[A]rchitecture et [O]ptimisation de [C]ode pour microprocesseur hautes performances

Memory Hierarchy

Allen D. Malony

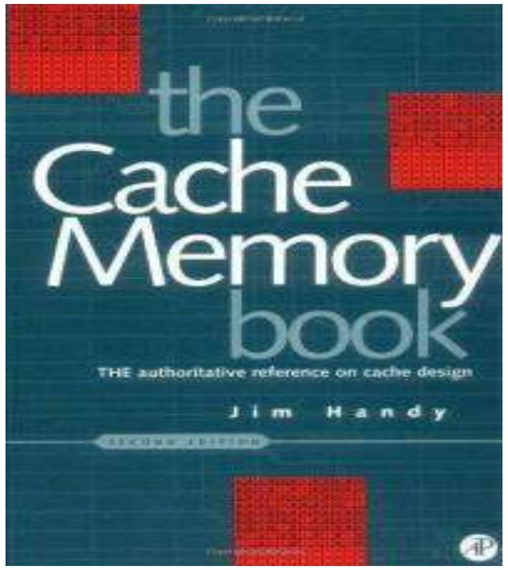
Department of Computer and Information Science
University of Oregon

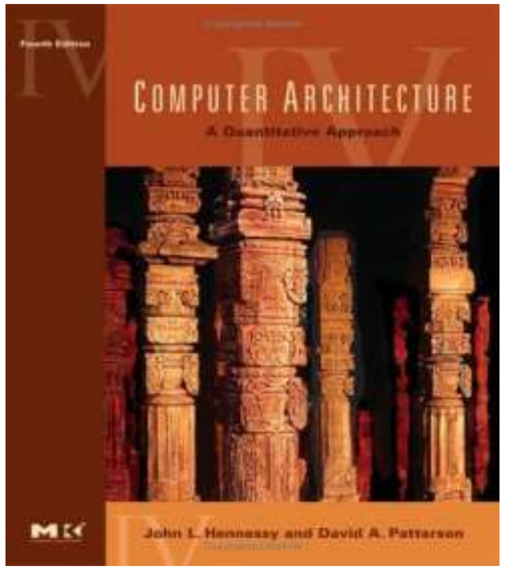
Memory Hiearchy

- □ General principles
- □ Direct mapped cache
- □ Associative cache
- □ Virtual and physical address
- □ Impact on code generation
- □ Advanced topics
- □ Sources

Origin of Course Materials

- J. Handy, The Cache Memory Book, 2nd Edition, ISBN: 9780123229809, Morgan Kaufmann, 1998.
- J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, 5th Edition, ISBN: 9780123838728, Morgan Kaufmann, 2011.
- Wikipedia



