ECE 485/585
Microprocessor System Design

Prof. Mark G. Faust

Maseeh College of Engineering
and Computer Science

PORTLAND STATE UNIVERSITY

Caches

Outline

- *Handouts: Hall*
- Topics
  - Cache Basics
    - Memory vs. Processor Performance
    - The Memory Hierarchy
      - Registers, SRAM, DRAM, Disk
    - Spatial and Temporal Locality
    - Cache Hits and Misses
    - Direct Mapped Caches
    - Two-Way, Four-Way Caches
    - Fully Associative (N-Way) Caches
    - Cache Line Replacement Algorithms

Outline

- Topics
  - Cache Performance
  - Improving Cache Performance
  - Cache Coherence
  - Intel Cache Evolution
  - Multicore Caches
  - Cache Design Issues

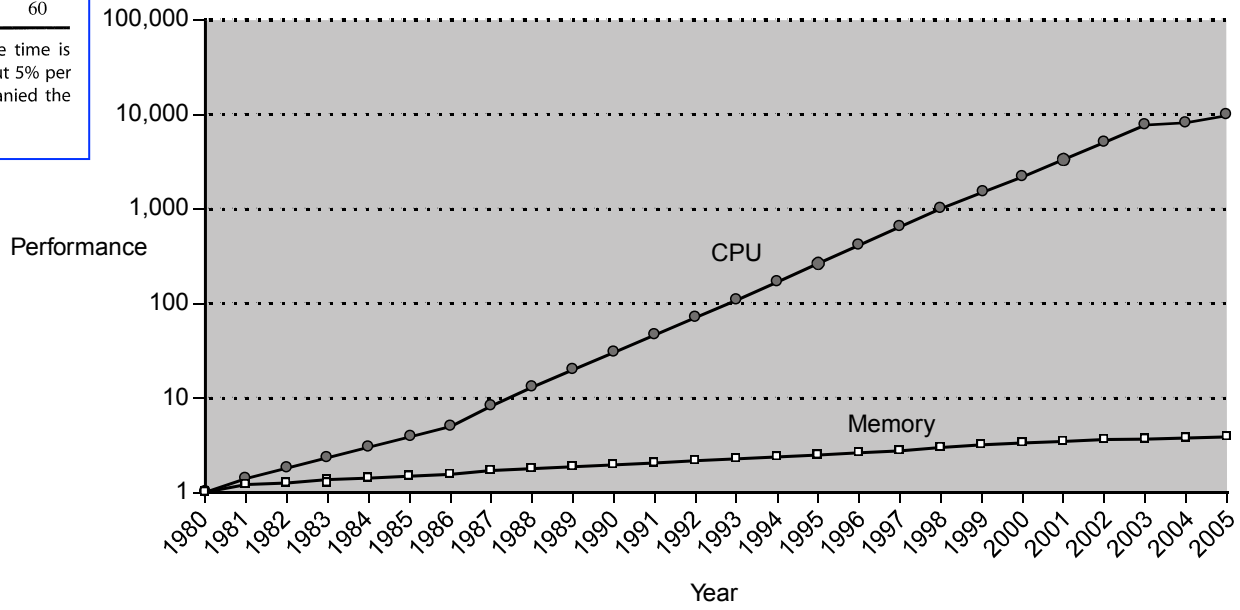
CPU/Memory Performance Gap

Year of introduction	Chip size	Row access strobe (RAS) Slowest DRAM (ns)	Row access strobe (RAS) Fastest DRAM (ns)	Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
1980	64K bit	180	150	75	250
1983	256K bit	150	120	50	220
1986	1M bit	120	100	25	190
1989	4M bit	100	80	20	165
1992	16M bit	80	60	15	120
1996	64M bit	70	50	12	110
1998	128M bit	70	50	10	100
2000	256M bit	65	45	7	90
2002	512M bit	60	40	5	80
2004	1G bit	55	35	5	70
2006	2G bit	50	30	2.5	60

Figure 5.13 Times of fast and slow DRAMs with each generation. (Cycle time is defined on page 310.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs.

Memory Technology Trends

Relative Performance Gains

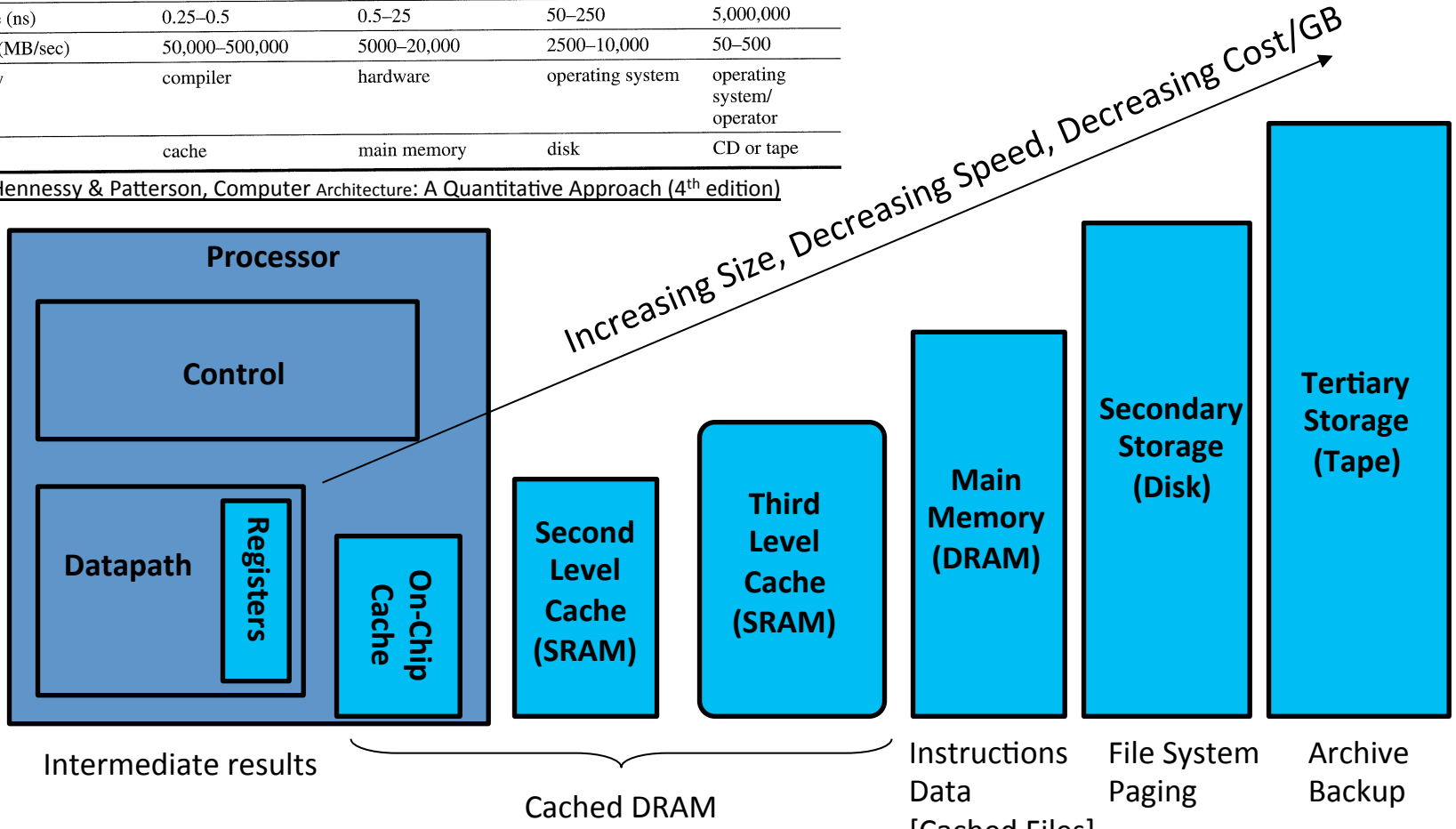


From Hennessy & Patterson, Computer Architecture: A Quantitative Approach (4th edition)

Computer Memory Hierarchy

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 512 GB	> 1 TB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25–0.5	0.5–25	50–250	5,000,000
Bandwidth (MB/sec)	50,000–500,000	5000–20,000	2500–10,000	50–500
Managed by	compiler	hardware	operating system	operating system/ operator
Backed by	cache	main memory	disk	CD or tape

From Hennessy & Patterson, Computer Architecture: A Quantitative Approach (4th edition)



Intel Pentium 4 3.2 GHz Server

Component	Access Speed (Time for data to be returned)
Registers	1 cycle = 0.3 nanoseconds
L1 Cache	3 cycles = 1 nanoseconds
L2 Cache	20 cycles = 7 nanoseconds
L3 Cache	40 cycles = 13 nanoseconds
Memory	300 cycles = 100 nanoseconds

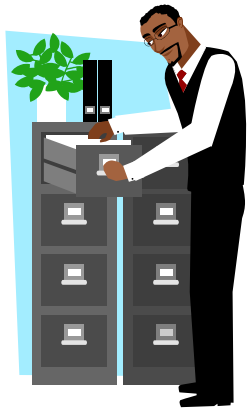
How is the Hierarchy Managed?

- Registers ←→ Memory
  - Compiler
  - Programmer
- Cache ←→ Memory
  - Hardware
- Memory ←→ Disk
  - Operating System (Virtual Memory: Paging)
  - Programmer (File System)

Principle of Locality

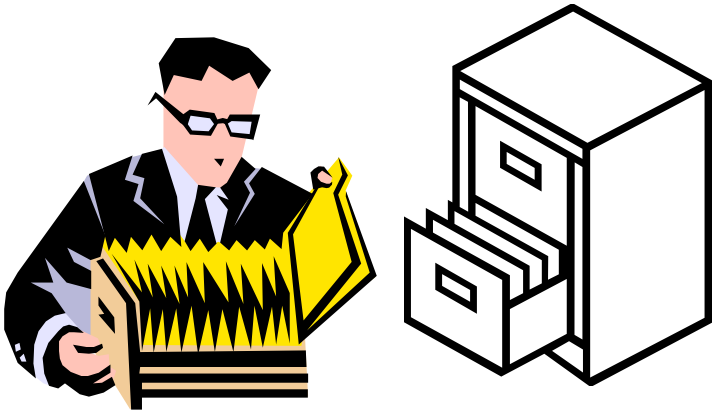
- Makes it possible for a cache to provide improvement in performance
- Temporal Locality
  - A referenced item is likely to be referenced again soon
    - Code example: a commonly used function or loop
    - Data example: commonly used variables, pointers
- Spatial Locality
  - Nearby items are likely to be referenced soon
    - Code example: most likely place for next instruction is IP+1
    - Data example: arrays, characters in strings

Caching – Student Advising Analogy



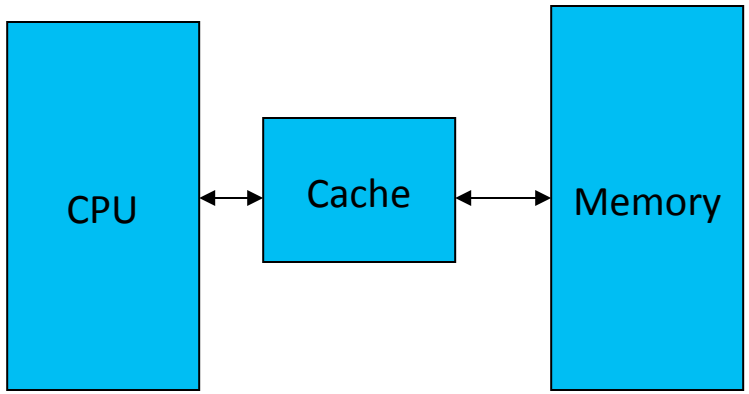
Thousands of student folders
Indexed by 9-digit student ID
Located down the hall – long walk

Space for 100 file folders at my desk
Located at my side – short access time



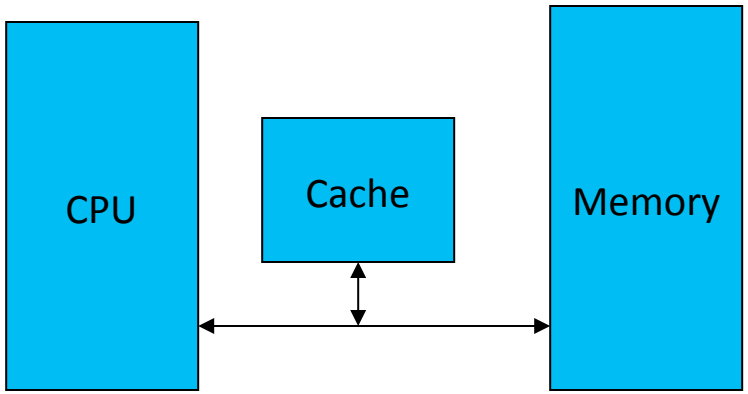
Look Through

- Pros
  - Fewer CPU/memory cycles
  - Less bus contention
    - e.g. 90% hit rate
    - Memory bus bandwidth available to other bus masters
  - No CPU wait on writes
- Cons
  - Possible increase in access time
  - Complexity



Look Aside

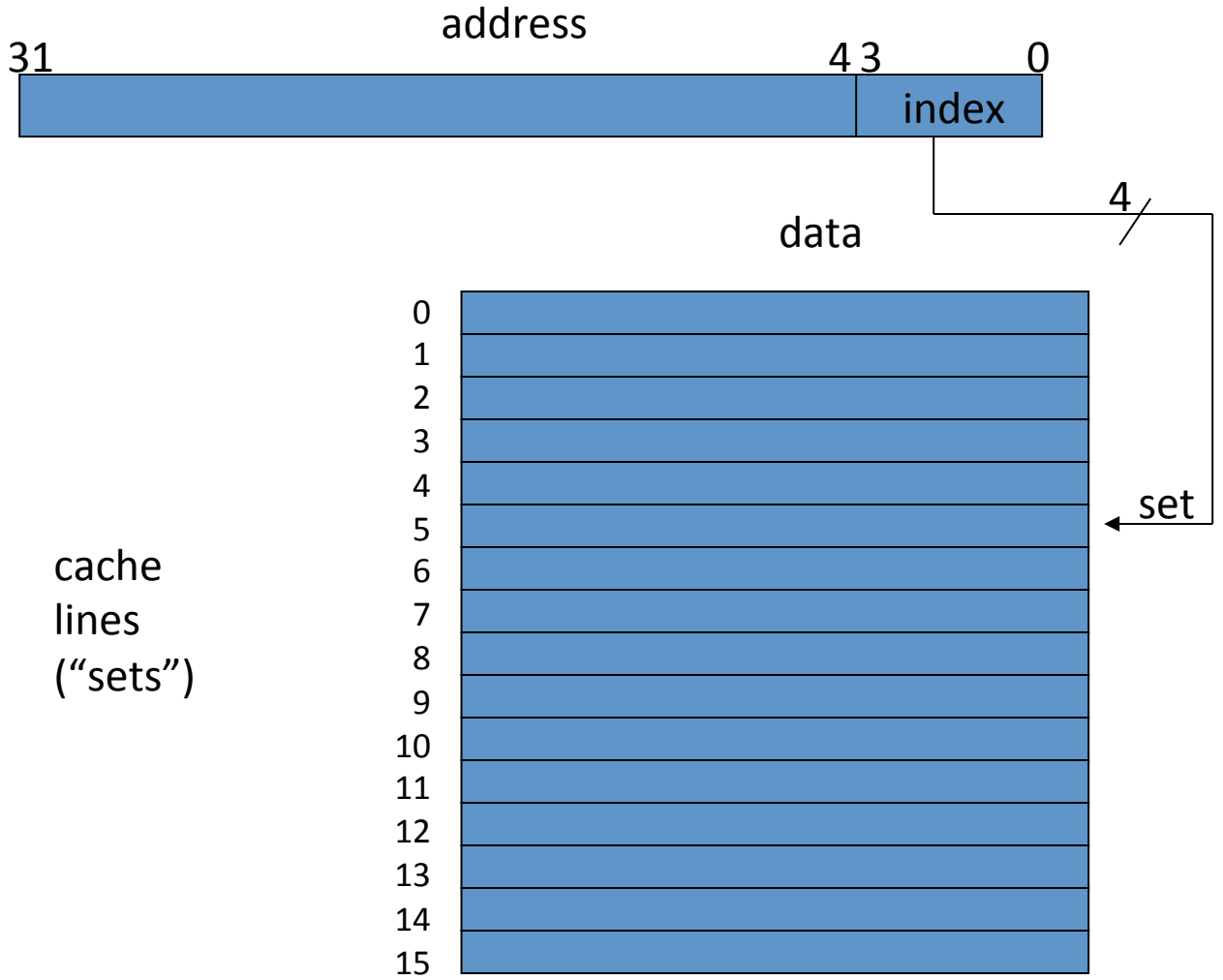
- Pros
  - Faster response to cache miss
    - Memory cycle already begun
  - Simpler design
- Cons
  - More main memory bus traffic
  - CPU/cache access blocks other bus masters



Cache Organization

- How Is The Cache Laid Out?
  - The cache is made up of a number of cache lines (sometimes called blocks)
  - Data is hauled into the cache from memory in "chunks" (may be smaller than a line)
  - If CPU requests 4 bytes of data, cache gets entire line (32/64/128 bytes)
  - Spatial locality says you're likely to need that data anyway
  - Incur cost only once rather than each time CPU needs piece of data
  - The Pentium P4 Xeon's Level 1 Data Cache contains 8K bytes
  - The cache lines are each 64 bytes
  - This gives 8192 bytes / 64 bytes = 128 cache lines

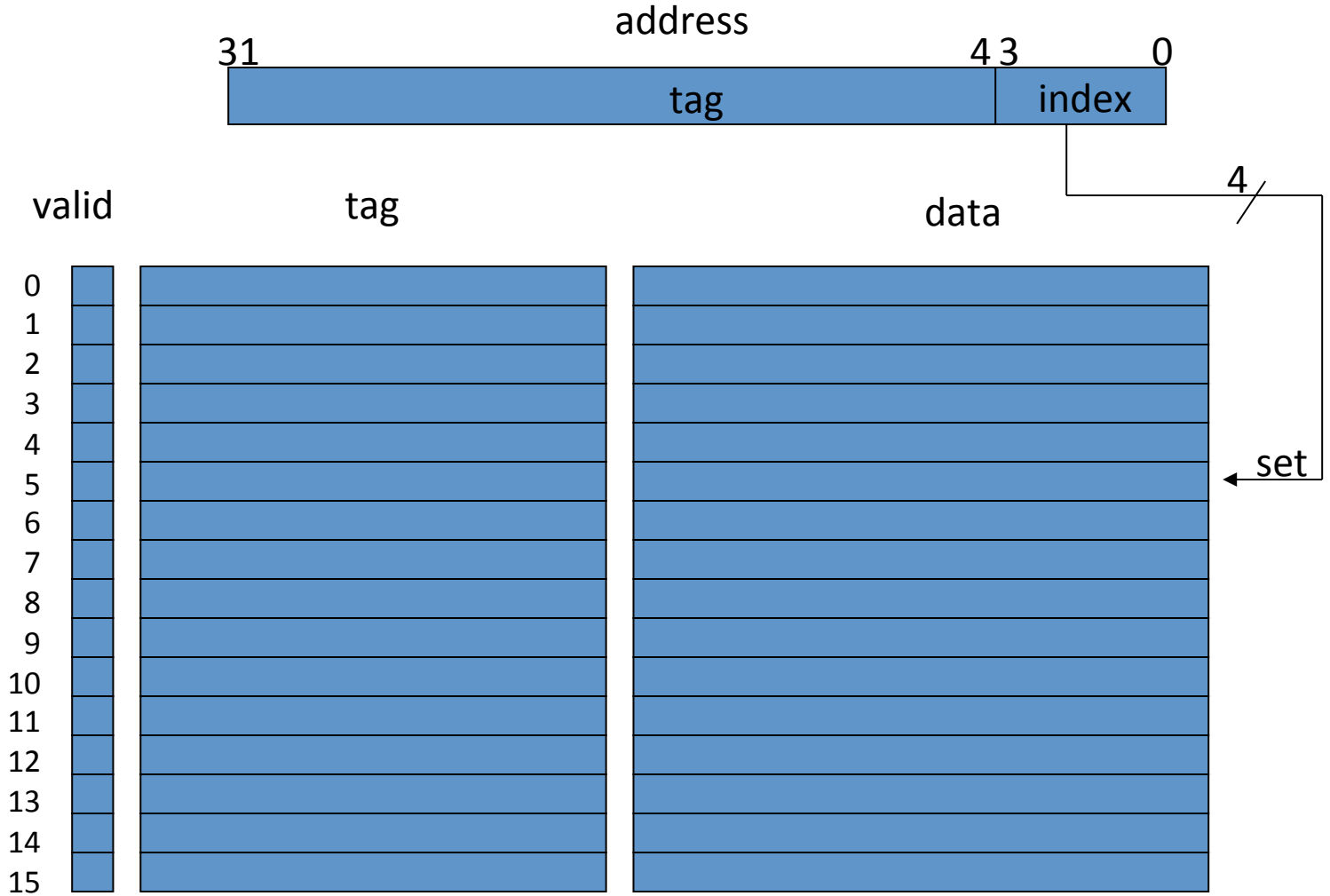
Simple Direct Mapped Cache



Use least significant 4 bits to determine which slot to cache data in

But…2^{28} different addresses could have their data cached in the same spot

Simple Direct Mapped Cache



Need to store tag to be sure the data is for this address and not another
(Only need to store the address minus the index bits – 28 bits in this example)

Cache Behavior – Reads

Read behavior

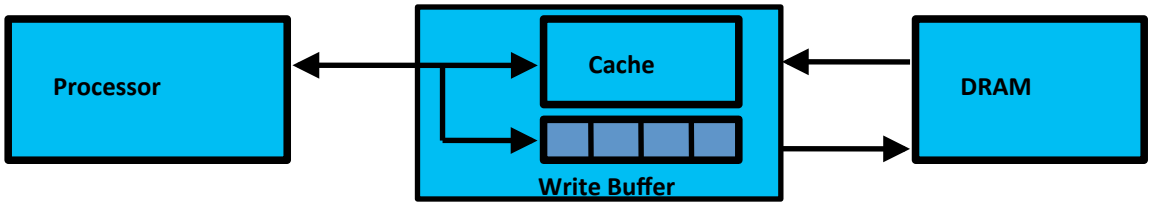
```
if Valid bit clear   /* slot empty – cache miss */
    stall CPU
    read cache line from memory
    set Valid bit
    write Tag bits
    deliver data to CPU
else                 /* slot occupied */
    if Tag bits match  /* cache hit!  */
        deliver data to CPU
    else             /* occupied by another – cache miss */
        stall CPU
        cast out existing cache line (“victim”)
        read cache line from memory
        write Tag bits
        deliver data to CPU
```

Cache Behavior – Writes

- Policy decisions for write
  - Write Through
    - Replace data in cache and memory
    - Requires write buffer to be effective
      - Allows CPU to continue without waiting for DRAM
  - Write Back
    - Replace data in cache only
    - Requires addition of “dirty” bit in tag/valid memory
    - Write back later
      - Cache flush
      - Line becomes victim and is cast out
- Policy decisions for write miss
  - Write Allocate
    - Place the data into the cache
  - Write No Allocate (or Write Around)
    - Don’t place the data in the cache
    - Philosophy – successive writes (without intervening read) unlikely
    - Saves not only the cache line fill for the requested cache line but possibility of casting out a line which is more likely to be used later

Write Buffer for Write-Through

- A Write Buffer is needed between cache and memory if using Write Through policy to avoid having processor wait
  - Processor writes data into the cache and the write buffer
  - Memory controller write contents of the buffer to memory
- Write Buffer is just a FIFO
  - Intel: “posted write buffer” PWB
  - Small depth
  - Store frequency << 1/DRAM write cycle



Cache Behavior – Writes

Write behavior

```
if Valid bit set      /* slot occupied */
   if Tag bits match   /* cache hit! */
      write data to cache
      write data to memory - or - set "dirty" bit for cache line
   else                /* occupied by another */
      stall CPU
      cast out existing cache line ("victim")
      read cache line from memory
      write Tag bits
      write data to cache
      write data to memory - or - set "dirty" bit for cache line

else                  /* slot empty */
      stall CPU
      read cache line from memory
      write Tag bits
      set Valid bit
      write data to cache
      write data to memory - or - set "dirty" bit for cache line
```

write through or write back

Why?

