefetch for block b + 1 when block b is accessed
  - Why is this different from doubling block size?
  - Can extend to N-block lookahead
- Strided prefetch
  - If observe sequence of accesses to line b, b+N, b+2N, then prefetch b+3N etc.

- Example: IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access

Software Prefetching

```
for(i=0; i < N; i++) {
    prefetch( &a[i + 1] );
    prefetch( &b[i + 1] );
    SUM = SUM + a[i] * b[i];
 }
```

Software Prefetching Issues

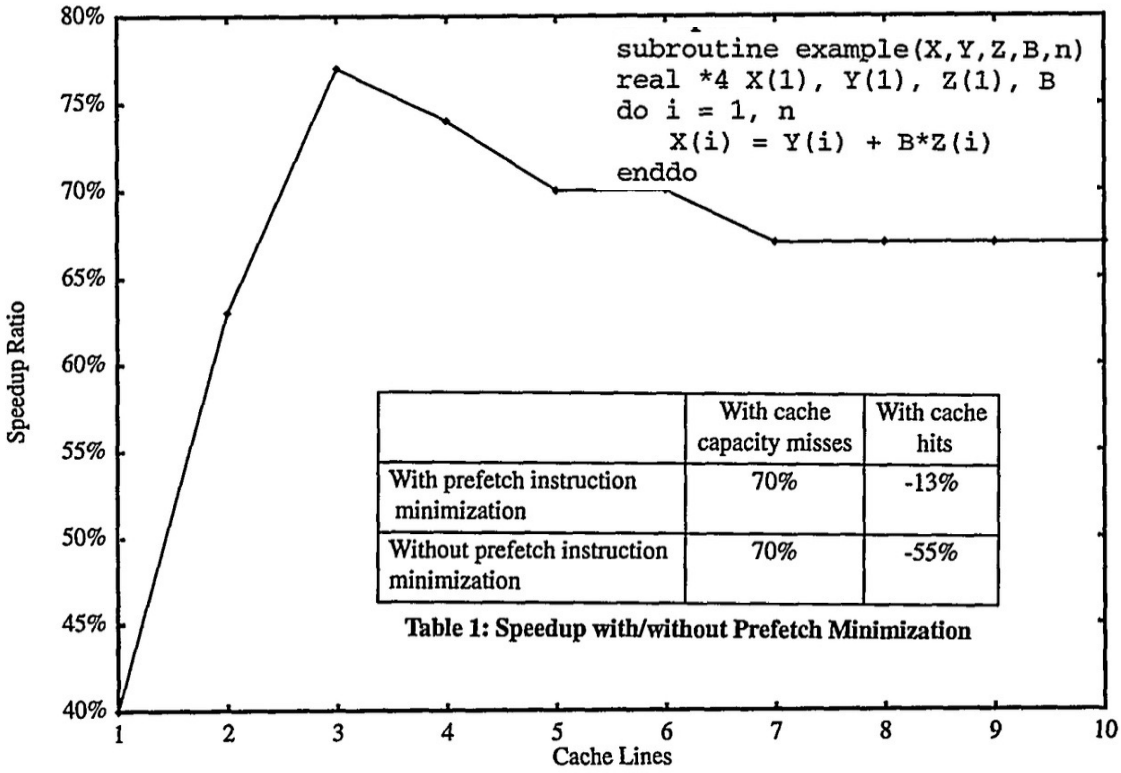
- Timing is the biggest issue, not predictability
  - If you prefetch very close to when the data is required, you might be too late
  - Prefetch too early, cause pollution
  - Estimate how long it will take for the data to come into L1, so we can set P appropriately
  - *Why is this hard to do?*

```
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
}
```

Must consider cost of prefetch instructions

Software Prefetching Example

["Data prefetching on the HP PA8000", Santhanam et al., 1997]



	With cache capacity misses	With cache hits
With prefetch instruction minimization	70%	-13%
Without prefetch instruction minimization	70%	-55%

Table 1: Speedup with/without Prefetch Minimization

Figure 2: Speedup Ratio for Different Prefetch Distances

Compiler Optimizations

- Restructuring code affects the data access sequence
  - Group data accesses together to improve spatial locality
  - Re-order data accesses to improve temporal locality
- Prevent data from entering the cache
  - Useful for variables that will only be accessed once before being replaced
  - Needs mechanism for software to tell hardware not to cache data ("no-allocate" instruction hints or page table bits)
- Kill data that will never be used again
  - Streaming data exploits spatial locality but not temporal locality
  - Replace into dead cache locations

Loop Interchange

```
for(j=0; j < N; j++) {
    for(i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

```
for(i=0; i < M; i++) {
    for(j=0; j < N; j++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

What type of locality does this improve?