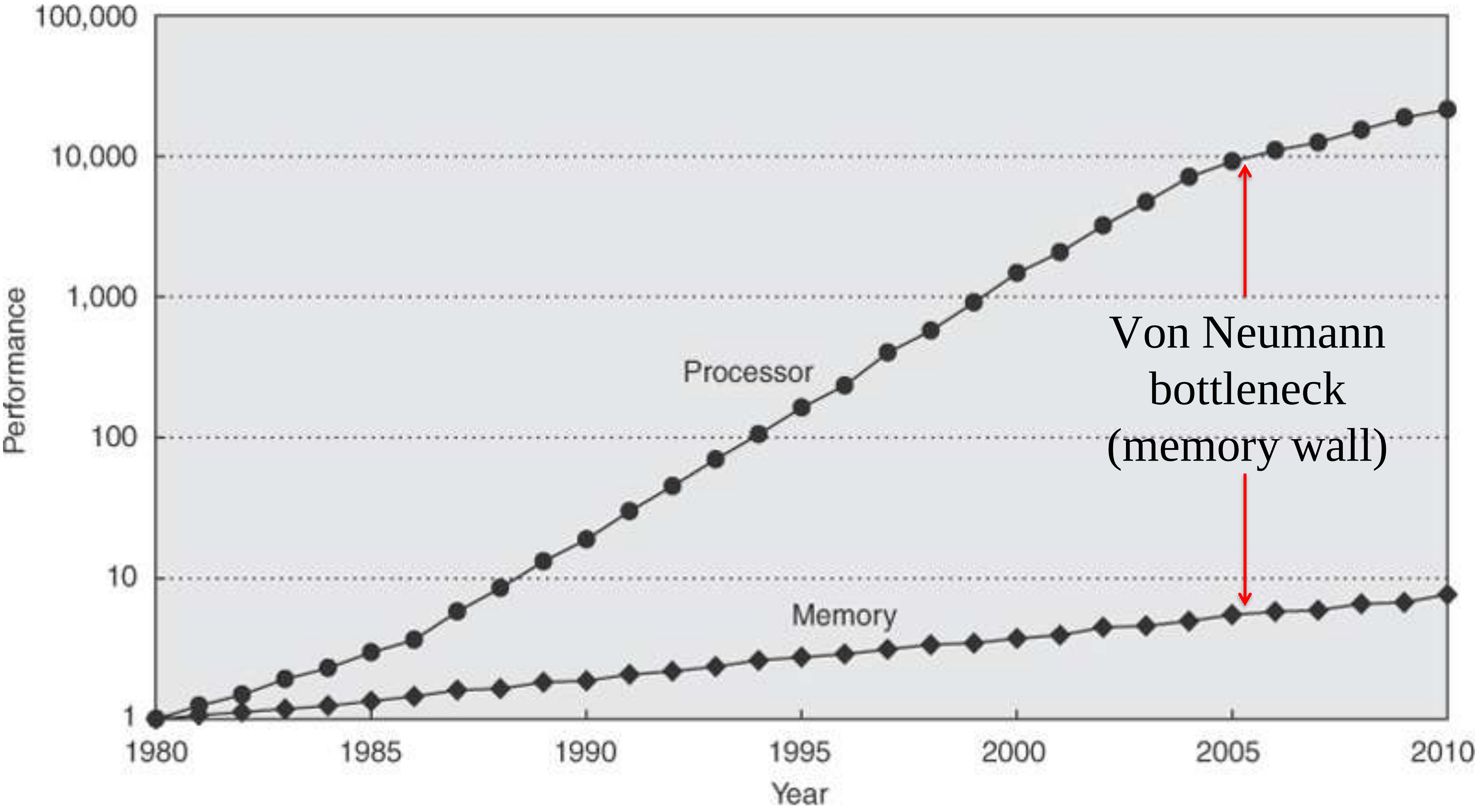


Memory Hierarchy Principles

- A *memory hierarchy* in a computer system makes it appear that the system has a *large* and *fast* memory
- Problem is that memory size is inversely proportional to memory speed for any specific memory technology
- Thus, a memory hierarchy implementation will use multiple levels of memory technology
  - Each level of memory will be faster, but slow