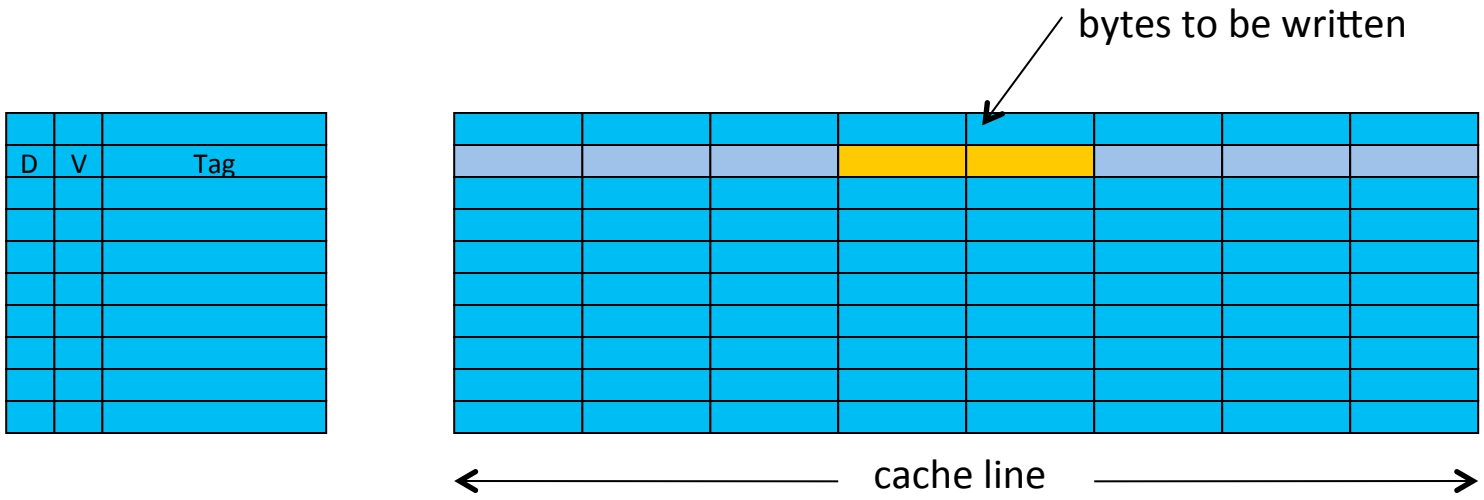
write through or write back

assumes write allocate

Why?

write through or write back

Why read a cache line for a write?



Data being written by CPU is smaller than cache line
Write misses in cache
Have only single valid bit and tag bits for entire line
Subsequent read operation must find valid data for rest of cache line

Casting out a victim

- Depends upon policies
  - Write Through
    - Data in cache isn't the only current copy (memory is up to date)
    - Just over-write victim cache line with new cache line (change tag bits)
  - Write Back
    - Must check dirty bit to see if victim cache line is modified
    - If so, must write the victim cache line back to memory

- Can lead to interesting behavior
  - A CPU "read" can cause memory "write" followed by "read"
    - Write back dirty cache line (victim)
    - Read new cache line
  - A CPU "write" can cause a memory "write" followed by a "read"
    - Write back dirty cache line (victim)
    - Read new cache line into which data will be written in cache

What if…

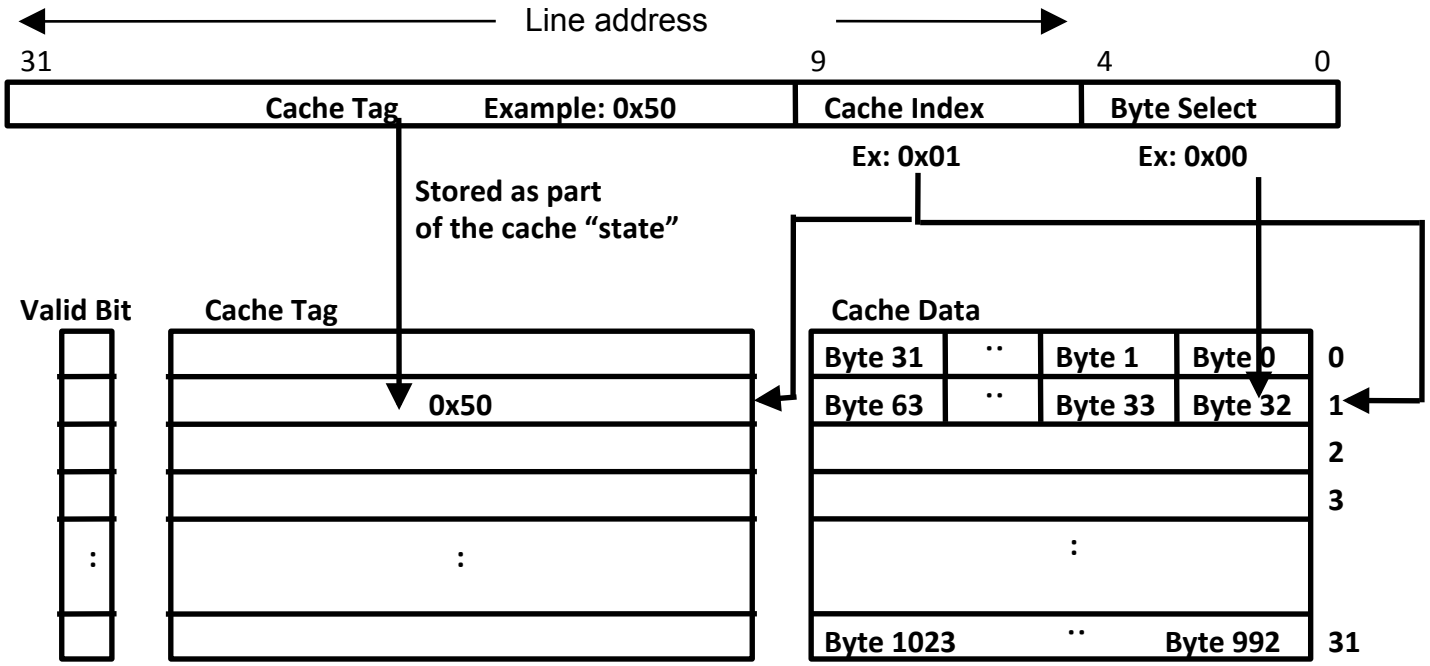
- Cache miss
- Cache location at Cache Index occupied
- Called a “cache conflict” or “collision”
- “Cast Out” existing entry (“victim”)
- Replace with new entry
- What if we need that earlier entry again?
  - Thrashing
- Solution – N-way set associative caches
  - Simultaneously hold in cache two (or more) lines that would have been forced to share same place in direct mapped cache

Cache Organization

- How Does The Cache Manage the Cache Lines?
  - Associativity describes how data is stored in the cache
  - Direct Mapped
    - Associativity == 1
    - Each set has single line
    - If it's in the cache there's only one place it could be
  - N-way Associativity
    - Each set contains N lines
    - There are N places ("ways") the line could be
  - Fully associative
    - All cache lines share the same possible places

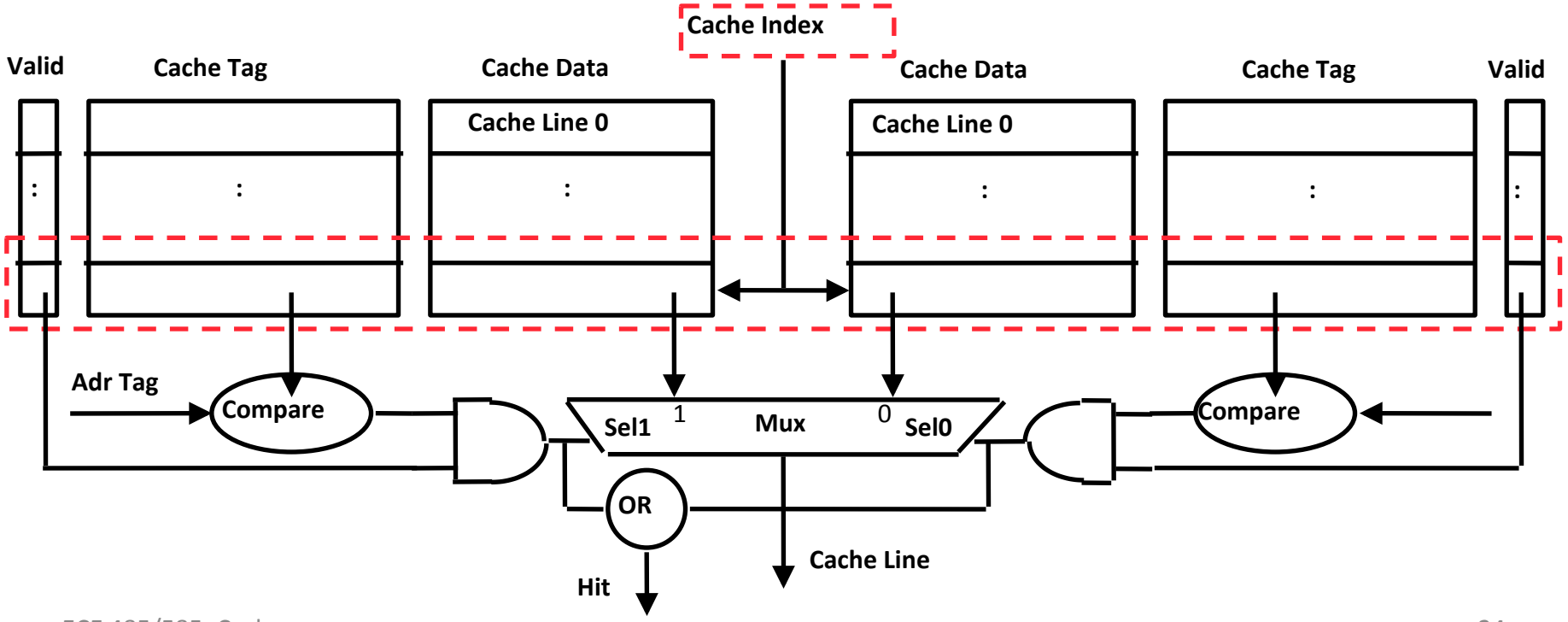
Direct Mapped Cache

- Example: A 2^N byte cache
  - 1 KB Direct−mapped cache with 32 byte lines (blocks)
  - The upper-most (32 - N) bits are always the cache tag
  - The lowest M bits are the byte select (line size = 2^M)
  - On cache miss, read in complete line (block)



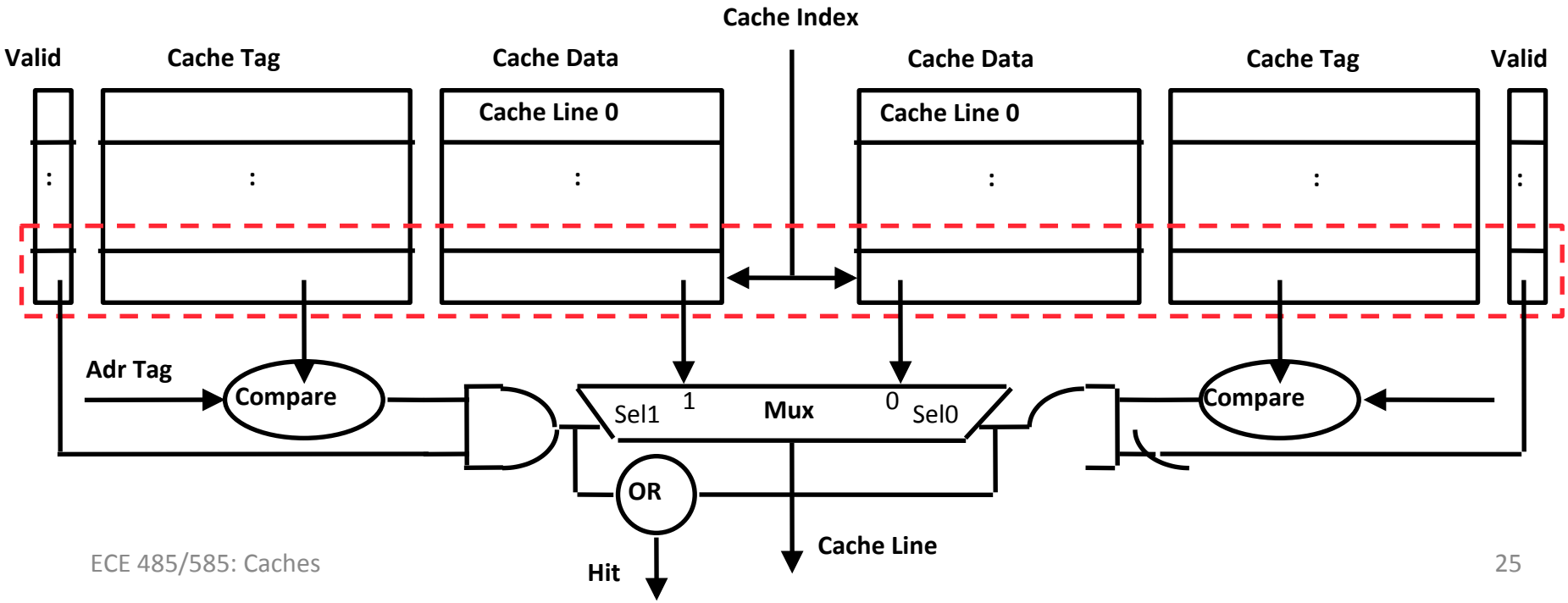
Set Associative Cache

- N-Way Set Associative
  - N cache entries with same index
  - N direct mapped caches operating in parallel
- Example:
  - 2 way set associative
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared to the input in parallel
  - Data is selected based on the tag result

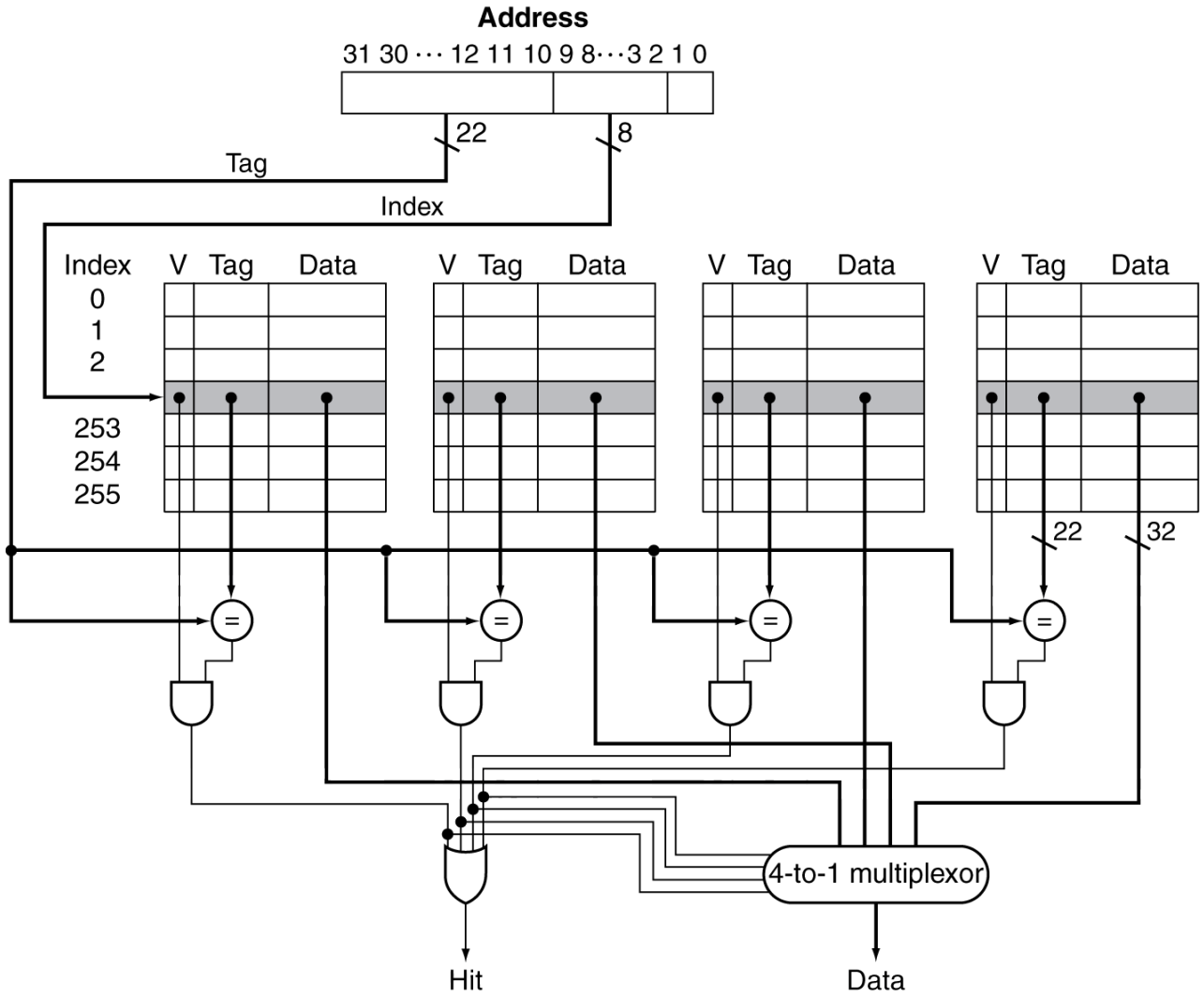


Comparison

- N-way Set Associative Cache vs. Direct Mapped Cache
  - N comparators vs. 1
  - Extra mux delay for the data
  - Data comes after hit/miss decision and set selection
  - Less thrashing (fewer "conflict" misses)
- In a direct mapped cache, cache line is available before hit/miss detection
  - Possible to assume a hit and continue. Recover later if miss.