Loop Interchange

- Previous code improves spatial locality by providing stride 1 accesses to array *x* (bottom) instead of stride *M* accesses (top)

- Changes in the stride accesses are dependent on the programming language

Loop Fusion

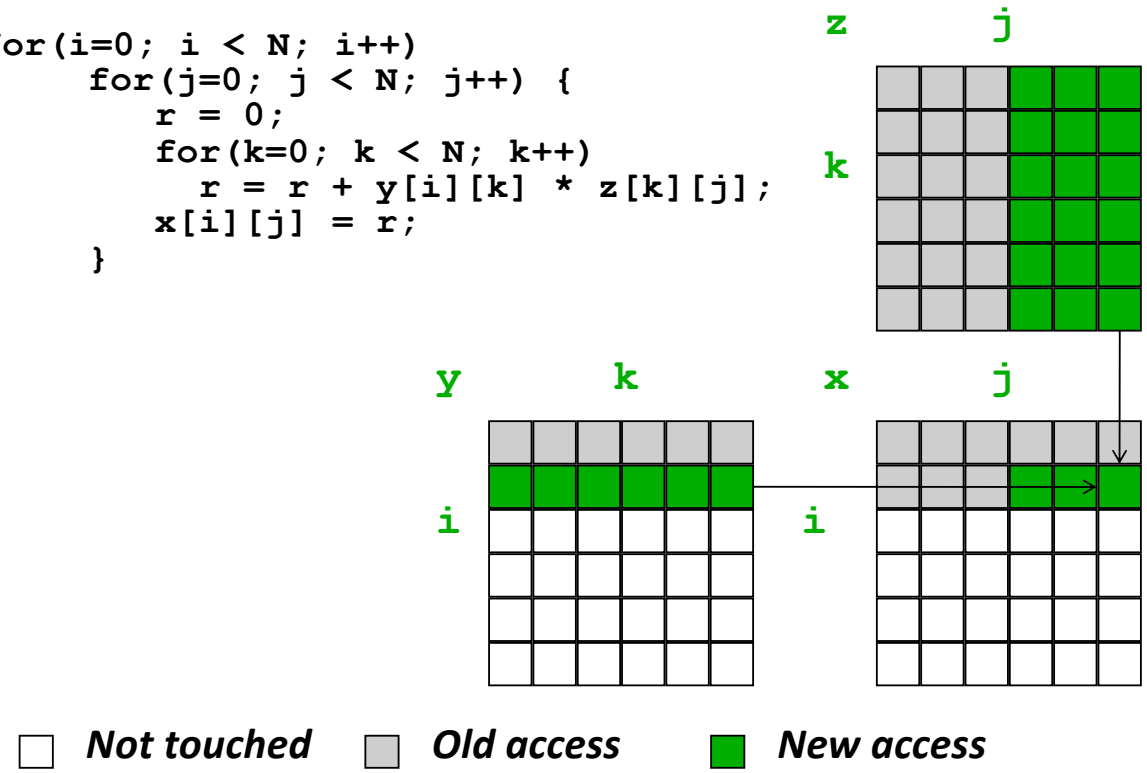
```
for(i=0; i < N; i++)
      a[i] = b[i] * c[i];

for(i=0; i < N; i++)
     d[i] = a[i] * c[i];
```

```
for(i=0; i < N; i++)
{
        a[i] = b[i] * c[i];
        d[i] = a[i] * c[i];
}
```

What type of locality does this improve?

Matrix Multiply, Naïve Code

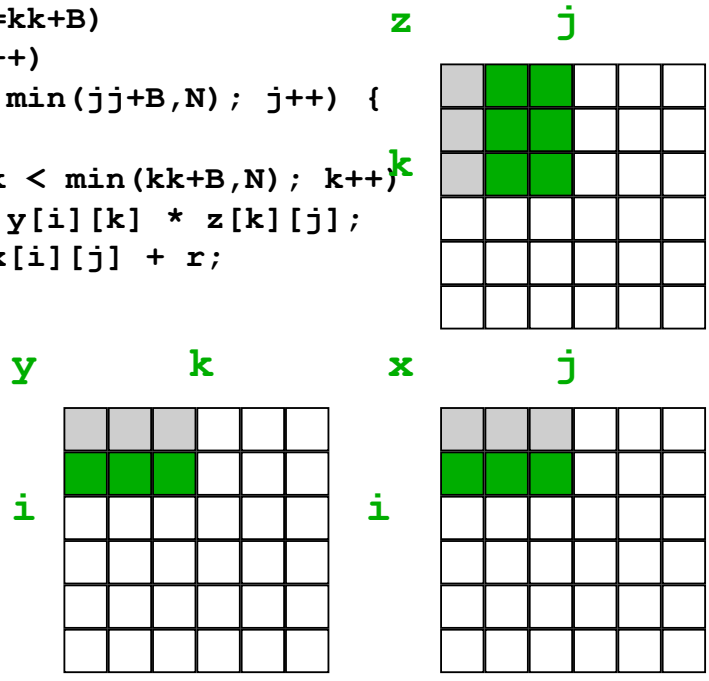
```
for(i=0; i < N; i++)
     for(j=0; j < N; j++) {
        r = 0;
        for(k=0; k < N; k++)
          r = r + y[i][k] * z[k][j];
        x[i][j] = r;
     }
```



□ *Not touched* ▒ *Old access* ■ *New access*

Matrix Multiply with Cache Tiling

```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
       for(i=0; i < N; i++)
           for(j=jj; j < min(jj+B,N); j++) {
              r = 0;
              for(k=kk; k < min(kk+B,N); k++)
                 r = r + y[i][k] * z[k][j];
              x[i][j] = x[i][j] + r;
           }
```



What type of locality does this improve?