  - Together they provide the CPU with fast *effective access*
- We are concerned with the problem of how the processor uses the physical memory system
  - A memory hierarchy works because of locality of reference

Processor versus Memory Performance

- Processors can not access data fast enough



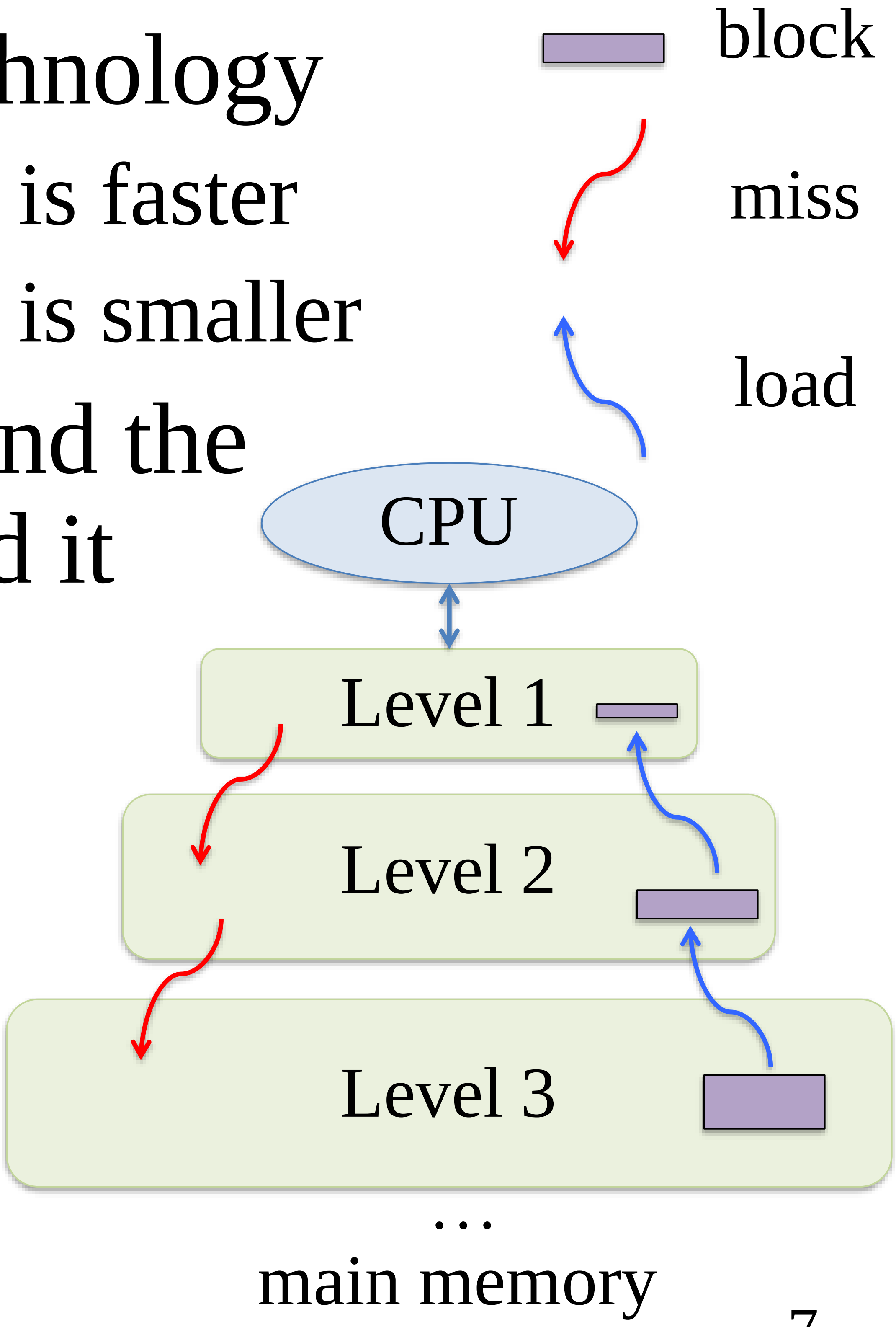
Von Neumann bottleneck (memory wall)