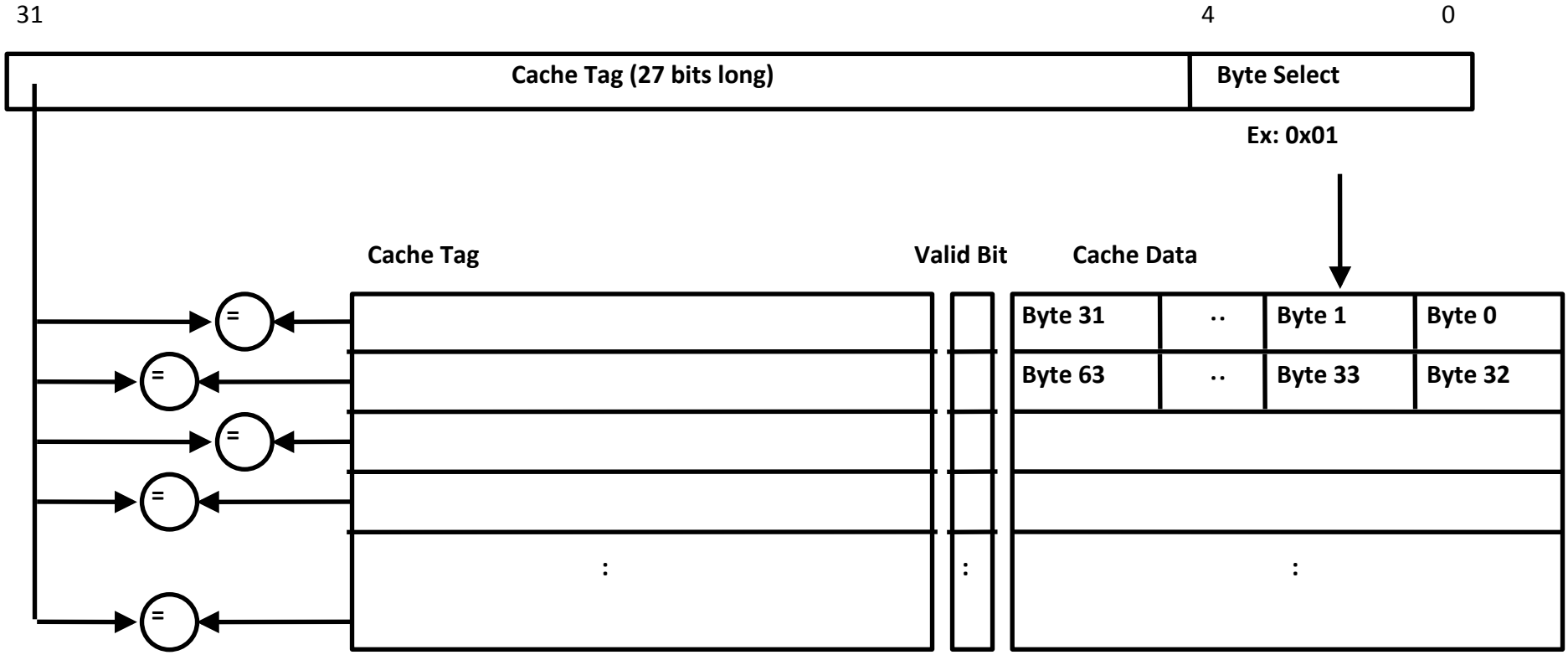


An Implementation



Fully Associative Cache

- Fully associative cache
  - Eliminate cache index
  - Compare the cache tags of all cache entries in parallel
- Example
  - Line size = 32 bytes, requires N 27-bit comparators

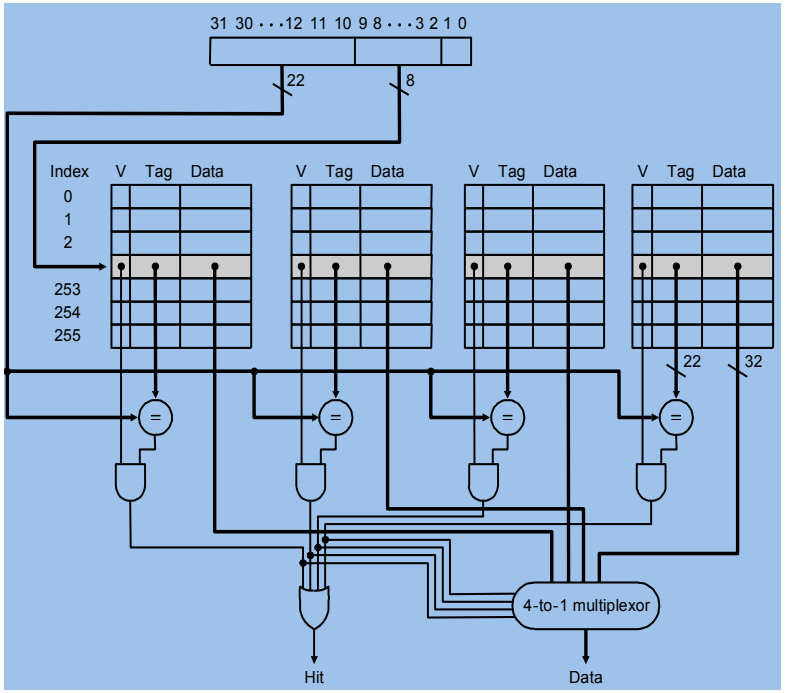


Sources of Cache Misses

- Compulsory
  - First access to a block (reboot, cache cleared)
  - Insignificant if you're executing trillions of instructions
  - Solution: Nothing you can do about it
- Capacity
  - Cache cannot simultaneously contain all blocks required
  - Solution: Increase cache size
- Conflict (Collision)
  - Block was cast out to make room for another block at same location in cache (doesn't occur for fully associative cache)
  - Solution: Increase associativity
  - Solution: Increase cache size
- Coherence (Invalidation)
  - Other process (I/O, processor) invalidated block

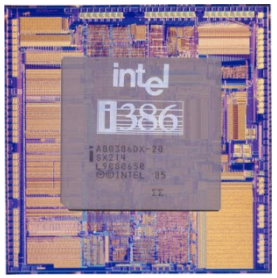
Which Line is Replaced on Miss?

- "Replacement Strategy" or "Replacement Policy"
- Direct-mapped
  - Easy – only one choice
- Associative
  - Random
  - FIFO (First In First Out)
  - LRU (Least Recently Used)
    - Addition of LRU bits
    - How many are needed?
    - What is the complexity?
    - Better scheme?
  - Pseudo-LRU

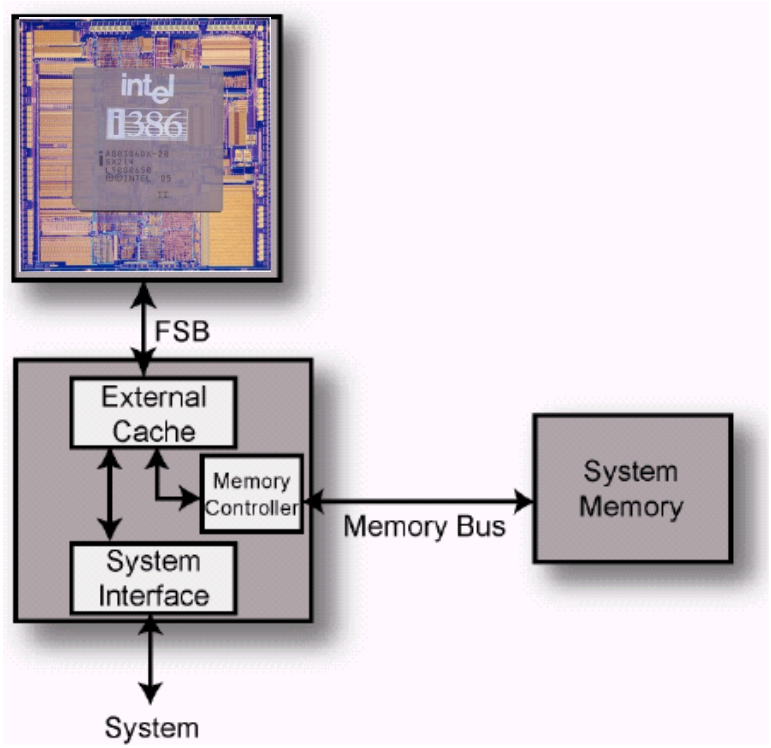


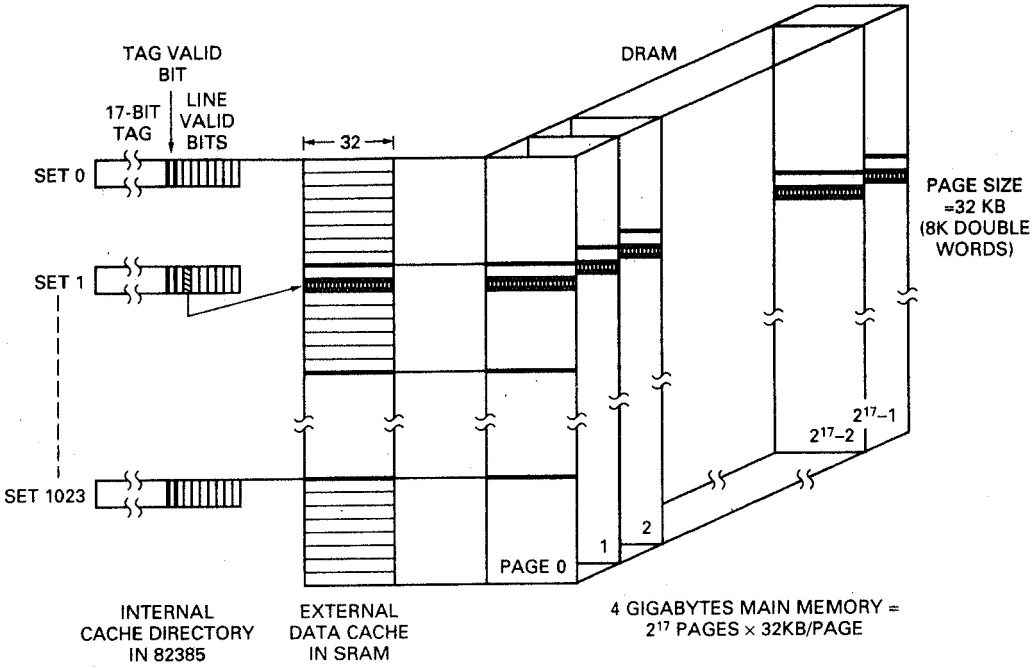
Alternative Cache Organizations

- What is the structure of cache line
  - Block of bytes (indexed by byte offset), words
  - Tag, valid, dirty, cache coherence bits for entire line
- Sector Caches
  - Cache lines consists of sub-blocks
  - Tag/valid for cache line + individual valid/status bits for sub-blocks
  - Why were these decisions made this way?
    - Cache cost (minimize tag SRAM)
    - Exploit spatial locality (share tags for adjacent 8 words)
    - No page mode DRAMs so time consuming to fill all 8 words while CPU stalled
  - Older examples
    - IBM System 360/85 (first commercially available system with cache)
    - Intel 80386 (off-chip cache) (Doug Hall's book)
  - Newer examples
    - IBM Power4 and Power 5 L1 instruction cache and L3 caches
      - CMP (Chip Multiprocessors): two cores each with 2-way SMT (Simultaneous Multithreading)
      - L1 instruction cache line 128 bytes (four sectors) -- why?
      - L3 cache line 512 byte lines – why?

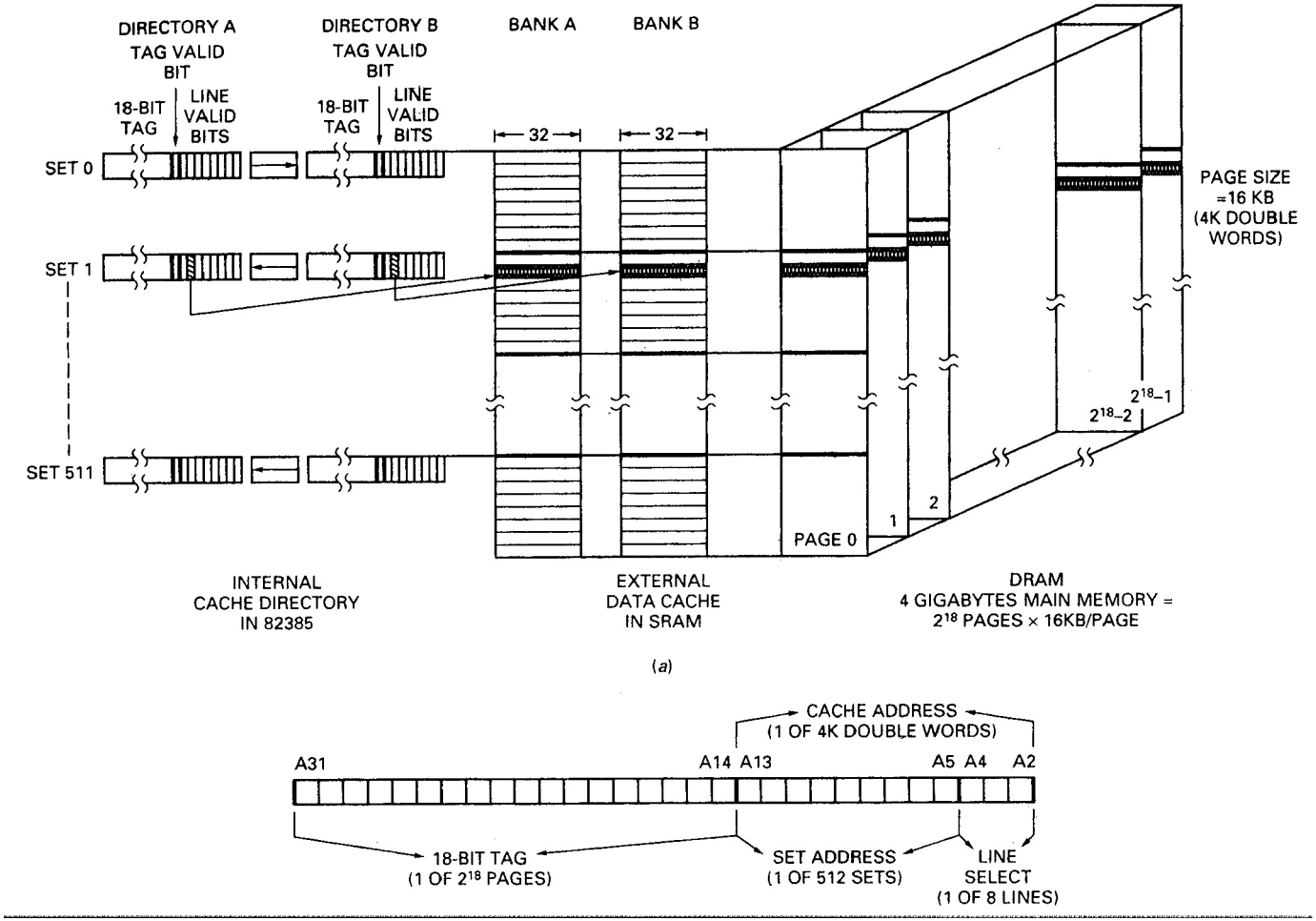


Direct Mapped Sector Cache





Two Way Associative Cache



Measuring Cache Performance

- Hit
  - Desired item is in the cache
- Miss
  - Desired item is not in the cache
  - Request from next level in storage hierarchy
- Hit Rate
  - Probability an item is in the cache
- Hit Time
  - Time to deliver the item if it's in the cache
    - Determine hit/miss, access cache
- Miss Rate
  - Probability an item isn't in the cache (1-Hit Rate)
- Miss Penalty (Time)
  - Time to deliver the item if it's not in the cache
- Performance -- not just hit rate!

Average Memory Access time (AMAT) =
(Hit Rate × Hit Time) + (Miss Rate × Miss Time)

An Example

- Assume
  - DRAM access time is 100 cycles
  - Cache access time is 2 cycles
    - To determine if data is in cache (and return it if present)
  - 90% hit rate

Memory Access time without cache

= 100 cycles

Average Memory Access time (AMAT) with cache

= (Hit Rate × Hit Time) + (Miss Rate × Miss Time)

= (0.90 × 2) + (0.1 × (2+100))

= 12 cycles

Measuring Cache Performance

- Sometimes use “misses/1000 instructions”

Improving Cache Performance

- Reduce Miss Rate
  - Larger line size
  - Larger cache size
  - Higher associativity
  - Victim caches
  - Hardware prefetch
  - Software optimizations
- Reduce Miss Penalty
  - Multilevel caches
  - Prioritizing reads over writes
  - Early restart
  - Critical word first
  - Non-blocking caches (throughput/bandwidth)
- Reduce Hit Time
  - Separate instruction and data caches (may also reduce miss rate)
  - Way prediction
  - Trace caches (may also reduce miss rate)
  - Pipeline cache access (throughput/bandwidth)
  - Multiple banks (throughput/bandwidth)
  - Avoiding address translation when indexing cache
    [Will cover with virtual memory]

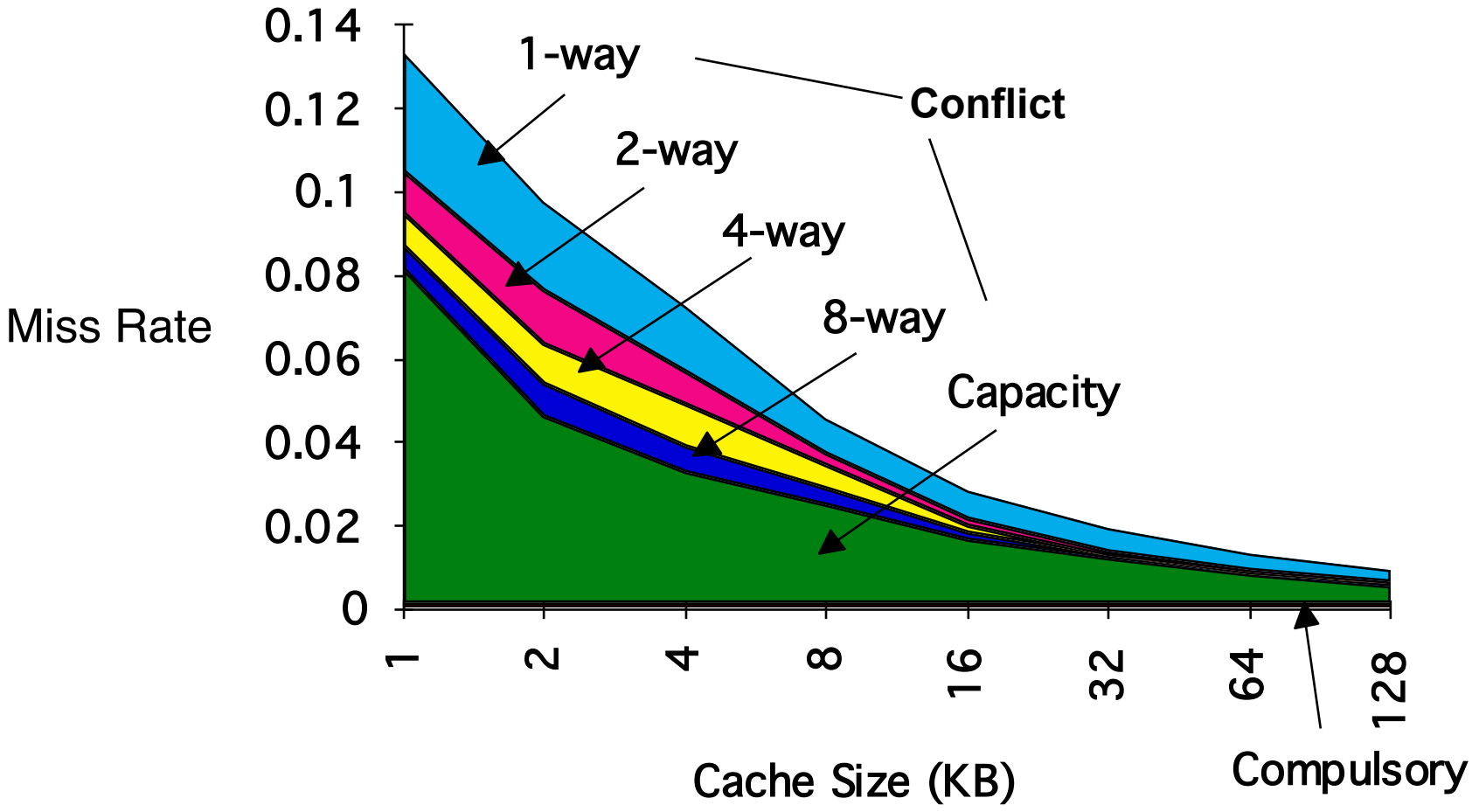
Reduce Miss Rate

- Increase size of cache → increase hit time, power
- Increase associativity → increase hit time
- Increase size of cache line → increase miss penalty

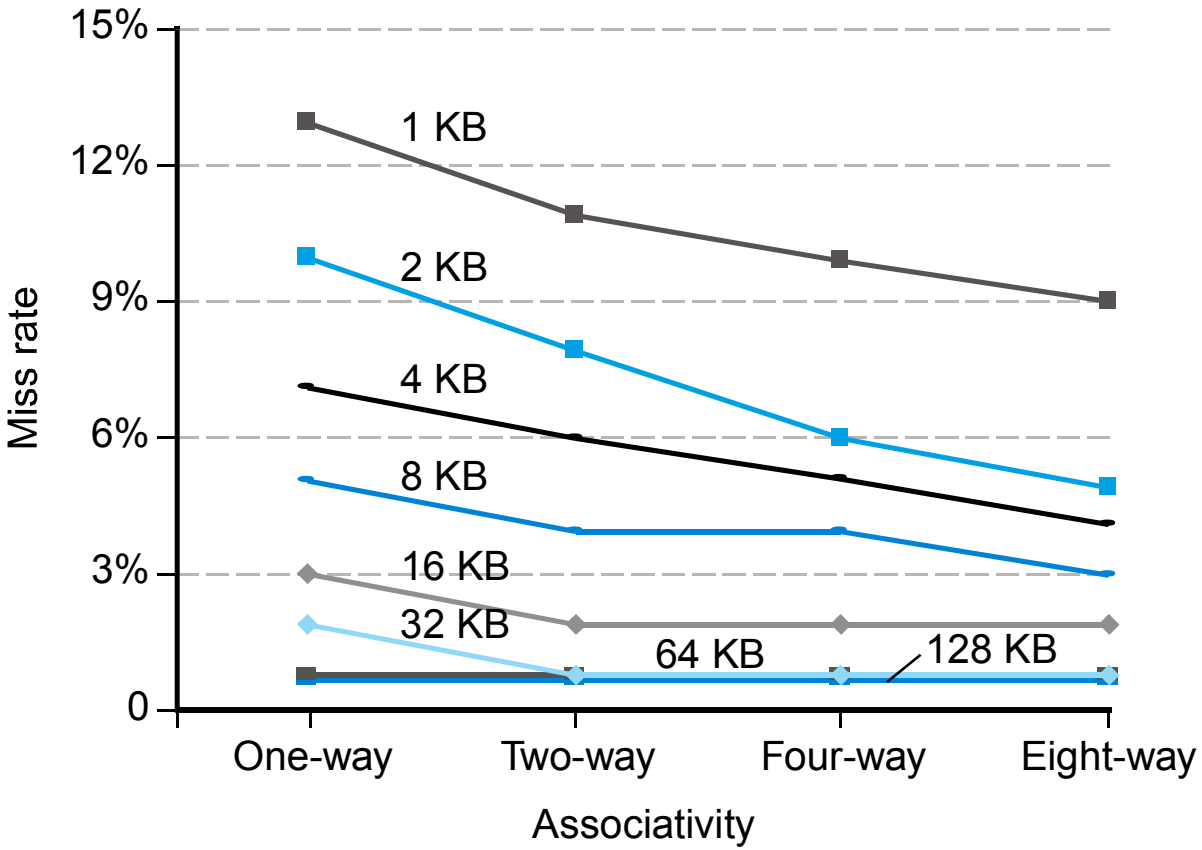
→ Trade-offs!!

- Victim cache
  - Small fully associative cache of replaced lines
- Hardware pre-fetch
  - If available memory bandwidth
- Software optimizations
  - How you write your code can affect cache performance

Conflict, Capacity, Cache Size

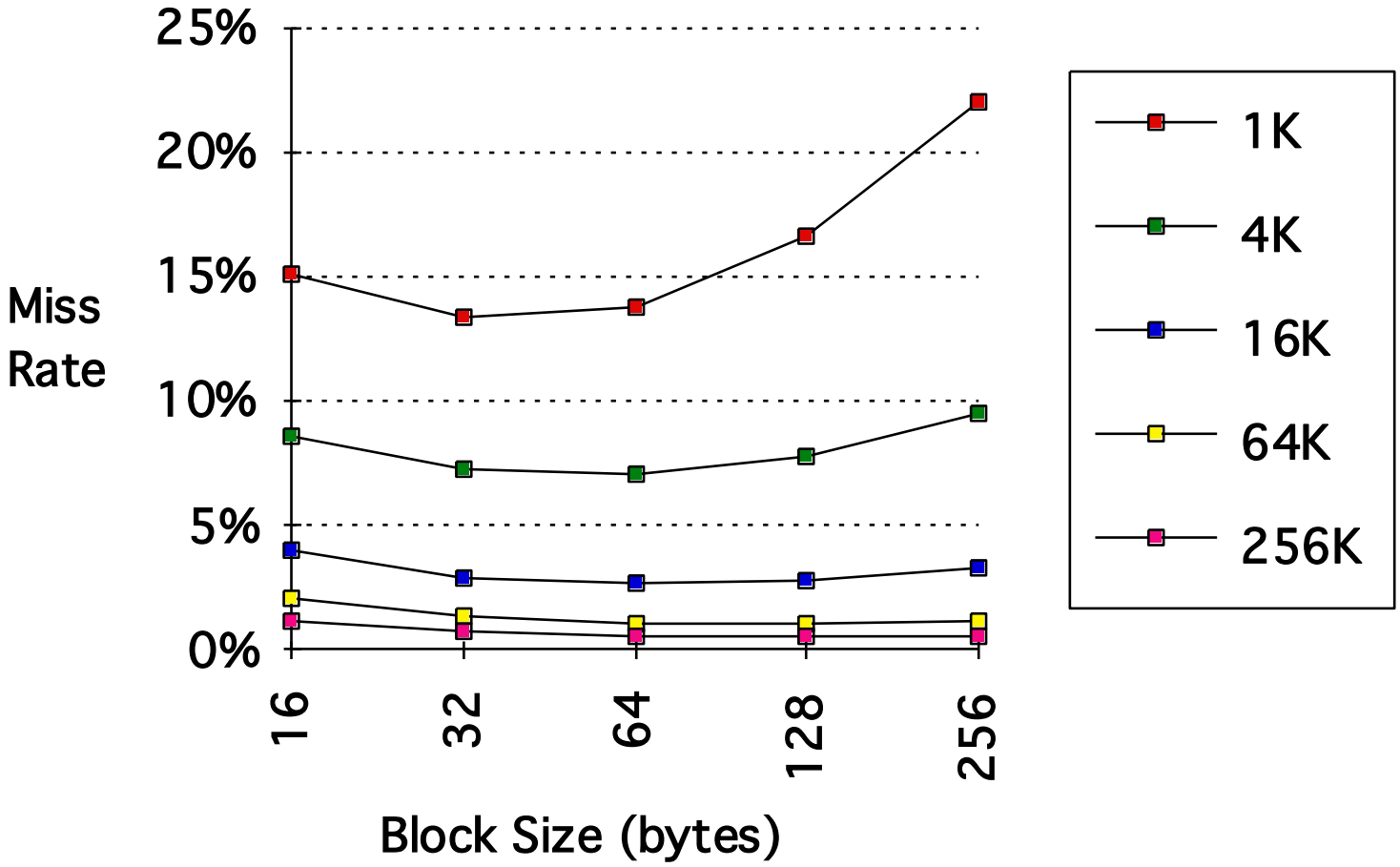


Performance



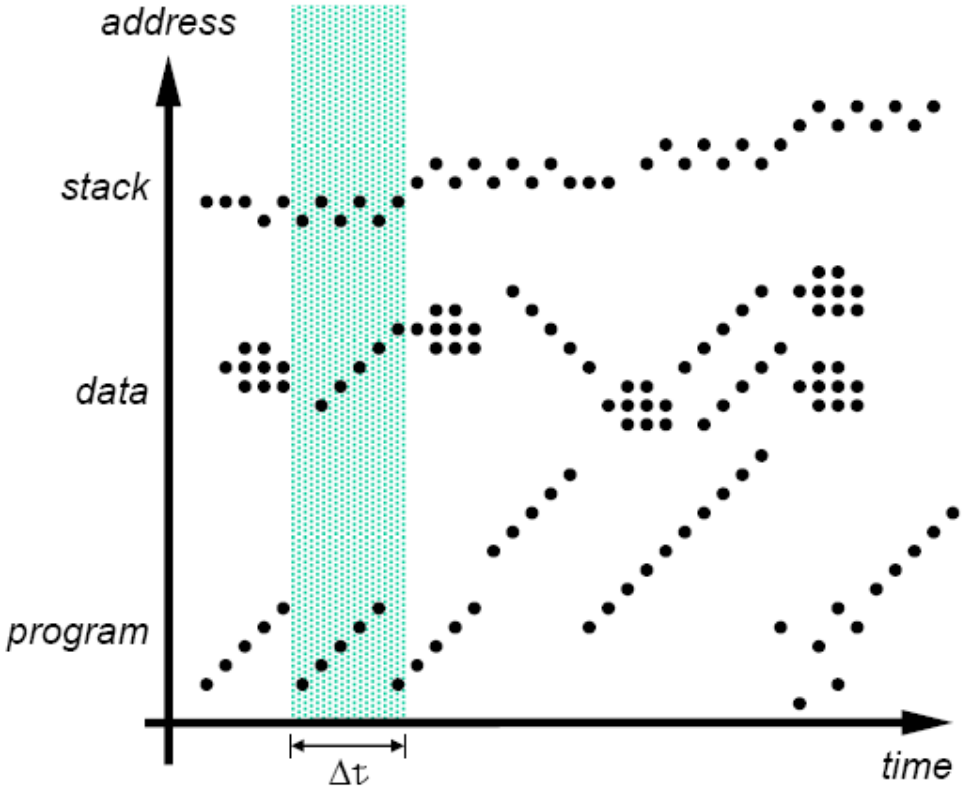
The smaller the cache, the greater the impact of associativity. Why?
→ more "aliasing"

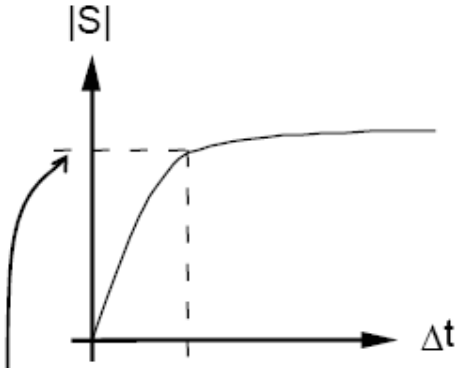
Larger Line Size



Why does the miss rate trend up at the end?
Hint: How many cache lines are there: 1K/256 = 4

Working Set





S is the set of locations accessed during Δt.

Working set: a set S which changes slowly w.r.t. access time.

Working set size, |S|

Software Optimizations

- Programmers, compilers, post-processors can play a role
- Ensure basic blocks start at beginning of cache line
- Access arrays to ensure cache hits ("stride")

```
for (i = 0; i<N; i++)
    for (j = 0; j<N; j++)
        A[i,j] = c * A[i,j];
```

[0,0] [0,1] [0,2] [0,3]
[1,0] [1,1] [1,2] [1,3]
[2,0] [2,1] [2,2] [2,3]
[3,0] [3,1] [3,2] [3,3]

row-major order
C Language

[0,0]
[0,1]
[0,2]
[0,3]
[1,0]
[1,1]
[1,2]
[1,3]

(1,1) (1,2) (1,3) (1,4)
(2,1) (2,2) (2,3) (2,4)
(3,1) (3,2) (3,3) (3,4)
(4,1) (4,2) (4,3) (4,4)

column-major order
Fortran, MatLab

(1,1)
(2,1)
(3,1)
(4,1)
(1,2)
(2,2)
(3,2)
(4,2)

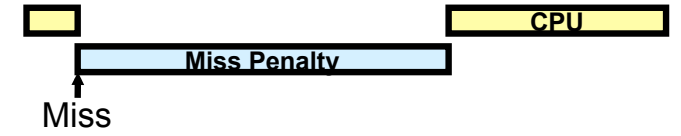
Reduce Miss Penalty

- Add another level of cache
- Prioritizing reads over writes
- Early restart
  - Restart CPU immediately after obtaining desired bytes from memory
  - Continue to fill cache line
- Critical Word first
  - Prioritize order of bytes returned in line
  - 64 byte line, but 4 may have been those requested
  - Facilitated by SDRAM burst ordering
- Use non-blocking (lockup-free) caches
  - Permit additional cache requests even while handling a cache miss
    - "hit under miss" allows access following cache miss if it's a hit
    - "miss under miss" allows even a miss to follow a cache miss
    - For data, not instruction caches (why?)
  - When CPU handles "out of order" requests/response (superscalar)
  - Makes AMAT less appropriate performance metric for cache
  - Complicates cache controller

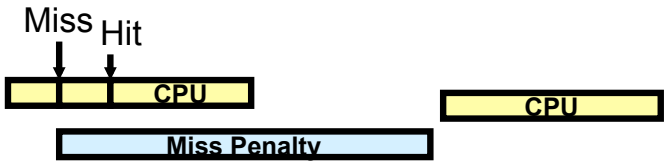
Non-blocking caches

- Idea
  - Allow hits while serving a miss ("hit under miss")
  - Allow misses while serving a miss ("miss under miss")
- When applicable
  - Superscalar processors (capable of >1 outstanding load/store)
  - Shared cache (shared by >1 core)
  - When lower level cache can have >1 pending access
    - Multibanks, pipelined
- Issues
  - Handling multiple misses
  - Misses may be satisfied out of order (DRAM optimization)
  - Handling load/store requests to pending misses

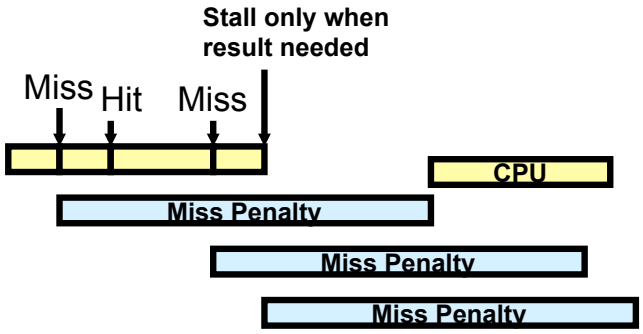
Non-blocking caches



Stall CPU on miss



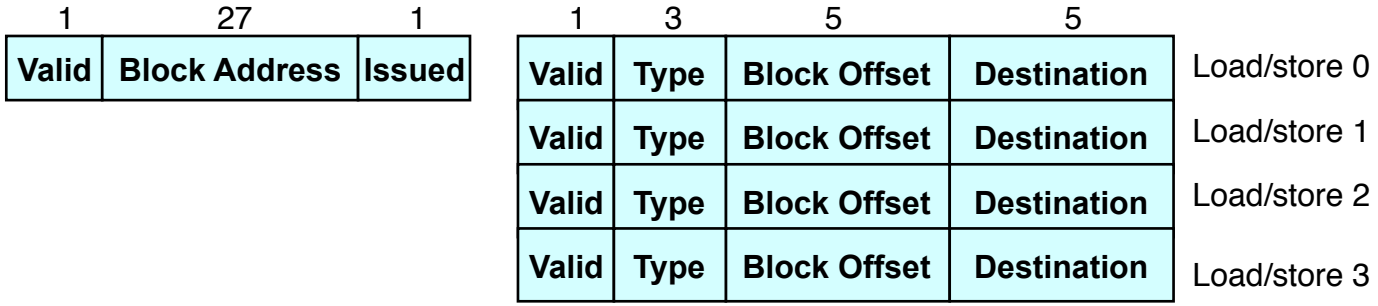
Hit under miss



Multiple out-standing misses

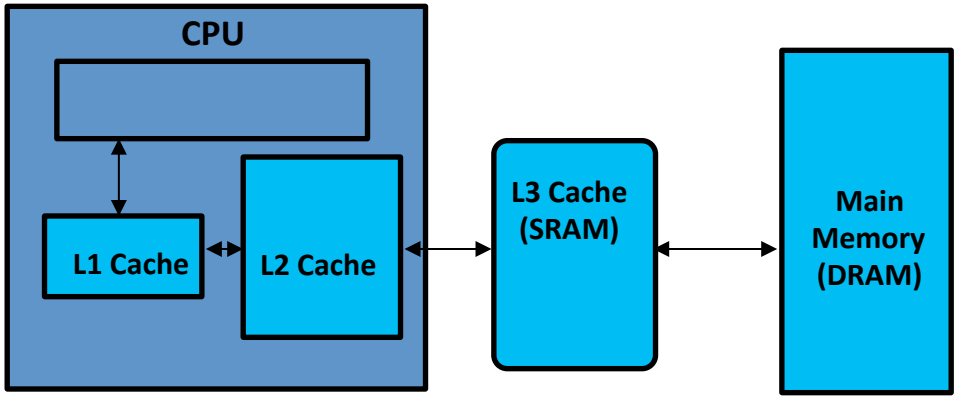
Non-blocking caches

- Miss Status Handling Register (MSHR)
  - Allocated for each (new) outstanding miss to a cache line
  - Tracks outstanding miss
  - Pending loads/stores to cache line
- Maintain
  - Valid bit to indicate MSHR is valid (in use)
  - Block (line) address (address of cache line)
  - Issued bit to indicate load/store has been issue to memory
  - Outstanding loads/stores
    - Valid
    - Type (load/store) and size (e.g. byte/word/double word)
    - Block offset (byte offset in line)
    - Destination (load/store buffer number or actual space for data)



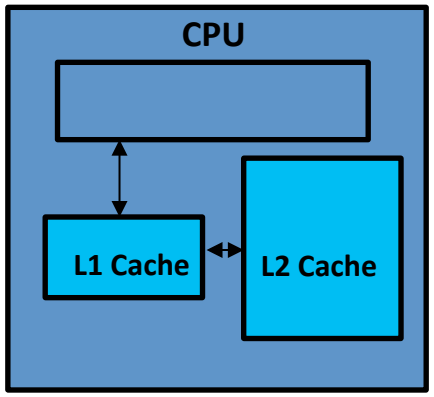
Multilevel Caches

- Multilevel Inclusion
  - Any item in L_i cache will also be in L_{i+1} cache
    - Simplifies cache coherence algorithms
    - Can preserve consistency by checking L2 cache, avoid contention with CPU for L1 cache (increasing importance for SMP)
    - Power conservation – power down underutilized portion of cache, still available in next level
    - Complications if lines size not same
    - Intel Pentium 4 L1 uses 64-byte lines, L2 uses 128-byte lines
- Multilevel Exclusion
  - Any item in L_i cache will never be found L_{i+1} cache
    - If L2 cache is not much larger than L1 why waste space on redundant copies?
    - AMD Opteron uses 2 64KB L1 caches and 1MB L2 cache
- Often use WriteThrough policy for L1 cache because L2 or L3 can backup



Performance in Multilevel Caches

Average Memory Access Time (AMAT) = (HitRate x HitTime) + (MissRate x MissTime)



$HitRate_{L1}$ = 95%
$HitRate_{L2}$ = 80%
$HitTime_{L1}$ = 1 cycle
$HitTime_{L2}$ = 5 cycles
$MissTime_{L2}$ = 100 cycles

Average Memory Access Time (AMAT)

= $(HitRate_{L1} \times HitTime_{L1}) + MissRate_{L1} \times (MissTime_{L1})$

= $(HitRate_{L1} \times HitTime_{L1}) + MissRate_{L1} \times [HitRate_{L2} \times HitTime_{L2} + MissRate_{L2} \times MissTime_{L2}]$

0.95 x 1 + 0.05 x [0.8 x (1+5) + 0.2 x (1+5+100)] = 2.25 cycles

Prioritizing Reads over Writes

- Assume
  - Direct-mapped, write-through cache
  - Memory addresses 512 and 1024 map to same cache index
  - Four word write buffer
  - Will R2 == R3?

```
SW R3, 512(R0)     ; M[512] ← R3        (index 0)
LW R1, 1024(R0)    ; R1 ← M[1024]       (index 0; cache miss; evicts M[512])
LW R2, 512(R0)     ; R2 ← M[512]        (index 0; cache miss; read from memory
```

  - Not if the write hasn't happened by the time 3rd instruction executes!
  - Two solutions:
    - Stall read until write buffer empty
    - Check contents of the write buffer on a read miss
      - Most desktop and server systems do this!
  - Another optimization: coalescing write buffer

Reduce Hit Time

- Separate instruction and data caches
  - In pipelined architecture allows two different stages to simultaneously access memory
    - Instruction fetch
    - Data operand read/write
  - Only require single-ported caches
  - Can optimize each cache separately (size, line size, associativity, policies)
  - Two smaller caches likely to have better hit times than one larger one
- Way prediction
  - Additional bits maintained with set to predict way of next cache access to that set
  - Single comparison is done on tag, valid bits for line in predicted way
  - Cache data read in parallel
  - If prediction accurate, lower hit time
  - If not, extra clock cycle (pipelined cache access)
  - Simulations suggest >85% accuracy for two-way associative caches
  - Good fit for speculative processors which must already be capable of undoing
  - Used in Pentium 4

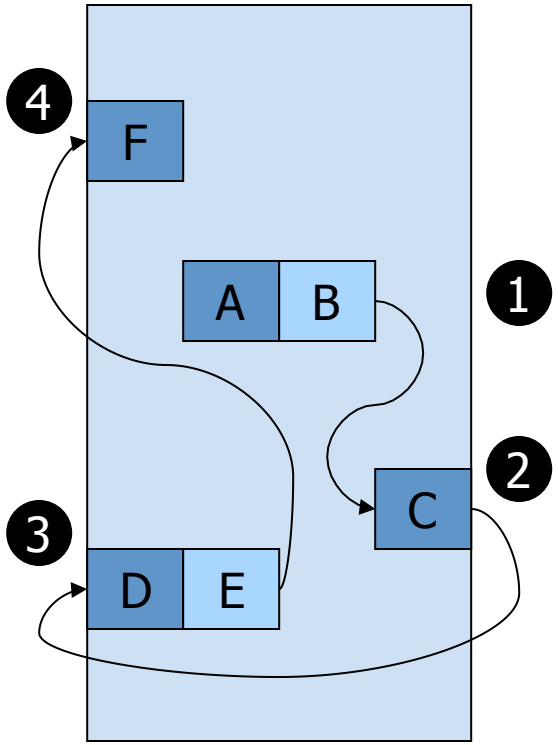
Reduce Hit Time

- Pipeline cache accesses
  - Pentium 1 single clock cycle instruction cache access
  - Pentium Pro – Pentium III took two cycles
  - Pentium 4 required four cycles
  - Bigger penalty for mis-predicted branches
- Multiple banks
  - Multiple issue (superscalar) processors may have more than one data access/cycle
  - Rather than multi-ported cache, partition cache into banks with independent access
  - AMD Opteron (L2 has two banks), SUN Niagara (L2 has four banks)
- Trace caches

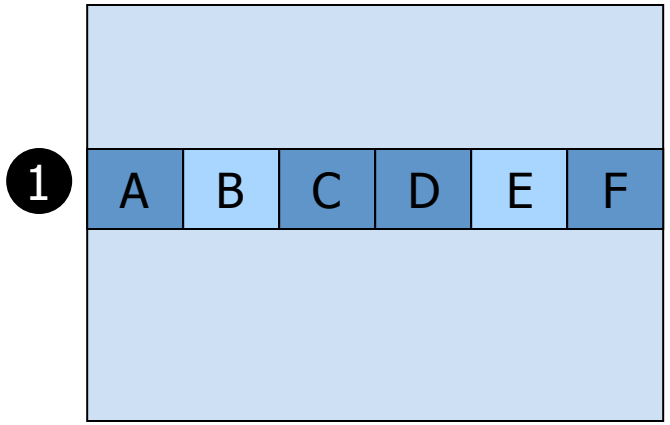
Trace Caches

- Traditional caches store (static) sequences of bytes in address order within cache line
  - Due to branching, there are wasted bytes in many cache lines
  - A branch to an address in interior of a cache line means bytes prior to that unused
  - A branch not on the last instruction in a cache line means bytes after that unused
  - Spatial over temporal locality (within cache line)
- Trace caches store (dynamic) sequences of instruction in execution order within cache line
  - As instructions are fetched they're also placed in a buffer (assembling the trace)
  - History of last n branches (take or not-taken) is maintained
  - When buffer full (or maximum predicted branches reached) it's written to the trace cache
  - Trace cache is indexed by PC and (predicted) branch information
  - Temporal over spatial locality (within cache line)
- Expensive in area, power, complexity
  - Not widely used
  - Used in Pentium 4 (stored decoded μops)
  - Not used in Nehalem (LSD: Loop Sequence Detector)
    - Avoids re-decoding instructions in highly used loops – primarily power optimization

Trace Caches



Traditional instruction cache



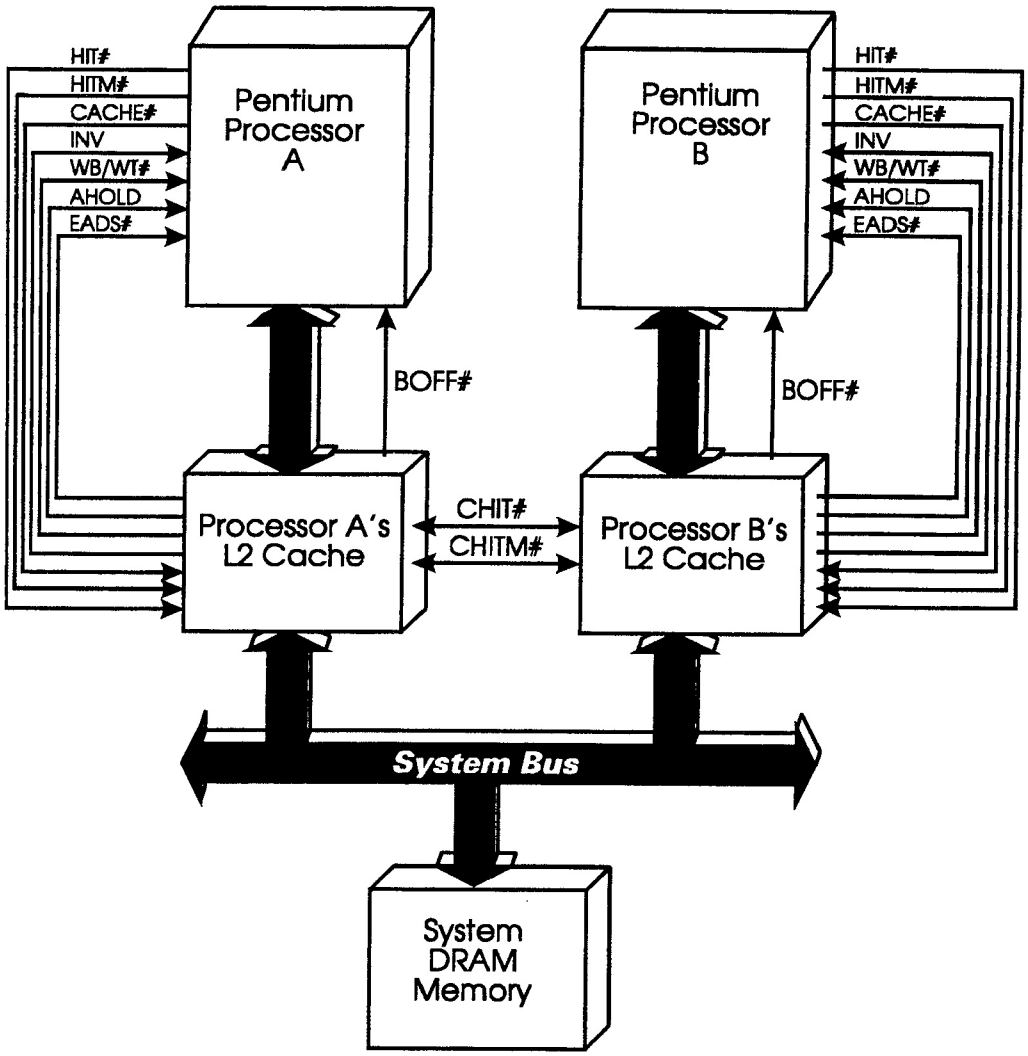
Trace cache

{PC, T, T} = PC (address of instruction A), branch B (predicted) taken, branch E (predicted) taken

Cache Coherence

- Normal operation may permit an item to be "stale"
  - Cache/memory
    - Needed in case of Write Back caches
    - Dirty bit suffices
  - L1/L2/L3 Caches
    - "Inclusivity" if next level caches are a superset of higher level caches
    - L_i cache needs to notify L_{i-1} cache when victim removed from cache
      - May require writeback to L_i cache from L_{i-1} cache
  - Single processors with multiple bus masters
    - e.g. DMA
  - Multiple processors sharing memory, but with own caches
- How do we ensure that the cache is "coherent"?