Memory Abstraction

- A processor references *physical memory* with a *physical* address of a particular size
  - 32 bits: 4 Gbytes maximum physical memory size
  - 48 bits: 256 Tbytes maximum physical memory size
  - 64-bit addressing used in many processor architectures
  - Typically an address is used to address a byte
- A processor reads and writes with *logical* addresses
  - Logical addresses are turned into physical addresses by *hardware address translation*
  - Physical address is then used in the memory system to read or write the physical memory location
- A read of a logical address should return the most recent value written to that logical address

Memory Hierarchy Architecture

- A hierarchical memory system is built from multiple levels of memory technology
  - Memory closer to the processor is faster
  - Memory closer to the processor is smaller
- A CPU will issue an address and the memory system will try to find it
- If an address is not in a level, a “miss” occurs and the next (slower) level is checked
- Blocks of data move between the levels

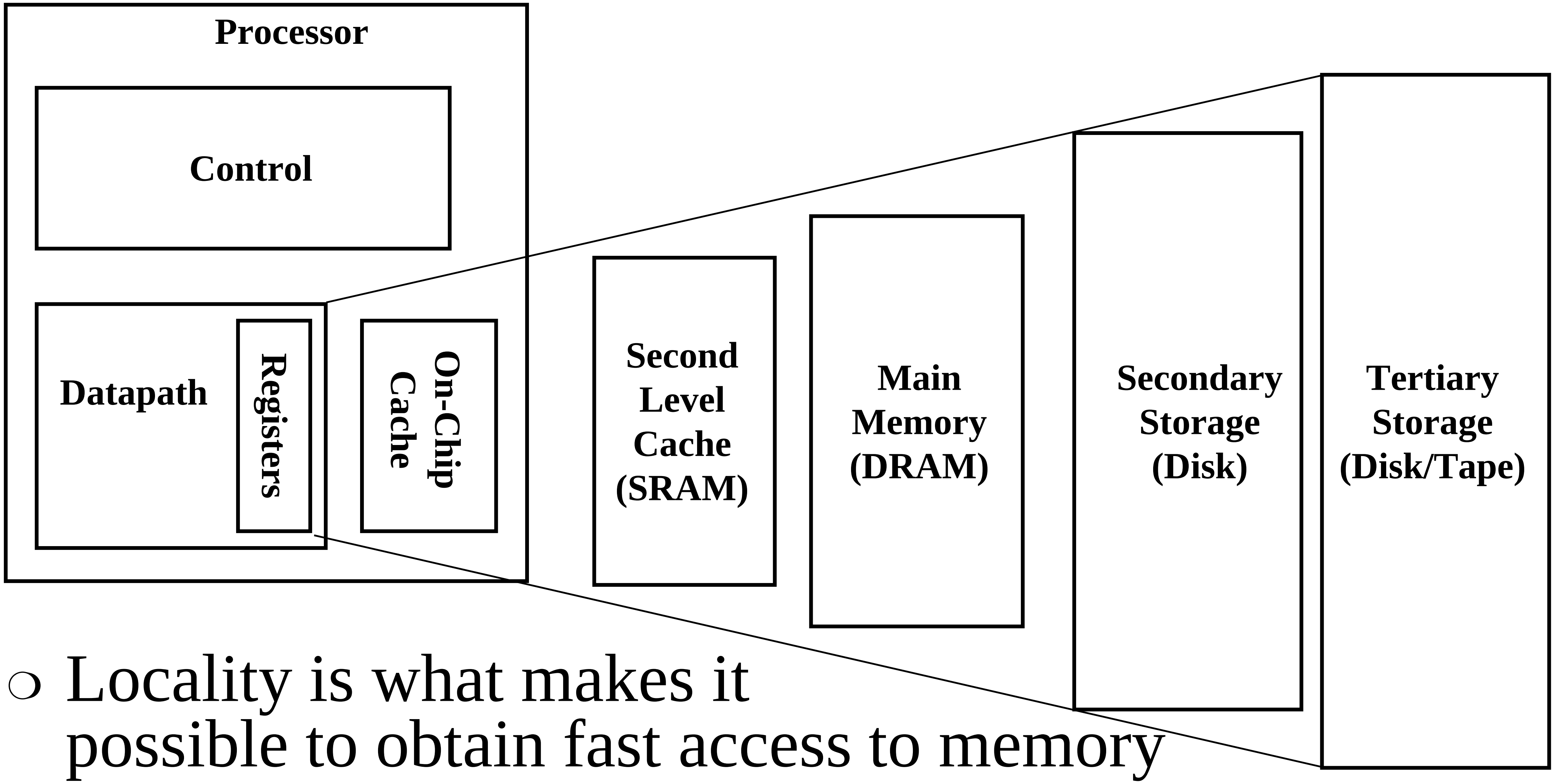


Principle of Locality of Reference

- Memory addresses produced by programs are not random, but show characteristics patterns of how instructions and data are accessed during execution
- Typically, programs reuse data and instructions that they have used recently
    - This is called *locality of reference*
- There are 2 types of locality of reference:
    - *Temporal locality*: recently accessed items are likely to be accessed in the near future
    - *Spatial locality*: items whose addresses are “near” each other will tend to be referenced close together in time

A Modern Memory Hierarchy

- Goal is to take advantage of locality
  - Provide as much memory as possible at the least cost
  - Provide memory access at the speed of the fast technology