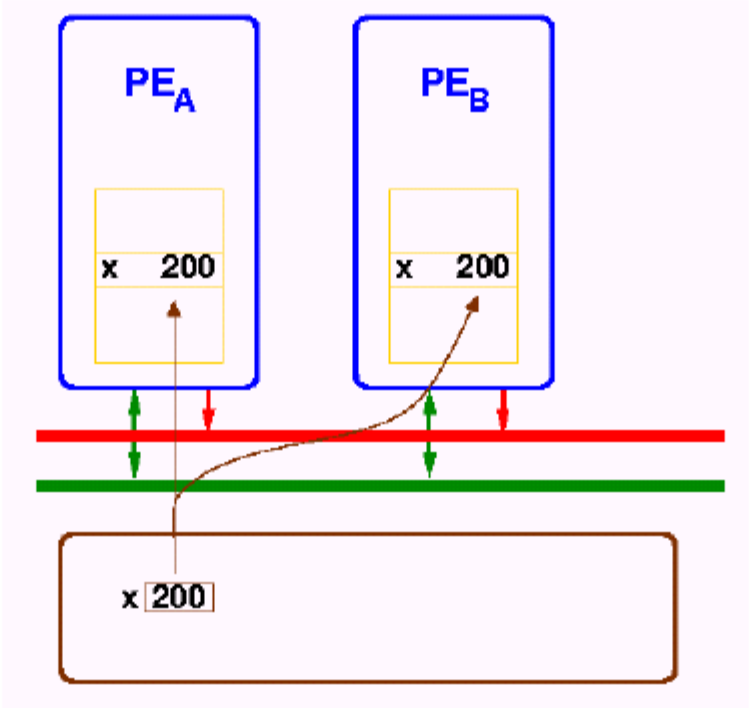
Shared Memory Multiprocessor



The Problem

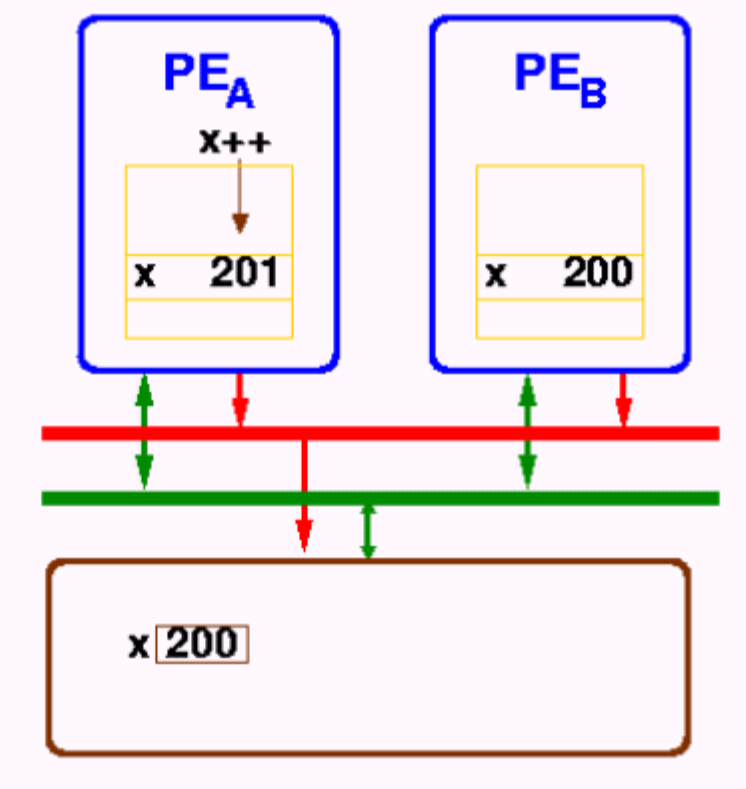
When multiple processors with on-chip caches are placed on a common bus sharing a common memory, then it's necessary to ensure that the caches are kept in a **coherent** state.

- PE_A reads location x. Copy of x transferred to PE_A's cache.
- PE_B also reads location x. Copy of x transferred to PE_B's cache too.



The Problem

- PE_A adds 1 to x. x is in PE_A's cache, so there's a cache hit.
- If PE_B reads x again (perhaps after synchronising with PE_A), it will also see a cache hit. However it will read a stale value of x.



The Solution

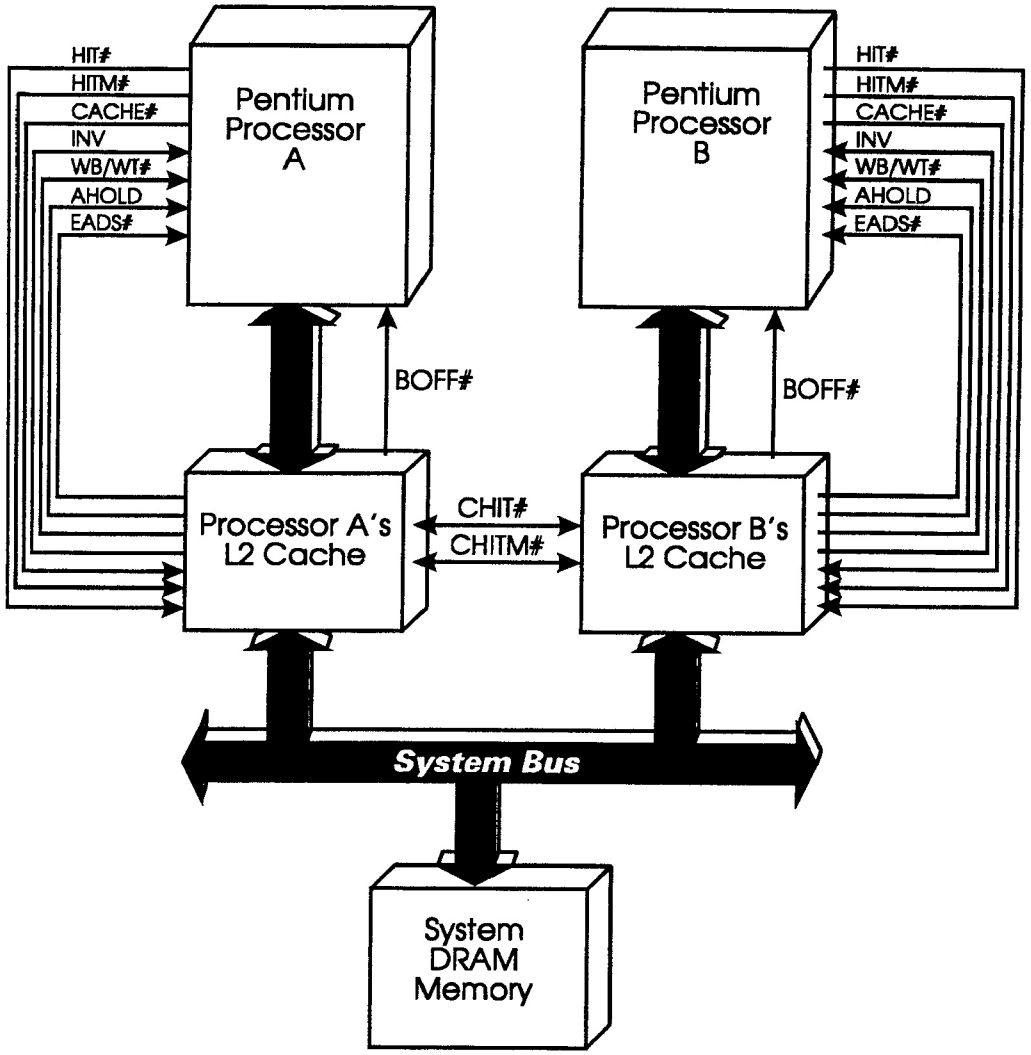
- All processors "snoop" on memory transactions
- Processors signal if they have a cached copy
- All processors implement same cache coherence policy
- Several variations in use
  - MEI, MESI, MOESI
- Most modern processors employ one of these
  - Intel Pentium, AMD, SunSparc, PowerPC
- MESI (Intel Pentium, PowerPC)
  - Modified/Exclusive/Shared/Invalid
  - 4 states for each cache line (instead of 2)

Shared Memory Multiprocessor

HIT# -- Snooping processor indicates it has copy in its cache (open drain, shared line)

HITM# -- Snooping processor indicates it has copy in its cache and it's been modified (open drain, shared line)

Memory Controller also sees HIT#, HITM# (so it won't service a read request since valid data is held not in memory but in a cache which needs to be written back to memory first).



MESI Protocol

Cache coherence hardware

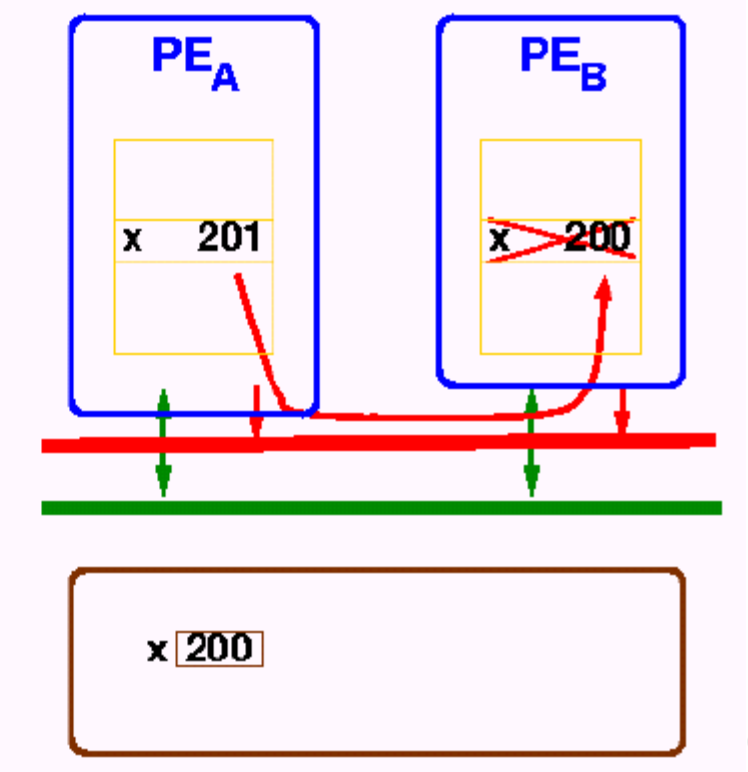
This problem is avoided by adding **snooping** hardware to the system interface. This hardware monitors the bus for transactions which affect locations cached in this processor.

The cache also needs to generate **invalidate** transactions when it writes to shared locations.

- When PE_A updates x, the cache generates an **invalidate** transaction.
- When PE_B's snooping hardware sees the invalidate x transaction, it finds a copy of x in its cache and marks it invalid.
- Now a read x by PE_B will cause a cache miss and initiate a databus transaction to read x from main memory.

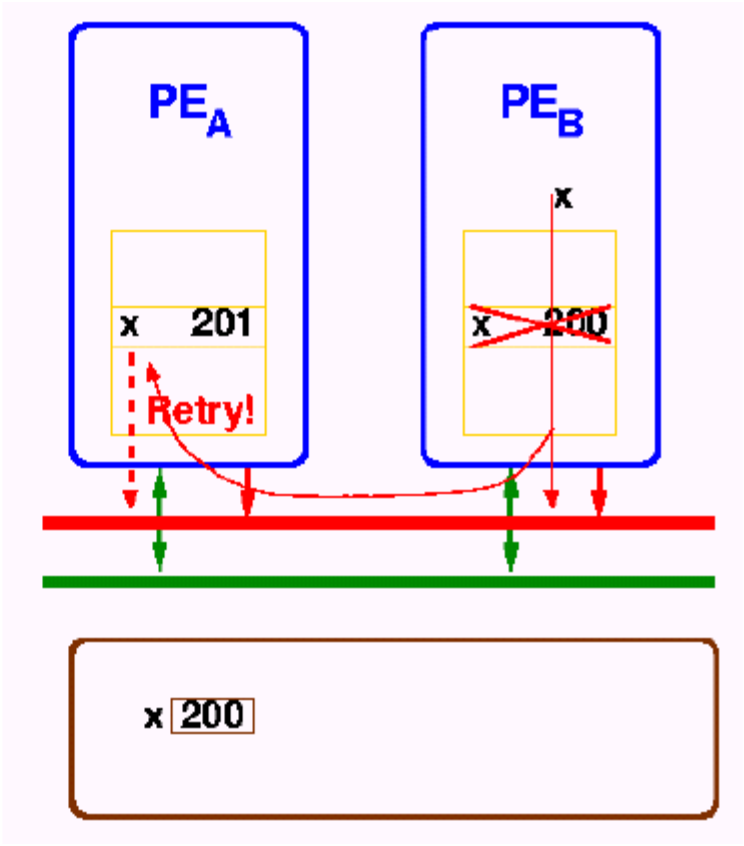
Note:

1. The invalidate transaction is an **address-only** transaction: it simply communicates the address of a cache line which has been invalidated to all the other processors. This is to save data-bus bandwidth: there's a possibility that no other cache holds x now. Even though it had been read by other PE's and held in their caches, it has since been replaced by them.
2. We are assuming that the cache is operating in **write-back** mode. So the updated value of x is held in PE_A's cache until that cache line is needed for something else, triggering a write-back of x to main memory.



MESI Protocol

- When PE_A's snooping hardware sees the memory read for x, it detects the modified copy in its own cache, and emits a retry response, causing PE_B to suspend the read transaction.

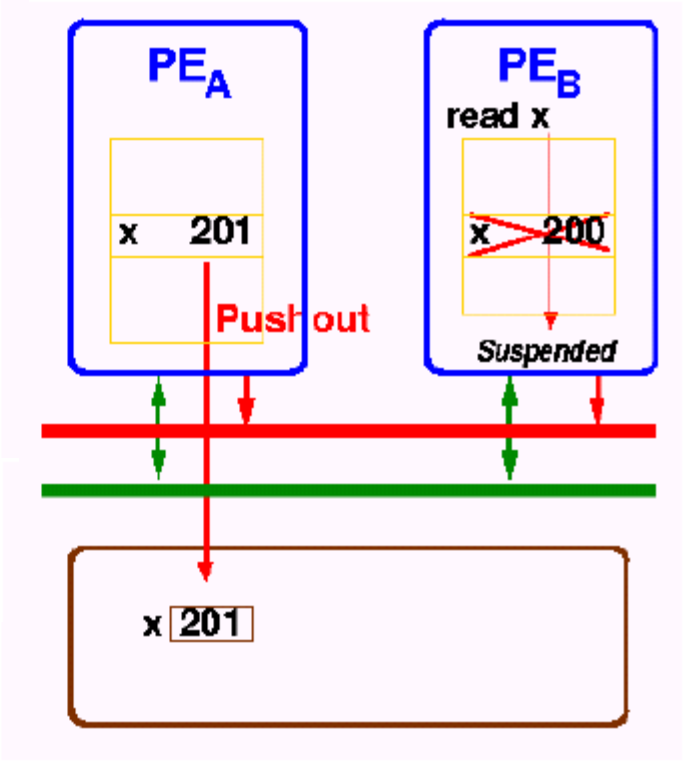


MESI Protocol

- PE_A now writes (flushes) the modified cache line to main memory.
- PE_B continues its suspended transaction and reads the correct value from main memory.

Note:

1. Some systems will permit cache-to-cache transfers (with or without simultaneous write to main memory). Although this would seem to be an obvious improvement - allowing PE_B to continue faster. The saving turns out to be minimal and so is not universally implemented.



MESI (Write Back Cache)

- Modified
  - The cached copy is the only valid copy of the line
  - No other processor caches contain the line
  - The memory copy is out of date
- Exclusive
  - No other processor has a copy of the cached item
  - The processor's cache and memory are identical
- Shared
  - At least one other processor has a copy of the line
  - The cached copies and memory are identical
- Invalid
  - The cache entry is invalid (doesn't hold a copy of the line)

CPU Write

- Cached item (Shared/Exclusive/Modified)
  - Employ Write Policy
    - Write Through
    - Write Back
- Non-cached item (Invalid)
  - Cache controller needs to do a memory read!
  - But let other processors' caches know intention
- So when are real memory writes done?
  - On Write Back (when line is evicted) or Write Through

RWITM, MRI, RFO

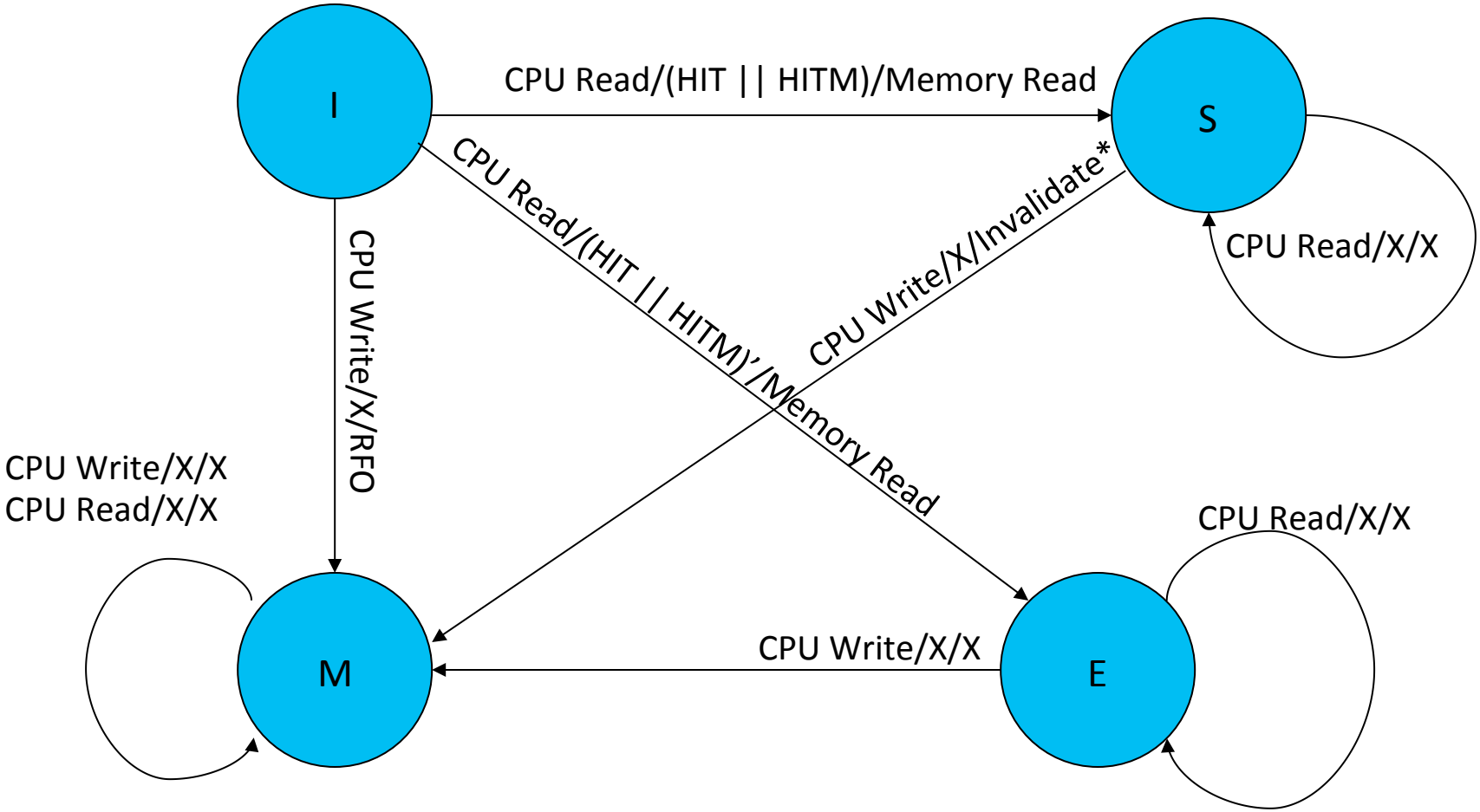
- Read With Intent to Write (RWITM)
- Memory Read and Invalidate (MRI)
- Read For Ownership (RWO)

Synonyms

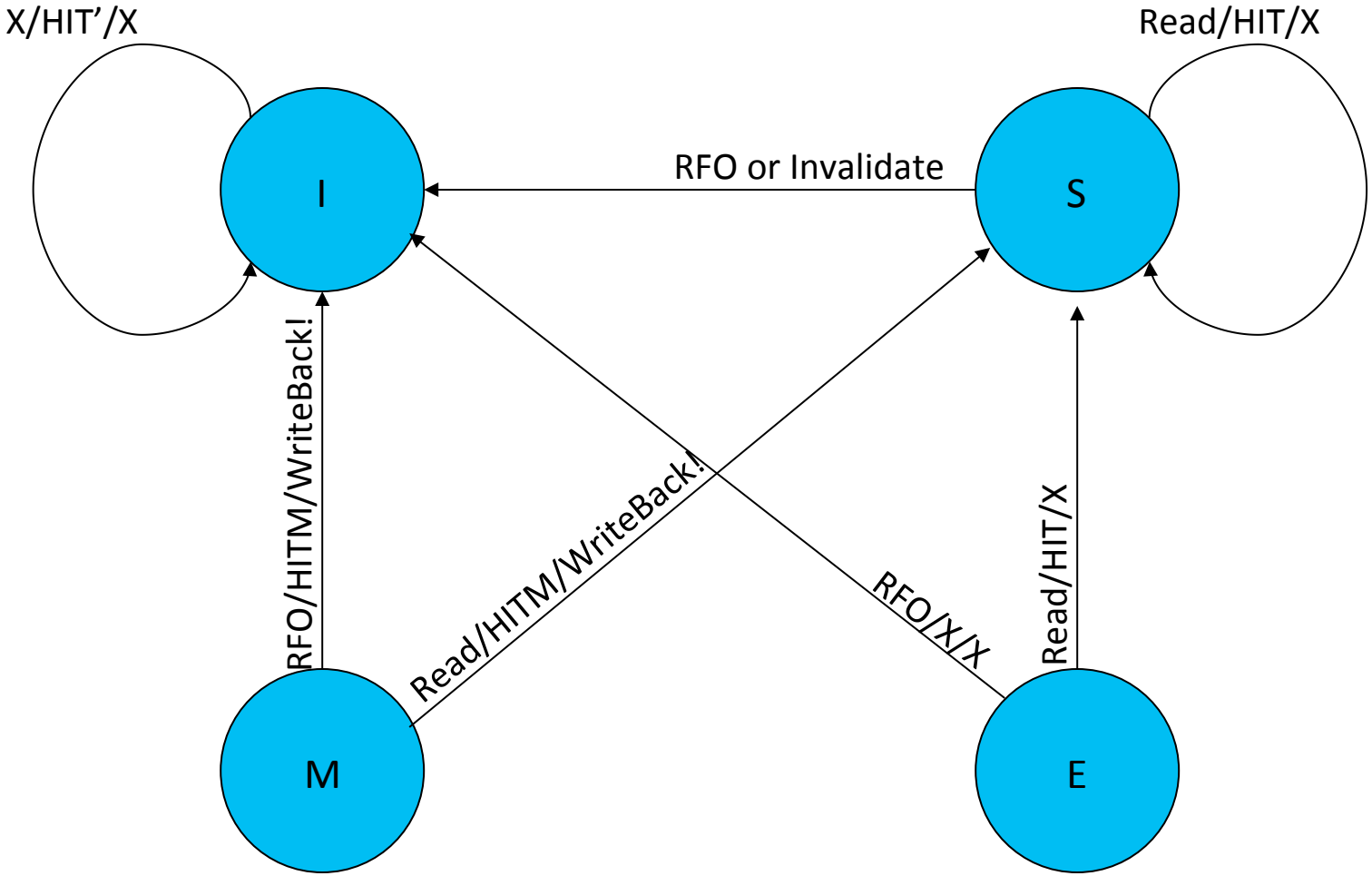
  - Do a memory read but indicate to snooping processors on FSB our intent to write that memory location

- Invalidate
  - "Invalidate your cached copy. I have the only valid one now"
  - Address only transaction
    - Saves memory bandwidth
    - Puts address on FSB but doesn't request data from memory

MESI (WB) State Transitions for Cache Controller of Processor Accessing Memory



MESI (WB) State Transitions for Cache Controller of Snooping Processor

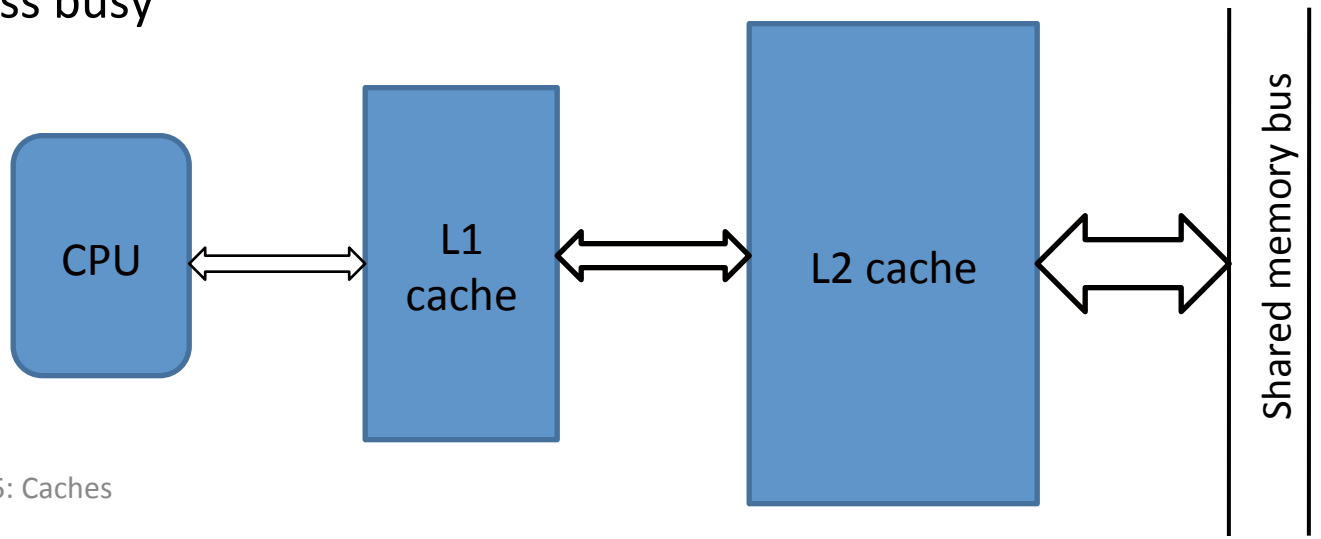


Source of Data When HITM

- In Pentium and earlier processors, a snooping cache controller detecting a hit on a modified line would "back off" the master, write back the dirty cache line, mark its copy shared, and then allow the other processor to complete its read
- In Pentium II processors any processor with a snoop hit on a modified line sources the line itself and marks it shared. Benefit: requires fewer FSB cycles.

Inclusivity Property

- A multilevel cache exhibits the inclusivity property if every lower level cache (farther from the CPU) contains a superset of the data in the next higher level cache (closer to the CPU)
- Purpose is to allow the lower level cache (closest to a shared memory bus) to be searched during snoops without involving the higher level cache unless the line is present in the higher level cache. This allows the higher level cache to be more available to the processor, off-loading snoop-related cache look ups to the lower level cache which is less busy

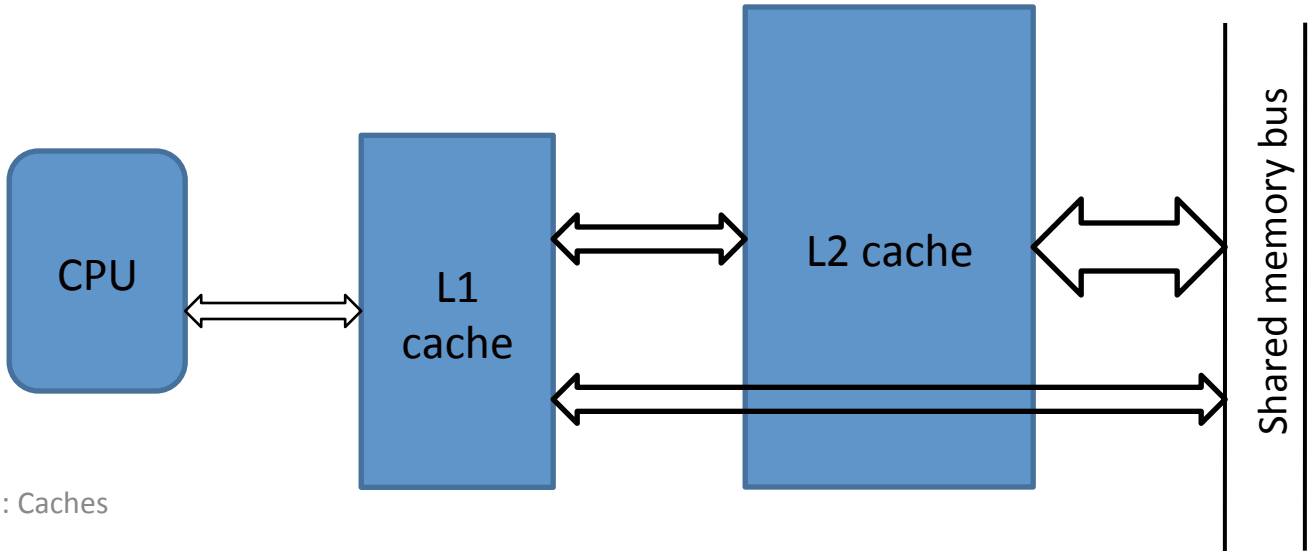


Maintaining Inclusivity Property (IP)

- Constraints: Single processor, L1 Writeback policy (trivial if L1 is WT)
  - If L1 line size = L2 line size require only
    - Size L2 > Size L1
    - L2 associativity >= L1 associativity
- On miss to L1 and L2, line is placed in both caches
  - If a dirty line is evicted from L1 to make room it will be written back to L2 (where it exists due to IP)
    - If dirty, it must be written back to L2 first before L2 then writes it back to memory
  - If a line in L2 is evicted to make room, the same line may be in L1 and must be evicted (to maintain IP)
- On miss to L1 and hit to L2
  - If a dirty line is evicted from L1 to make room it will be written back to L2 (where it exists due to IP)
- On a snoop hit to L2, L2 instructs L1 to invalidate the line in its cache
  - Minimize unnecessary invalidates by keeping single bit/line in L2 to indicate in L1
- Can relax some constraints
  - If allow L2 line size = c * L1 line size (where c integer >=1) (need additional inclusion bits if c > 1)
    - L2 associativity >= L1 associativity insufficient
    - In fact, require Associativity L2/Line Size L2 >= Associativity L1/Line Size L1

Exclusivity Property

- A multilevel cache exhibits the exclusivity property if every lower level cache contains only data not found in a higher level cache.
- Effectively just a large victim cache for lines evicted from next higher level cache
- Requires snoop lookups in higher level cache
  - Accomplished through redundant tag arrays
  - One used by processor requests, one for snoop lookups
  - Coordinate only when dirty bits set, on misses, successful invalidates



Review Cache Miss Types

- Compulsory (Cold Start)
- Capacity
- Conflict
- Coherence
  - True sharing
    - Read attempt to cached line that was invalidated due to another processor's write to the same (invalidating the line)
  - False sharing
    - Although in same cache line, bytes being read are not the same as bytes written by other processor – would not have been miss if line size had been different or had individual dirty bits

Alternatives

- MESI seems complicated. Why not…
  - Make all shared memory non-cacheable?
    - Performance!
  - Broadcast all writes → update (or invalidate) all caches?
    - Unnecessary memory access
      - e.g. writes to lines in Exclusive and Modified states
    - Need to cache items not likely to be used
- Snooping protocols aren't the only solution to coherence
  - Directory-based approaches
  - Scalability with number of processors (n)
    - Bus bandwidth

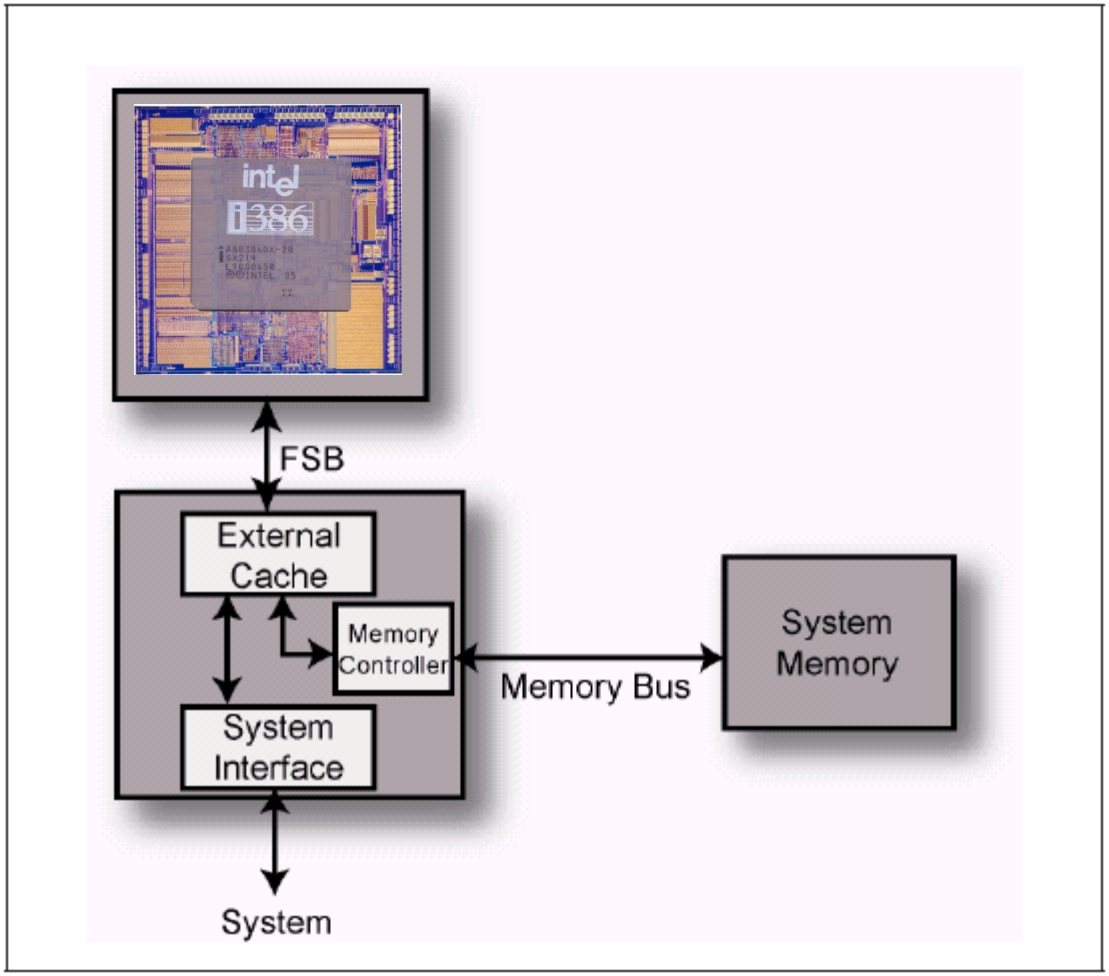
      Proportional to: n x (Cache Miss Rate + Invalidate Rate + WriteBack Rate)

    - Cache lookup bandwidth (for snooping)

      Proportional to: n x (Cache Miss Rate + Invalidate Rate)

Evolution of Caches in Intel IA-32 Architecture

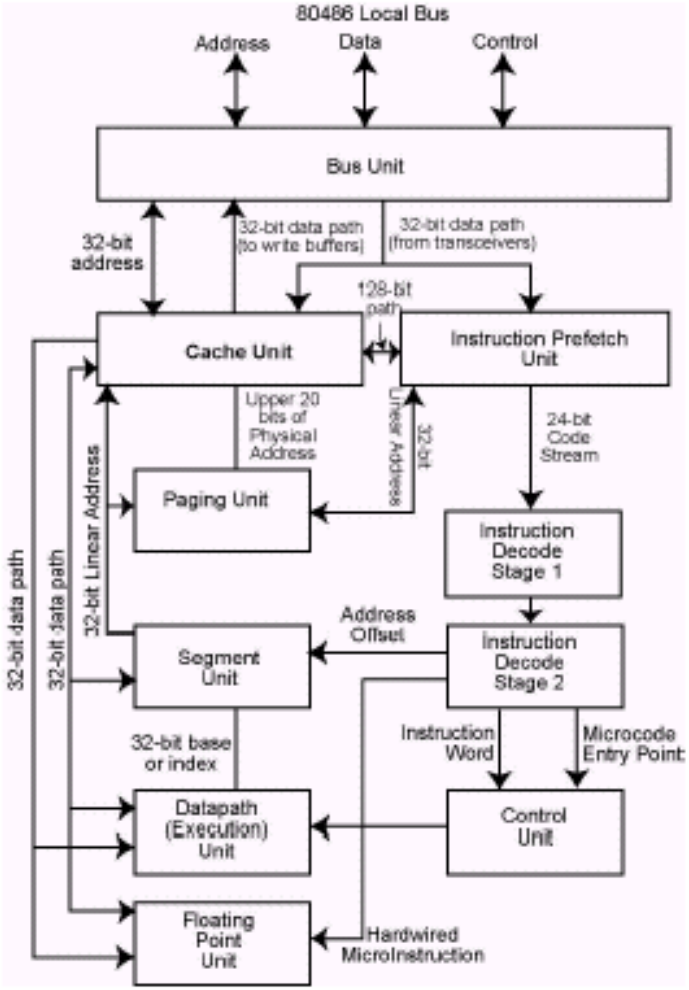
External Cache (386)



On-Chip Cache (486)

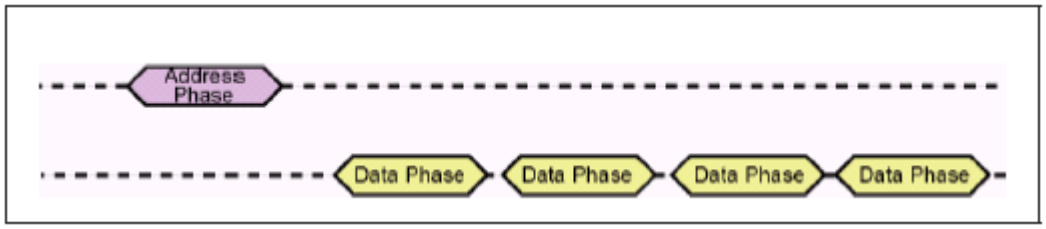
486 incorporated
L1 unified code/data cache

8K
4-way set associative
16 byte cache lines



Assume Use of 386 FSB Protocol

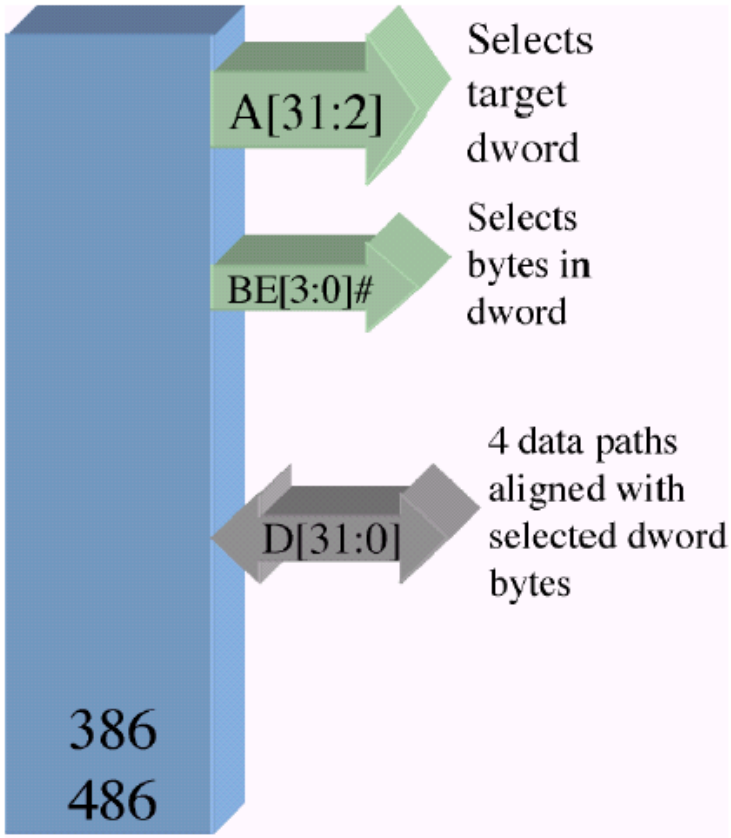
- 16 Byte Cache Line Read to DRAM
  - 4 separate dword (4 bytes) transactions
    - Arbitrate for bus ownership
    - Address dword
    - Await delivery of data
- New Burst Cache Line Fill transaction introduced in 486 (and subsequent IA-32 processors)



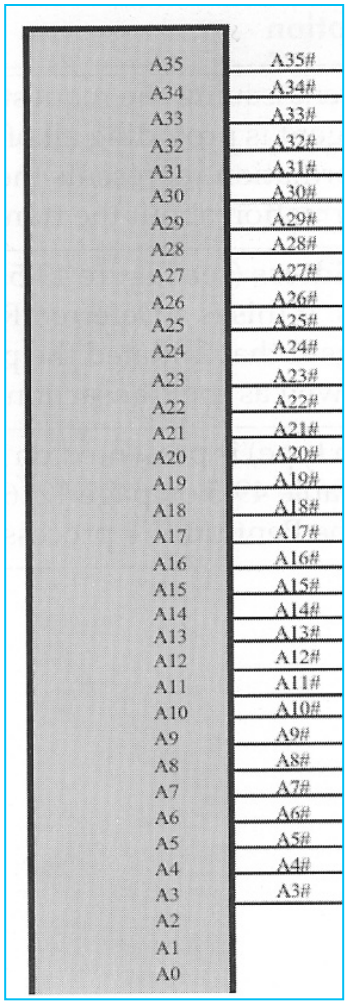
IA32 Memory Access

- Evolution of larger physical address spaces
  - 16→32→36
- Wider data buses
- Quad-word aligned memory access
- Burst mode memory reads