  - Locality is what makes it possible to obtain fast access to memory

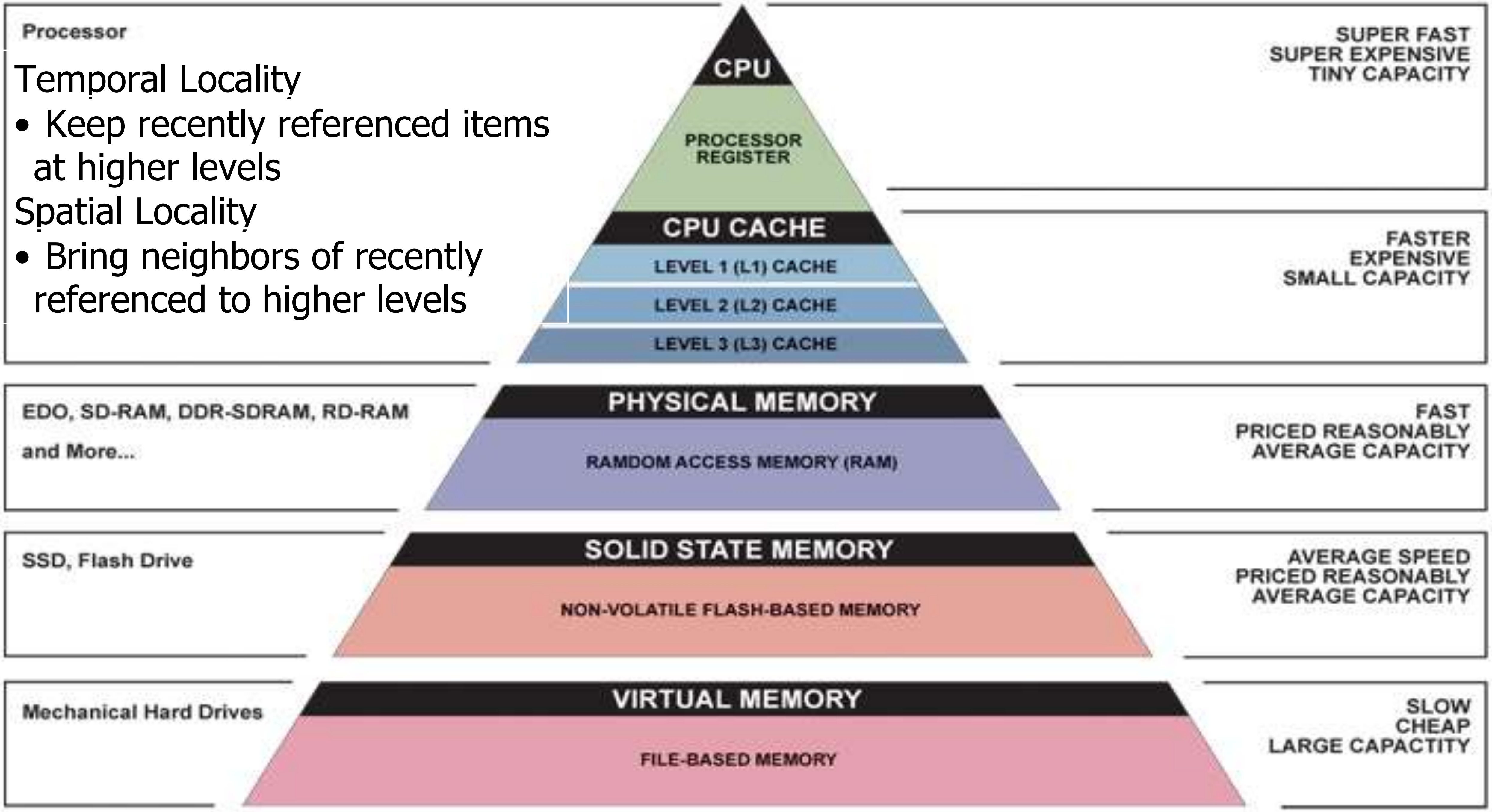
Memory Hierarchy Technology

Temporal Locality

- Keep recently referenced items at higher levels

Spatial Locality

- Bring neighbors of recently referenced to higher levels



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

Memory Hierarchy Organization

- Registers
  - Fastest memory element where data can reside
  - Accessed directly in instructions with register names
  - Managed by program (not the memory system)
- Caches
  - Next fastest memory
  - Can have a hierarchy with caches (L1, L2, L3)
  - Fastest cache (L1) is typically local to a processor
  - Slower caches might be shared between processors
- “Main” memory
  - Actual “true” location of data in physical memory
  - Registers and caches hold “copies” of data
    - any changes to data must be written back to memory

Logical vs. Physical Address Space

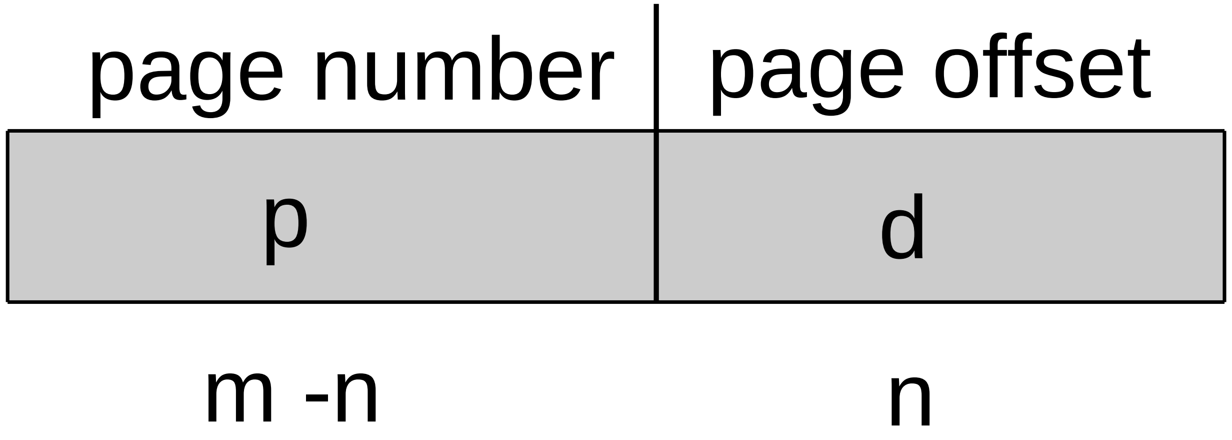
- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management
  - *Logical address* – generated by the CPU
  - *Physical address* – address seen by the memory unit
- *Logical address space* is the set of all logical addresses generated by a program
- *Physical address space* is the set of all physical addresses generated by a program

Paging

- Divide physical memory into fixed-sized blocks (*frames*)
  - Typically, frames have a size that is a power of 2, between 512 bytes and 16 Mbytes
  - Think of the physical address space of a process being allocated to physical memory frames
- Divide logical memory (of a process) into fixed-sized blocks (pages) of same size as frames
  - Think of the logical address space of a process being allocated to pages
- Need to map logical pages to physical frames
- OS (operating system) keeps track of all free frames
  - To run a program of size N pages, the OS needs to find N free frames in order to load program
- Set up a page table to translate logical to physical addresses