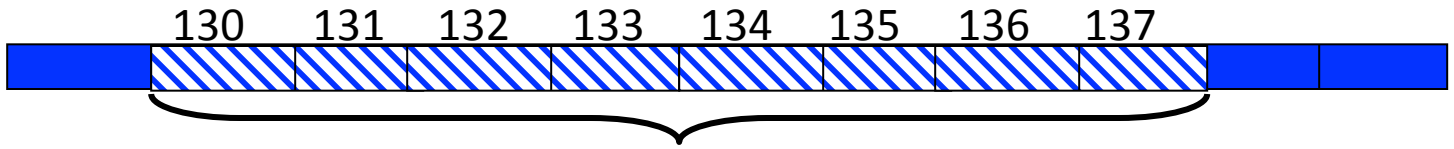
Processor	Line Size
486	16 bytes
Pentium P6 Family	32 bytes
Pentium 4, Xeon	128 bytes
Pentium M	64 bytes



Quad word aligned access



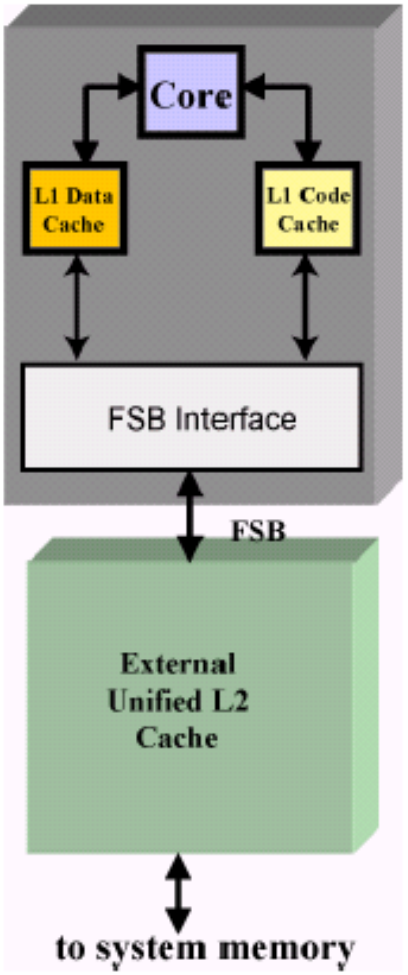
- 2^{36} (64 GB) physical address space
- Access quadword at a time (8 bytes)
- Don't need lower 3 address bits
- Need to specify which byte(s) interested in
- BE[7:0] (Byte Enables)



- Consider a reference to a mis-aligned quadword
  - E.g. 8 bytes beginning at location 134
  - Two quadword memory references
- Consider a reference to a mis-aligned word
  - E.g. 2 bytes beginning at location 137
  - Two quadword memory references

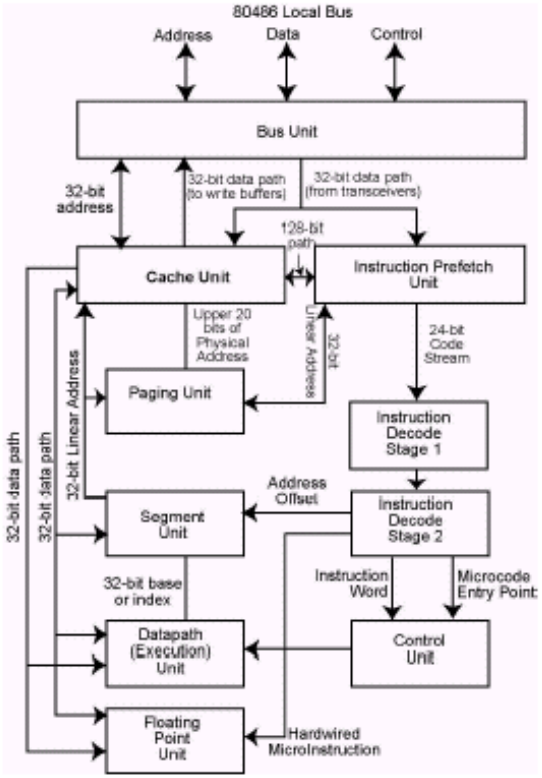
Add External Second Level (L2) Cache

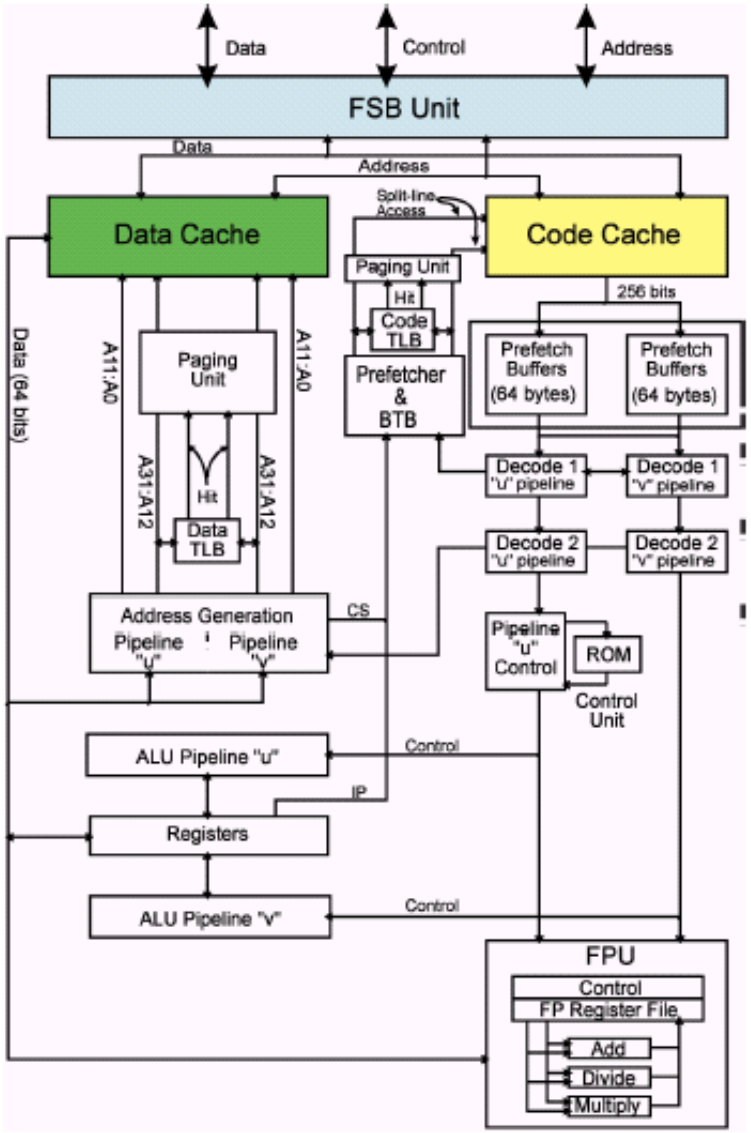
Full FSB Speed
(no wait states)



Separate Code and Data Caches

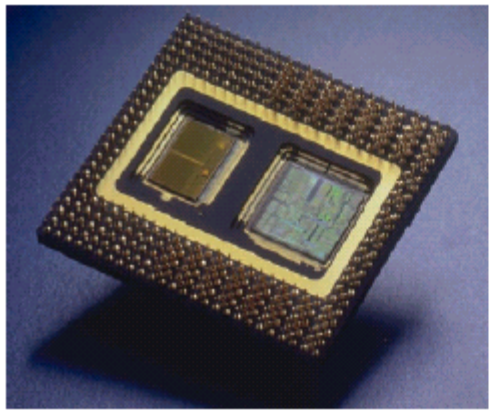
Execution Unit and Instruction Prefetch Unit contended for cache access…

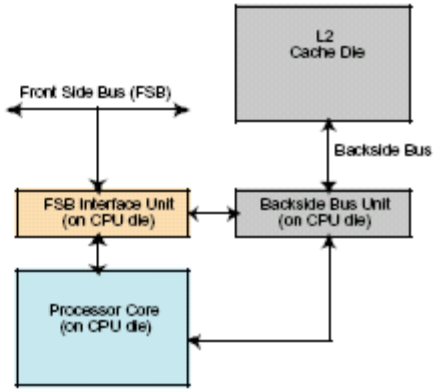




Move L2 Closer to Processor

- Pentium Pro
  - Processor and L2 Cache in "dual cavity"
    - Single package, separate die
  - 16K (8K Data, 8K Code) L1, 256K/512K L2





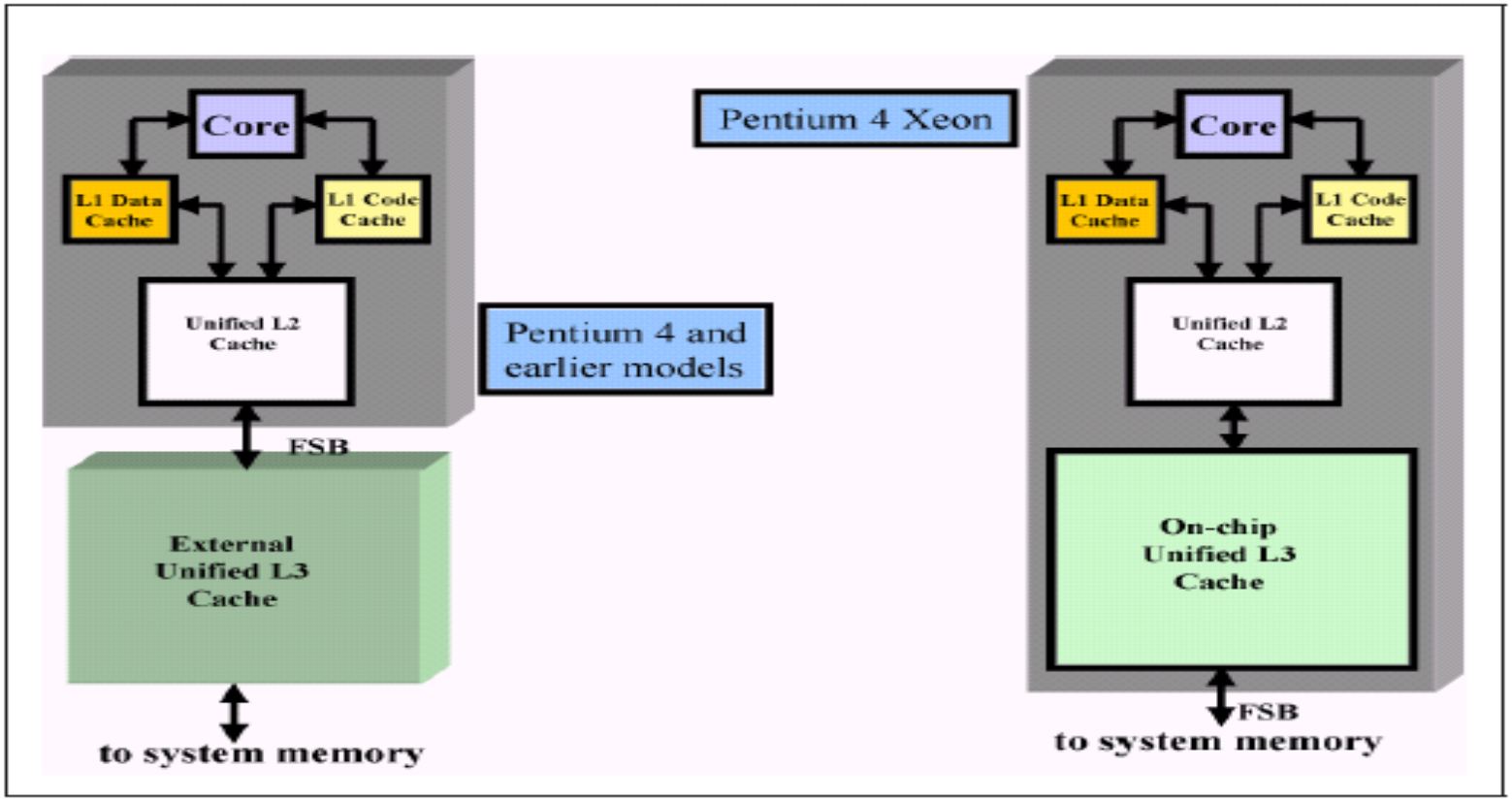
Celeron Mendocino (and subsequent processors)

- Put L2 Cache on processor die (BSB at core speed)

Trace Cache

- L1 Code Cache stored IA opcodes
  - Need to re-decode them each time
- Pentium 4
  - "Trace" cache replaced L1 Code Cache
  - Caches a "trace": sequence of instructions which may include a taken or untaken branch (i.e. cached instructions may not be contiguous in memory)
  - Stores 12K μops
  - Uses unified L2 cache (for opcodes) if miss

Add Another Cache (L3)

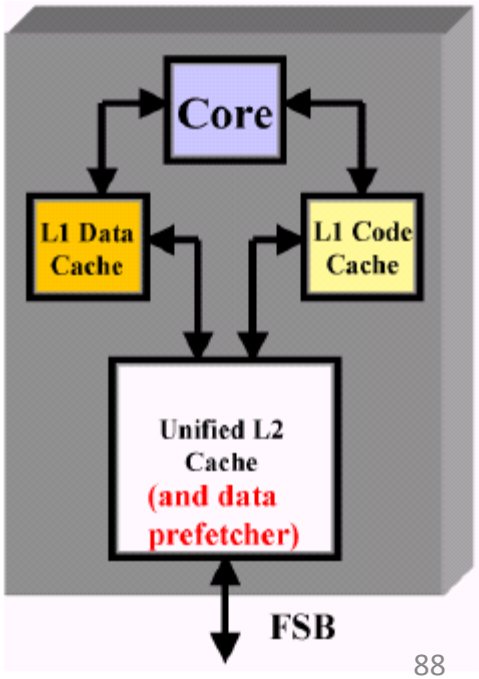


Cache Stalls

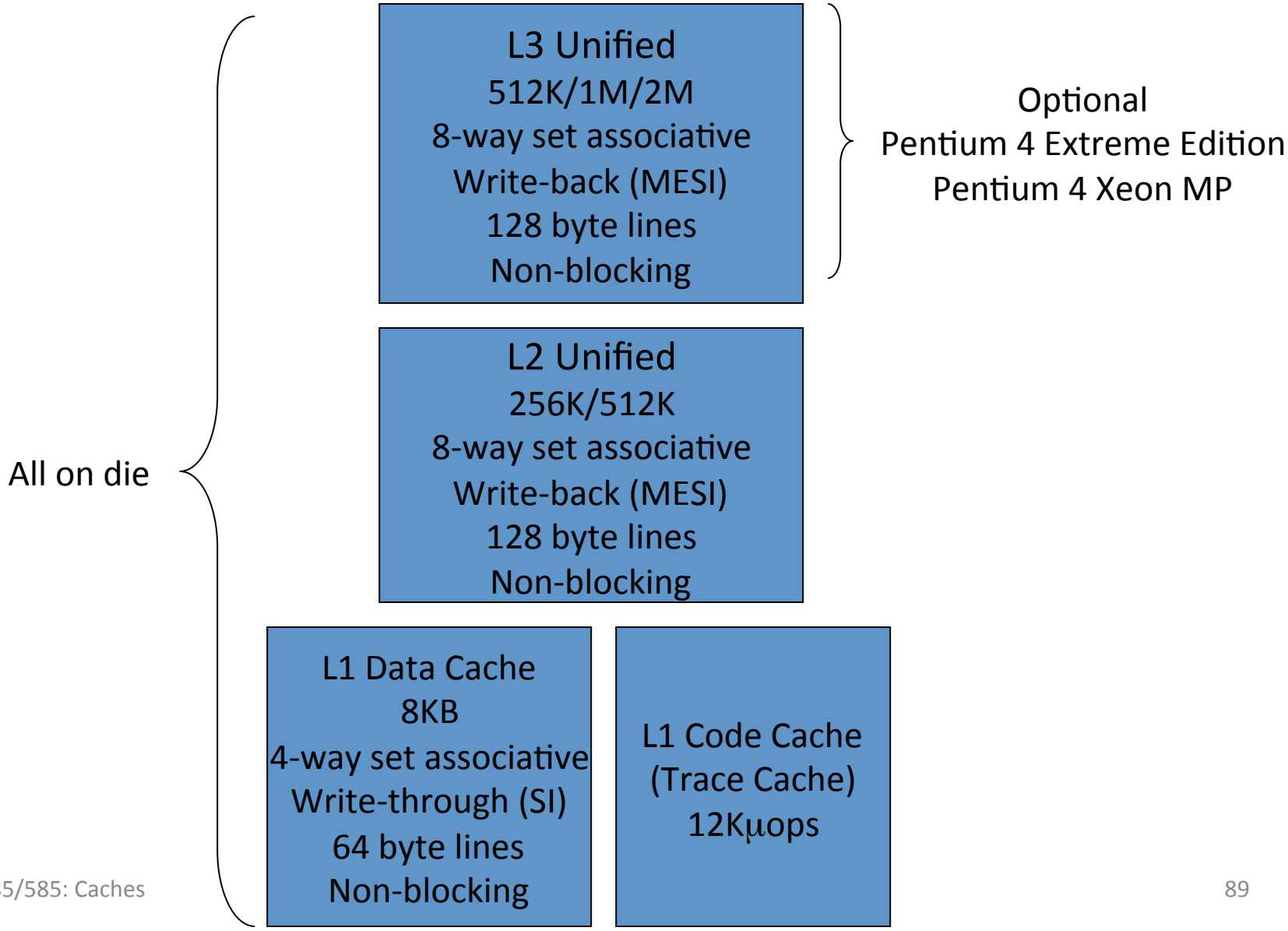
- Cache miss resulted in processor "stall"
  - No further memory requests could be submitted to cache until completion of required cache line fill
- Pentium Pro (and subsequent processors)
  - Non-blocking cache
    - Hit under miss
    - Miss under miss

Data Pre-fetching

- Prior to Pentium 4, IA32 processors pre-fetched instructions but not data
- Pentium 4 added data pre-fetching capability to L2 Cache
  - Use memory bandwidth if available



Pentium 4 Caches



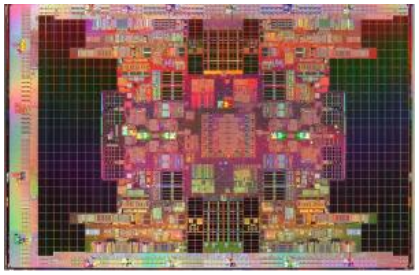
Caches: History

Processor	Type	Year of Introduction	L1 cache[a]	L2 cache	L3 cache
IBM 360/85	Mainframe	1968	16 to 32 KB	—	—
PDP-11/70	Minicomputer	1975	1 KB	—	—
VAX 11/780	Minicomputer	1978	16 KB	—	—
IBM 3033	Mainframe	1978	64 KB	—	—
IBM 3090	Mainframe	1985	128 to 256 KB	—	—
Intel 80486	PC	1989	8 KB	—	—
Pentium	PC	1993	8 KB/8 KB	256 to 512 KB	—
PowerPC 601	PC	1993	32 KB	—	—
PowerPC 620	PC	1996	32 KB/32 KB	—	—
PowerPC G4	PC/server	1999	32 KB/32 KB	256 KB to 1 MB	2 MB
IBM S/390 G4	Mainframe	1997	32 KB	256 KB	2 MB
IBM S/390 G6	Mainframe	1999	256 KB	8 MB	—
Pentium 4	PC/server	2000	8 KB/8 KB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 KB/32 KB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 KB	2 MB	—
Itanium	PC/server	2001	16 KB/16 KB	96 KB	4 MB
SGI Origin 2001	High-end server	2001	32 KB/32 KB	4 MB	—

[a] Two values seperated by a slash refer to instruction and data caches

[b] Both caches are instruction only; no data caches

Itanium Tukwila PC/Server 2008 30 MB (total) on die cache



Modern Systems

Characteristic	Intel Pentium P4	AMD Opteron
Virtual address	32 bits	48 bits
Physical address	36 bits	40 bits
Page size	4 KB, 2/4 MB	4 KB, 2/4 MB
TLB organization	1 TLB for instructions and 1 TLB for data Both are four-way set associative Both use pseudo-LRU replacement Both have 128 entries TLB misses handled in hardware	2 TLBs for instructions and 2 TLBs for data Both L1 TLBs fully associative, LRU replacement Both L2 TLBs are four-way set associativity, round-robin LRU Both L1 TLBs have 40 entries Both L2 TLBs have 512 entries TLB misses handled in hardware

Characteristic	Intel Pentium P4	AMD Opteron
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	8 KB for data, 96 KB trace cache for RISC instructions (12K RISC operations)	64 KB each for instructions/data
L1 cache associativity	4-way set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-through	Write-back
L2 cache organization	Unified (instruction and data)	Unified (instruction and data)
L2 cache size	512 KB	1024 KB (1 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	128 bytes	64 bytes
L2 write policy	Write-back	Write-back

Modern Systems

MPU	AMD Opteron	Intrinsity FastMATH	Intel Pentium 4	Intel PXA250	Sun UltraSPARC IV
Instruction set architecture	IA-32, AMD64	MIPS32	IA-32	ARM	SPARC v9
Intended application	server	embedded	desktop	low-power embedded	server
Die size (mm^2) (2004)	193	122	217		356
Instructions issued/clock	3	2	3 RISC ops	1	4×2
Clock rate (2004)	2.0 GHz	2.0 GHz	3.2 GHz	0.4 GHz	1.2 GHz
Instruction cache	64 KB, 2-way set associative	16 KB, direct mapped	12000 RISC op trace cache (~96 KB)	32 KB, 32-way set associative	32 KB, 4-way set associative
Latency (clocks)	3?	4	4	1	2
Data cache	64 KB, 2-way set associative	16 KB, 1-way set associative	8 KB, 4-way set associative	32 KB, 32-way set associative	64 KB, 4-way set associative
Latency (clocks)	3	3	2	1	2
TLB entries (I/D/L2 TLB)	40/40/512/ 512	16	128/128	32/32	128/512
Minimum page size	4 KB	4 KB	4 KB	1 KB	8 KB
On-chip L2 cache	1024 KB, 16-way set associative	1024 KB, 4-way set associative	512 KB, 8-way set associative	—	—
Off-chip L2 cache	—	—	—	—	16 MB, 2-way set associative
Block size (L1/L2, bytes)	64	64	64/128	32	32

IBM Pares Speed Gap In Memory Circuitry

Design for Data Storage May Be in Use Next Year; Performance Could Double

By Don Clark

International Business Machines Corp. is claiming a breakthrough in developing circuitry to store data on future microprocessor chips.

The big computer maker said its approach—based on exploiting the most widely used memory technology in a new way—could triple the data stored on a typical microprocessor with a resulting doubling of computing performance.

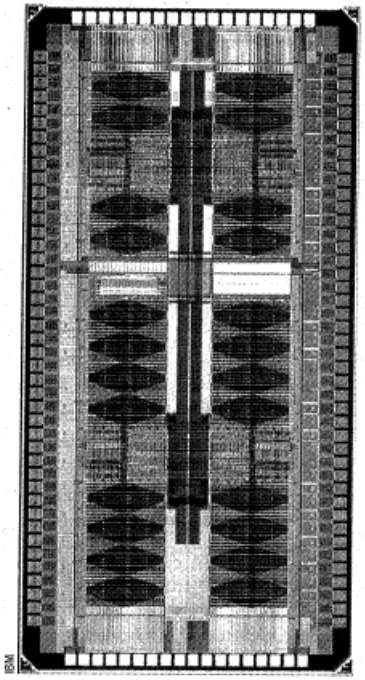
"We think this is the next big thing in getting more system performance," said Lisa Su, IBM's vice president of semiconductor research and development.

Microprocessors, the calculating engines for computers, increasingly come with storage circuitry to minimize the delays associated with fetching data from external memory chips. This "cache memory" typically uses a kind of circuitry used on chips called SRAMs, or static random-access memories.

SRAMs are fast but require six transistors to store a single bit of data. The more widely used chip known as dynamic random-access memories, or DRAMs, only need one transistor and another component, a capacitor, to store a bit. But DRAMs, though they can store more data in a smaller space, have generally been considered too slow for cache memory.

IBM researchers are discussing their progress in closing the speed gap at a conference in San Francisco today.

Exploiting a manufacturing technology called silicon-on-insulator, the company has developed unusually fast DRAM circuitry for use as cache memory. Subramanian Iyer, a director of IBM's manufacturing-process development, estimates it takes 1.5 nanoseconds—or billionths of a second—to fetch data from its enhanced DRAM technology, compared with 10 to 12 nanoseconds for conventional DRAMs and 0.8 to 1 nanoseconds for SRAMs. Mr. Iyer said three times more data can be stored in the same amount of space by switching from SRAM to DRAM circuitry; he expects the technology to be incorporated on microprocessors that will be manufactured next year using a new production process.



IBM plans to use new memory technology, shown in this diagram of a prototype chip.

Such benefits could help IBM's Power microprocessors in a performance race with chips from **Intel** Corp. and others. But Shekhar Borkar, the director of Intel's microprocessor technology lab, said extra manufacturing costs associated with using DRAM circuitry could outweigh the benefits.

IBM is a technology partner with **Advanced Micro Devices** Inc., an Intel rival that could benefit from the computer maker's memory research. Meanwhile, other alternatives to SRAM for cache memory are also being studied.

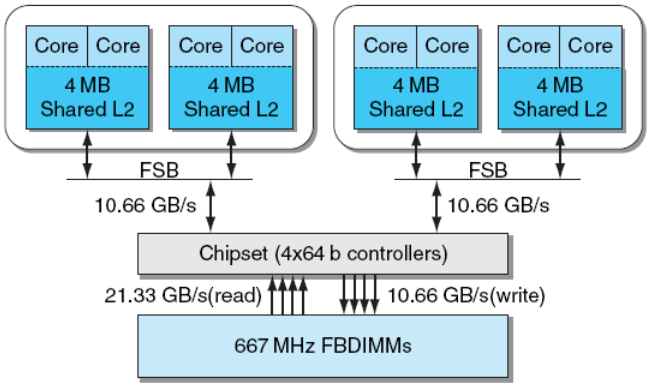
Innovative Silicon Inc., a start-up, has been promoting a technology it calls Z-RAM that stores data using a single transistor. Its licensees include AMD.

An AMD spokesman said it is "evaluating a number of new and emerging technologies" for cache memory.

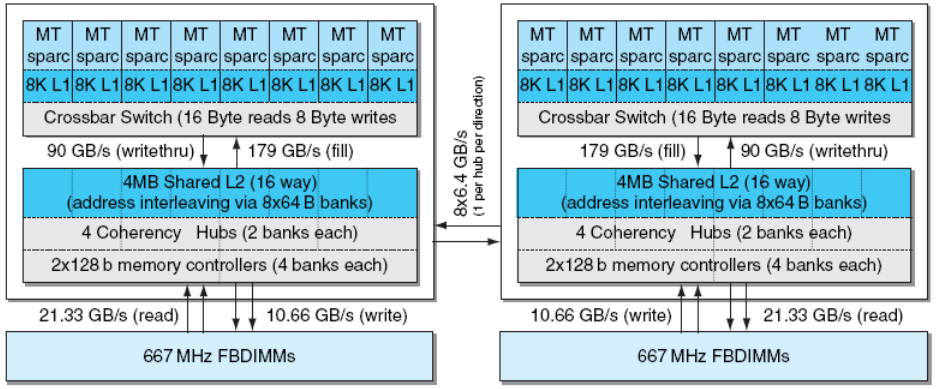
Use of Various Cache Coherence Schemes

Feature	AMD Opteron 8439	IBM Power 7	Intel Xenon 7560	Sun T2
Transistors	904 M	1200 M	2300 M	500 M
Power (nominal)	137 W	140 W	130 W	95 W
Max. cores/chip	6	8	8	8
Multithreading	No	SMT	SMT	Fine-grained
Threads/core	1	4	2	8
Instruction issue/clock	3 from onē thread	6 from one thread	4 from one thread	2 from 2 threads
Clock rate	2.8 GHz	4.1 GHz	2.7 GHz	1.6 GHz
Outermost cache	L3; 6 MB; shared	L3; 32 MB (using embedded DRAM); shared or private/core	L3; 24 MB; shared	L2; 4 MB; shared
Inclusion	No, although L2 is superset of L1	Yes, L3 superset	Yes, L3 superset	Yes
Multicore coherence protocol	MOESI	Extended MESI with behavioral and locality hints (13-state protocol)	MESIF	MOESI
Multicore coherence implementation	Snooping	Directory at L3	Directory at L3	Directory at L2
Extended coherence support	Up to 8 processor chips can be connected with HyperTransport in a ring, using directory or snooping. System is NUMA.	Up to 32 processor chips can be connected with the SMP links. Dynamic distributed directory structure. Memory access is symmetric outside of an 8-core chip.	Up to 8 processor cores can be implemented via Quickpath Interconnect. Support for directories with external logic.	Implemented via four coherence links per processor that can be used to snoop. Up to two chips directly connect, and up to four connect using external ASICs.

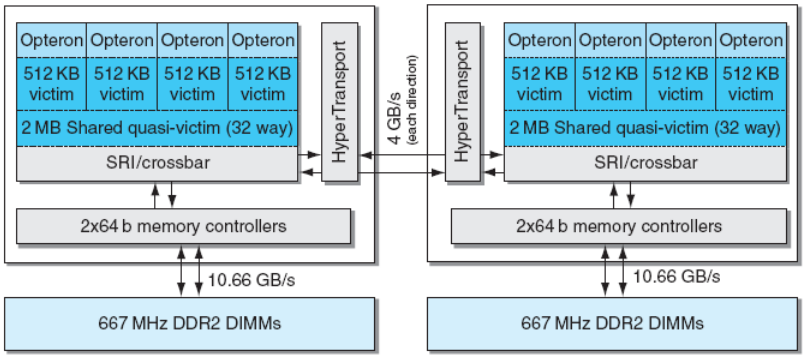
Caches in Multicore Processors



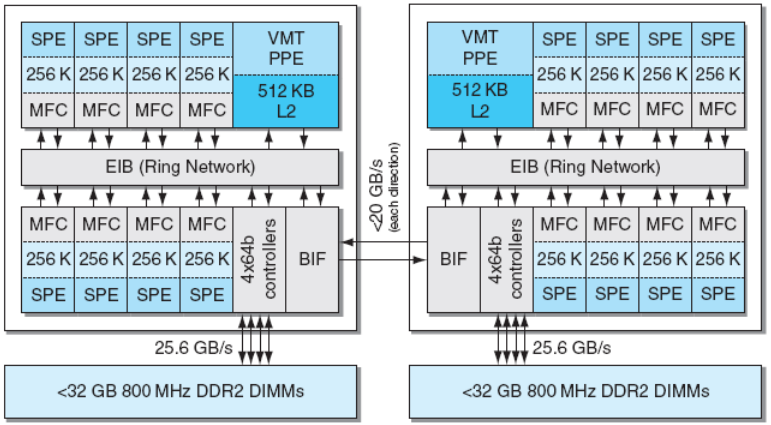
(a) Intel Xeon e5345 (Clovertown)



(c) Sun UltraSPARC T2 5140 (Niagara 2)



(b) AMD Opteron X4 2356 (Barcelona)



(d) IBM Cell QS20

ARM and Intel Core i7

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

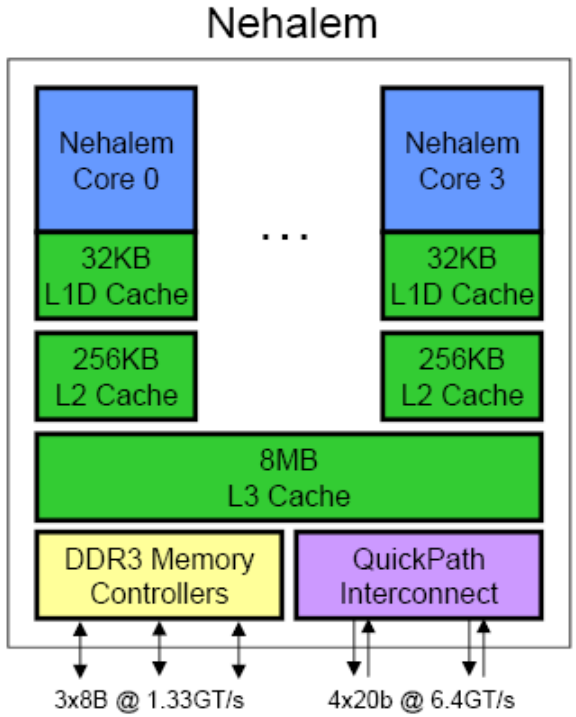
Intel Nehalem and AMD Opteron X4

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 cache associativity	4-way (I), 8-way (D) set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 cache associativity	16-way set associative	32-way set associative
L3 replacement	Not Available	Evict block shared by fewest cores
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

AMD Opteron uses exclusivity (L1, L2), not in L3
Intel Nehalem uses inclusivity

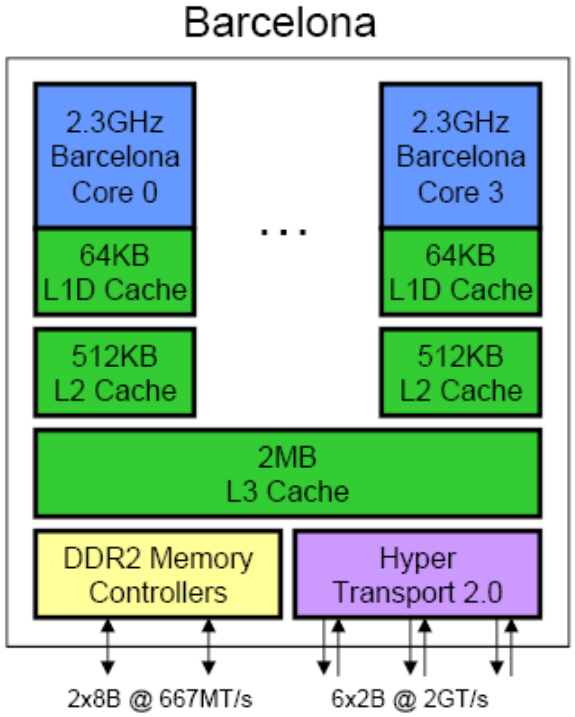
Intel Nehalem

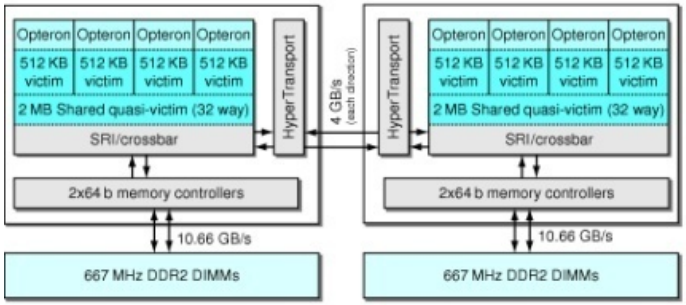
- Inclusivity in private L1, L2 caches
  - If line present in L1, will also be present in L2, L3
  - Facilitates cache coherence
- L3 cache shared across cores
  - Hint indicates which core's L2 holds line



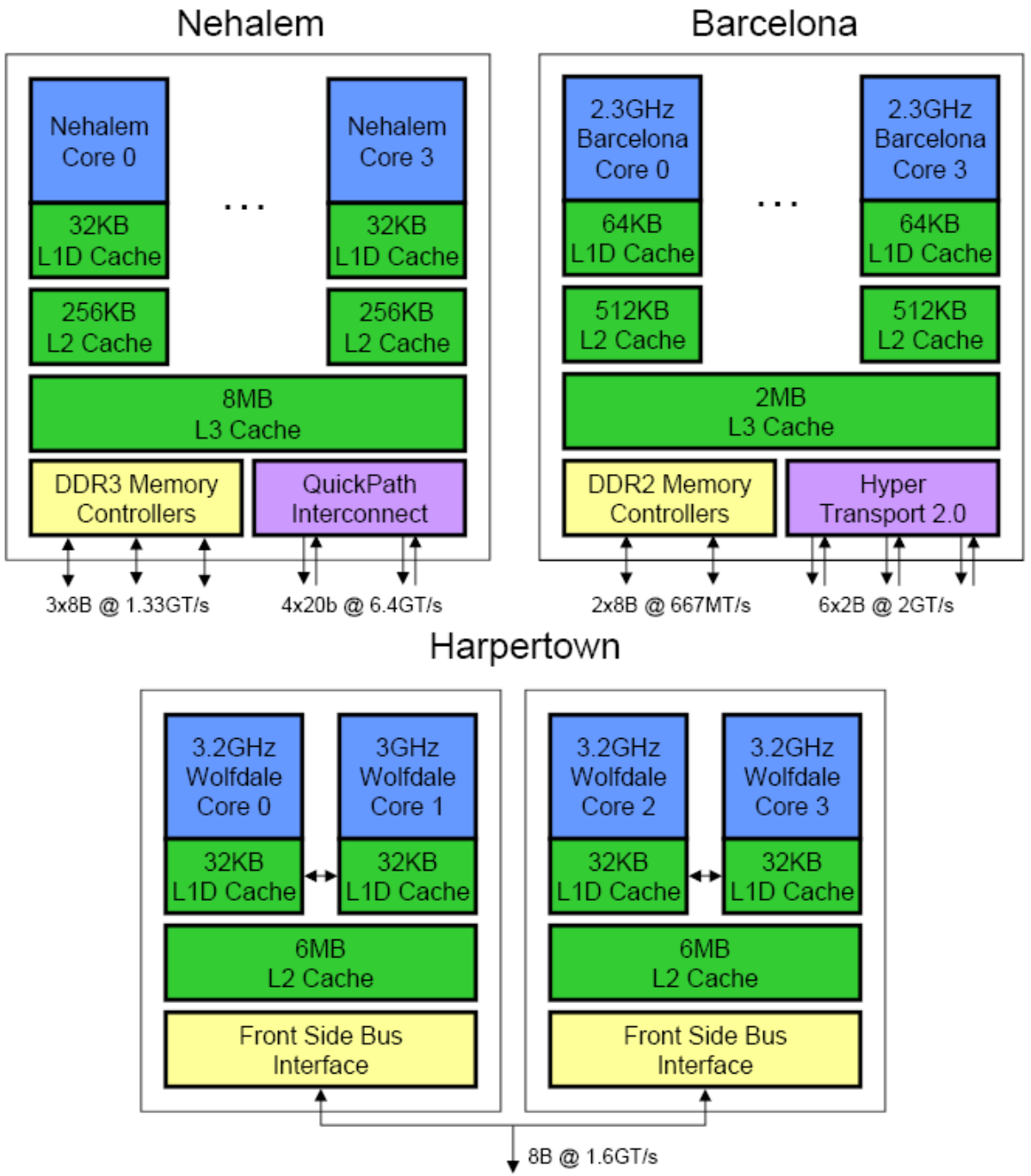
AMD Opteron x4 (Barcelona)

- Exclusivity in private L1, L2 caches
  - If line present in L1, will not be present in L2
  - L2 serves as large victim cache
- L3 cache shared across cores
  - Not exclusive
  - Line in L3 could be in one or more private L1 or L2 caches
- Miss in L1 can be satisfied by L2, L3, or SDRAM





- Associativity of shared caches
  - Want n-way associativity where n ≥ number of threads (to avoid conflicts/thrashing)


Nehalem
Nehalem Core 0
Nehalem Core 3
32KB L1D Cache
32KB L1D Cache
256KB L2 Cache
256KB L2 Cache
8MB L3 Cache
DDR3 Memory Controllers
QuickPath Interconnect
3x8B @ 1.33GT/s
4x20b @ 6.4GT/s
Barcelona
2.3GHz Barcelona Core 0
2.3GHz Barcelona Core 3
64KB L1D Cache
64KB L1D Cache
512KB L2 Cache
512KB L2 Cache
2MB L3 Cache
DDR2 Memory Controllers
Hyper Transport 2.0
2x8B @ 667MT/s
6x2B @ 2GT/s
Harpertown
3.2GHz Wolfdale Core 0
3GHz Wolfdale Core 1
3.2GHz Wolfdale Core 2
3.2GHz Wolfdale Core 3
32KB L1D Cache
32KB L1D Cache
32KB L1D Cache
32KB L1D Cache
6MB L2 Cache
6MB L2 Cache
Front Side Bus Interface
Front Side Bus Interface
8B @ 1.6GT/s

Some Caching Issues

- Caching Using Virtual or Physical Addresses?
- DMA Transfers
  - PCI bus adapter may access memory that's cached
  - Solutions
    - Processors snoop non-processor traffic
      - Northbridge (MCH) puts out FSB transactions with address only (0 byte) so that processors can snoop them
    - Mark some portions of memory as non-cacheable
- Memory Mapped I/O
  - Read from memory mapped I/O location
    - Would read other memory device addresses in "line"
    - Future accesses would result in cache hits
      - Read: Wouldn't actually access the I/O device!
        - » Device driver out of sync
        - » Bogus/stale data
      - Write: Wouldn't actually write to the I/O device!
        - » Lost data

MTRRs

- Memory Type and Range Registers
  - 486 and Pentium used 2 bits in Page Table Entries
    - PTE[PCD] and PTE[WT]
    - Cacheable Write Back, Cacheable Write Through, Uncacheable
    - Require Operating System to deal with it
  - Divides memory into regions
  - BIOS (motherboard-specific) sets up
  - H/W deals with it

MTRRs

- Uncacheable (UC)
  - Memory mapped I/O locations
- Write Combining (WC)
  - Video frame buffers
- Write Through (WT)
- Write Protect (WP)
  - Shadow RAM
    - Contains copy of ROM code (for performance)
    - Cached but as read only
- Write Back (WB)

Memory Hierarchy Design

- Range of interdependent cache design parameters
  - Number of caches
  - Cache size
  - Line size
  - Associativity
  - Split/Unified
  - Write Policy
  - Inclusion/Exclusion
  - Coherence
  - Block replacement algorithm
  - Error Detection
    - Soft errors not uncommon in SRAM-based caches
    - Use parity for tags, SECDED for data (why?)
    - Why not multiple bit correction?

How do you choose?

- Create a design space
- Run benchmarks on points in design space
- Observe behavior/performance
- Three principal techniques
  - Execution driven simulation
  - Trace driven simulation
  - Actual measurement

Execution Driven Simulation

- Simulation model of entire computer system
- Faithful, clock-accurate simulation
  - Not just instruction set architecture but microarchitecture
  - Capture effect of pipelining, prefetching, branch prediction, speculative execution
- Simulate benchmarks (programs) on model
  - Observe behavior
- Limitations
  - Impact of pipelining, branch prediction, speculative execution must be accounted for in processor simulation
  - Usually omits context switches (between benchmark and other processes and/or OS, interrupts)
  - Need to re-run entire simulation to gauge impact of change
  - Simulation speed orders of magnitude slower
    - Limits number and size of benchmarks

Trace Driven Simulation

- Capture benchmark's cache/memory references (a "trace")
- Using execution driven simulation or actual measurement
- Later run traces against a cache/memory model only
- Change parameters, re-run only cache/memory model
- Limitations
  - Impact of pipelining, branch prediction, speculative execution must be accounted for in processor simulation
  - Usually omits context switches (between benchmark and other processes and/or OS, interrupts)

Actual Measurement

- Add hardware to capture/record traces/statistics
- Ensures actual, real data
  - Including OS, other processes, I/O
- Fast: realtime
- Limitations
  - Hardware platform must exist
  - Access points: external bus, additional pins
  - Limited access to other state/parameters
  - Repeatability (starting from known state)
  - Difficult to build acquisition hardware
    - Speeds and volume of data preclude logic analyzers
    - On-chip support requires die array, design time (both precious)
      - Sometimes limited "counters" provided

Caching: A General Concept

- Disk caching
  - Part of the storage hierarchy
  - Disk blocks cached in OS and on HDD
  - Ratio of DRAM to disk access time >> than SRAM to DRAM!
- Web page caching