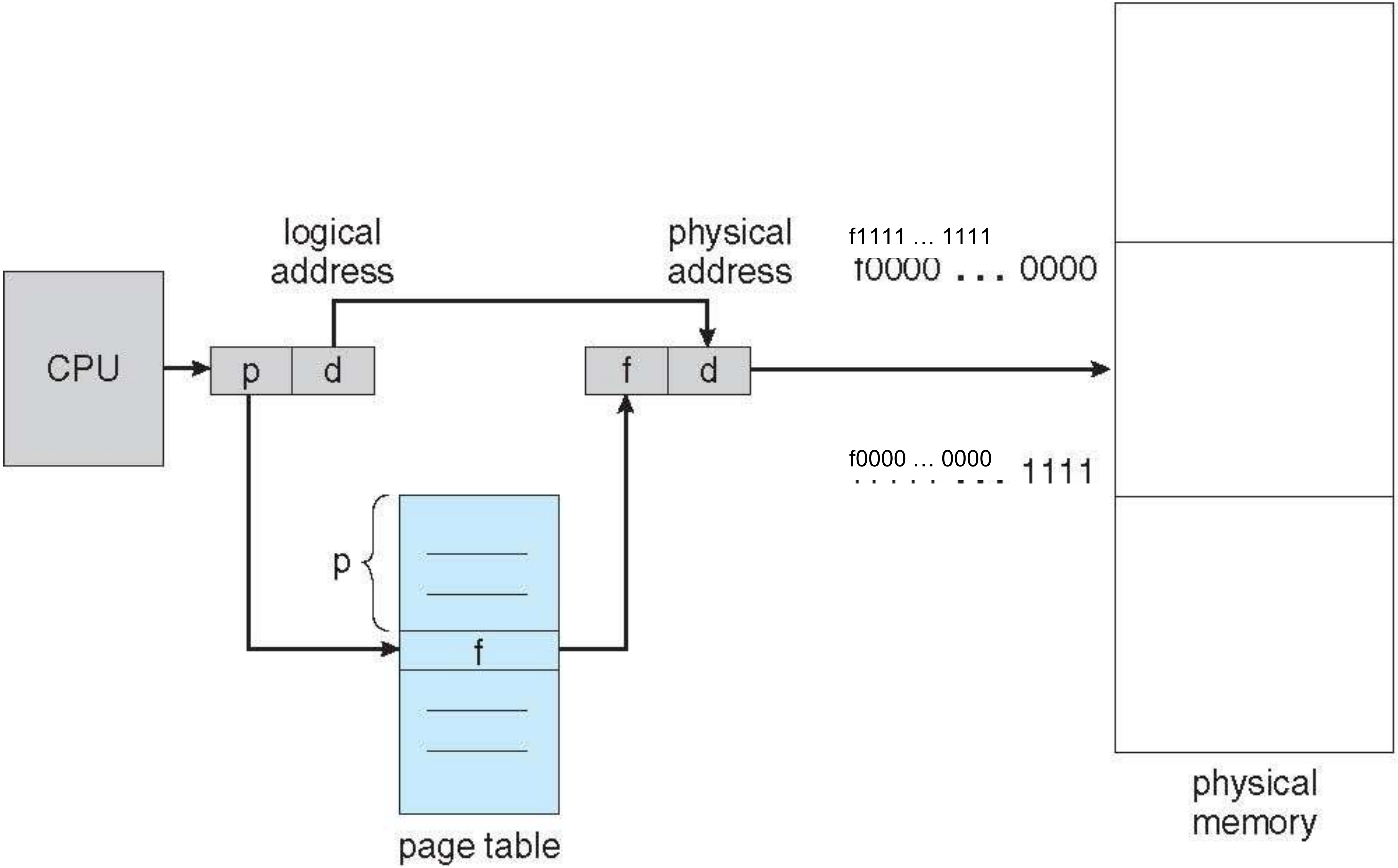
Address Translation Scheme

- Logical address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit

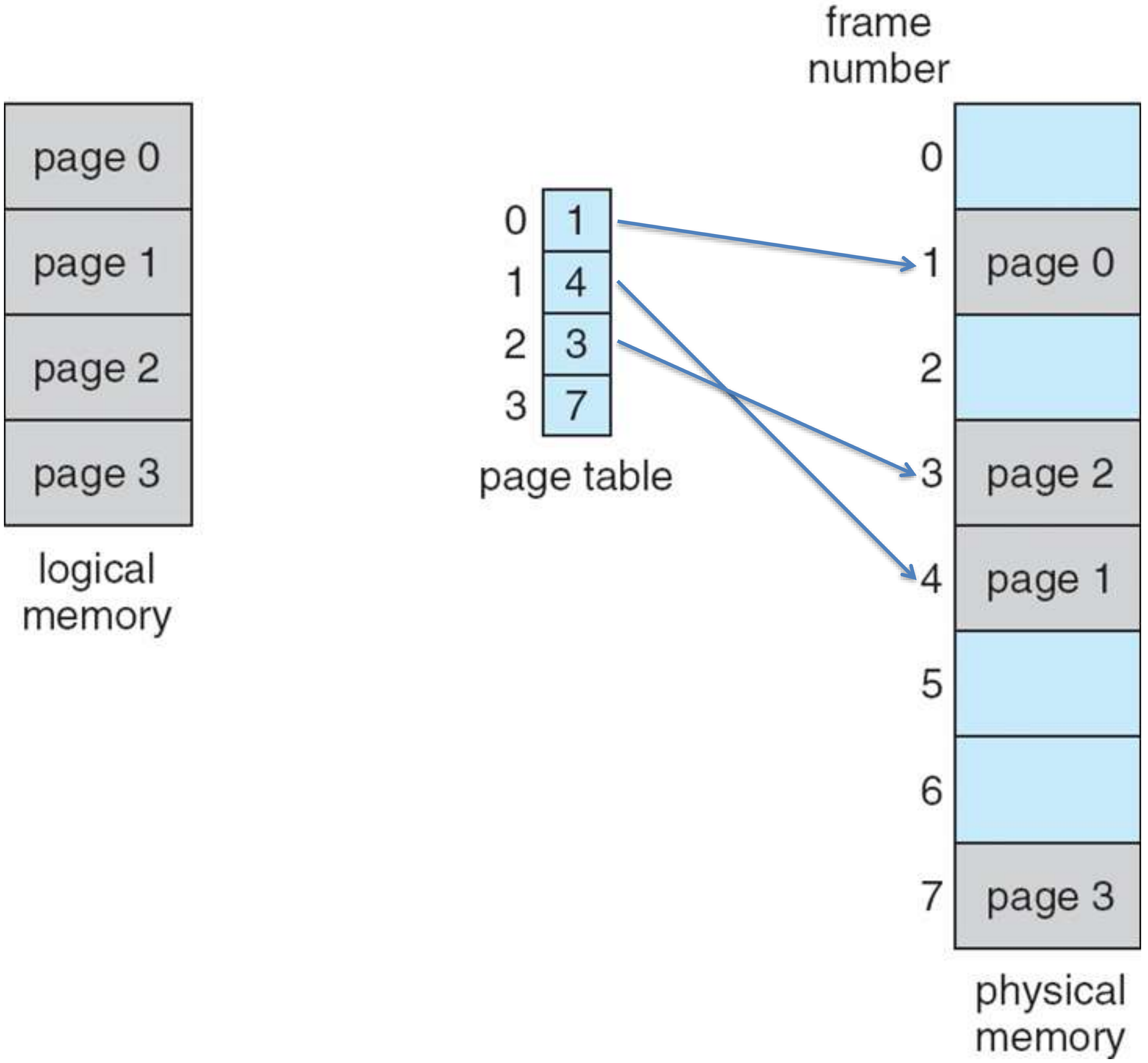


  - For given logical address space 2^m and page size 2^n

Paging Hardware



Paging Model of Logical / Physical Memory



Implementation of Page Table

- Page table is kept in main memory
- *Page-table base register (PTBR)* points to the page table
- *Page-table length register (PTLR)* indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- Instead, use a special fast-lookup hardware cache called *a translation look-aside buffers (TLBs)*
  - Fast associative memory that keeps recent frame #s

Page Table TLB

- Some TLBs store *address-space identifiers (ASIDs)* in each TLB entry – uniquely identifies each process to provide address-space protection for that process
    - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
    - Replacement policies must be considered
    - Some entries can be wired down for permanent fast access

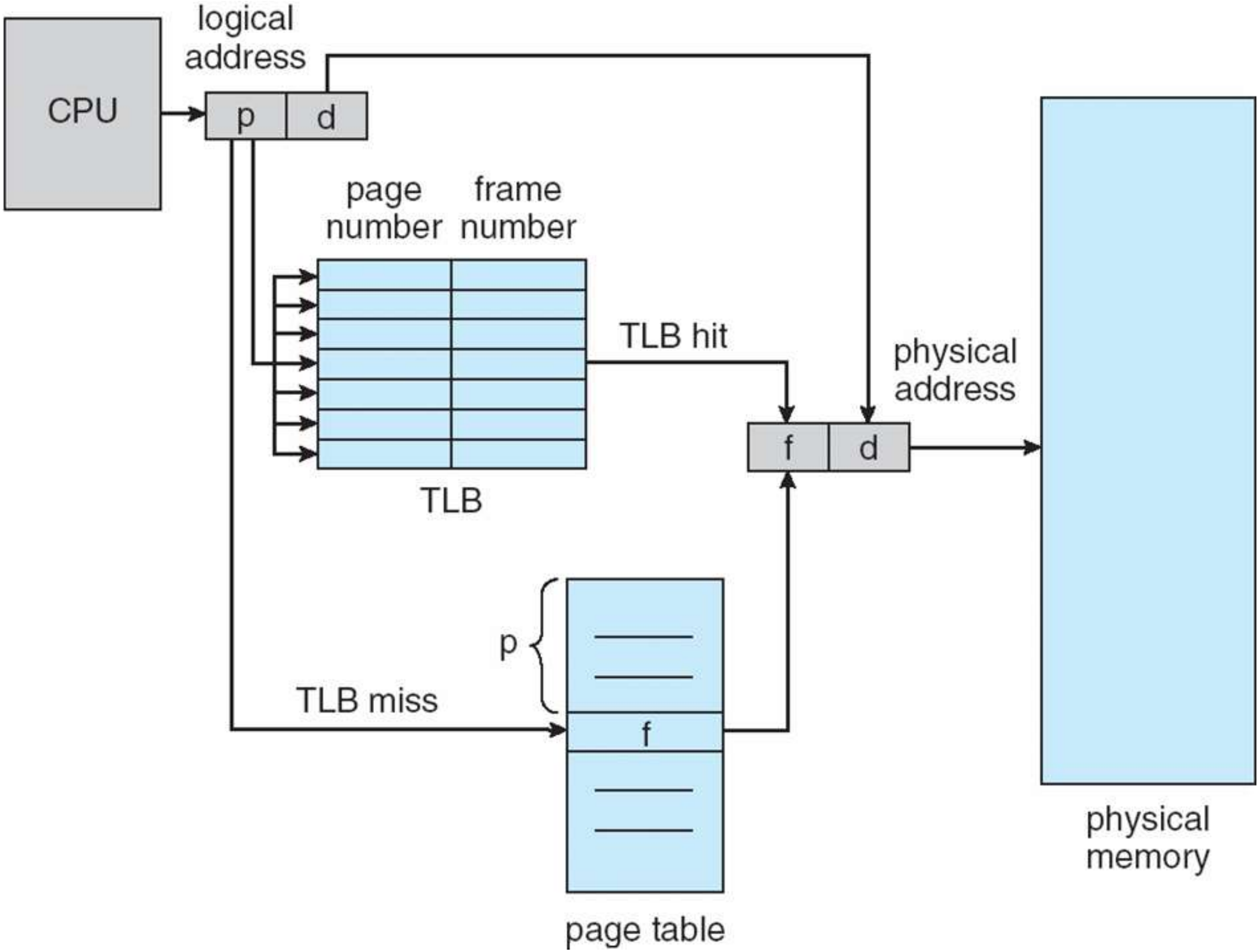
Associative Memory

- ❑ Associative memory is a memory that can be searched in parallel
  - ❍ All entries can be compared to a “key” simultaneously

Page #	Frame #

- ❑ Use for address translation (p, d)
  - ❍ If p is in associative register, get frame # out
  - ❍ Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

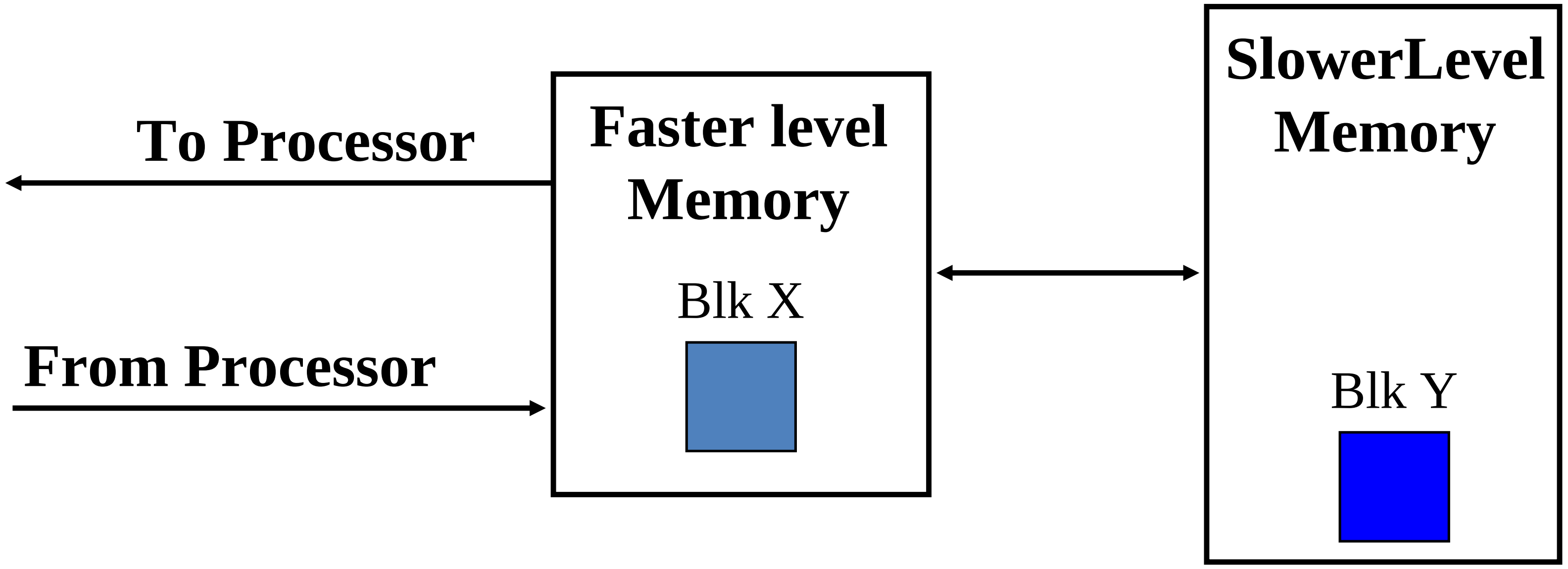
- Associative lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - *Hit ratio* – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider $\alpha = 80\%$, $\varepsilon = 20$ns for TLB search, 100ns for memory access
- *Effective Access Time (EAT)*

  $$\begin{aligned} \text{EAT} &= (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

- Consider $\alpha = 80\%$, $\varepsilon = 20$ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> $\alpha = 99\%$, $\varepsilon = 20$ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Terminology

- *Hit*: data appears in some block in the faster level memory
  - *Hit Rate*: fraction of memory access found in the faster level
  - *Hit Time*: RAM access time + time to determine hit/miss
- *Miss*: data is retrieved from a block in the slower level
  - *Miss Rate* = 1 - (Hit Rate)
  - *Miss Penalty*: time to replace a block + RAM access time
- Goal: Hit Time << Miss Penalty



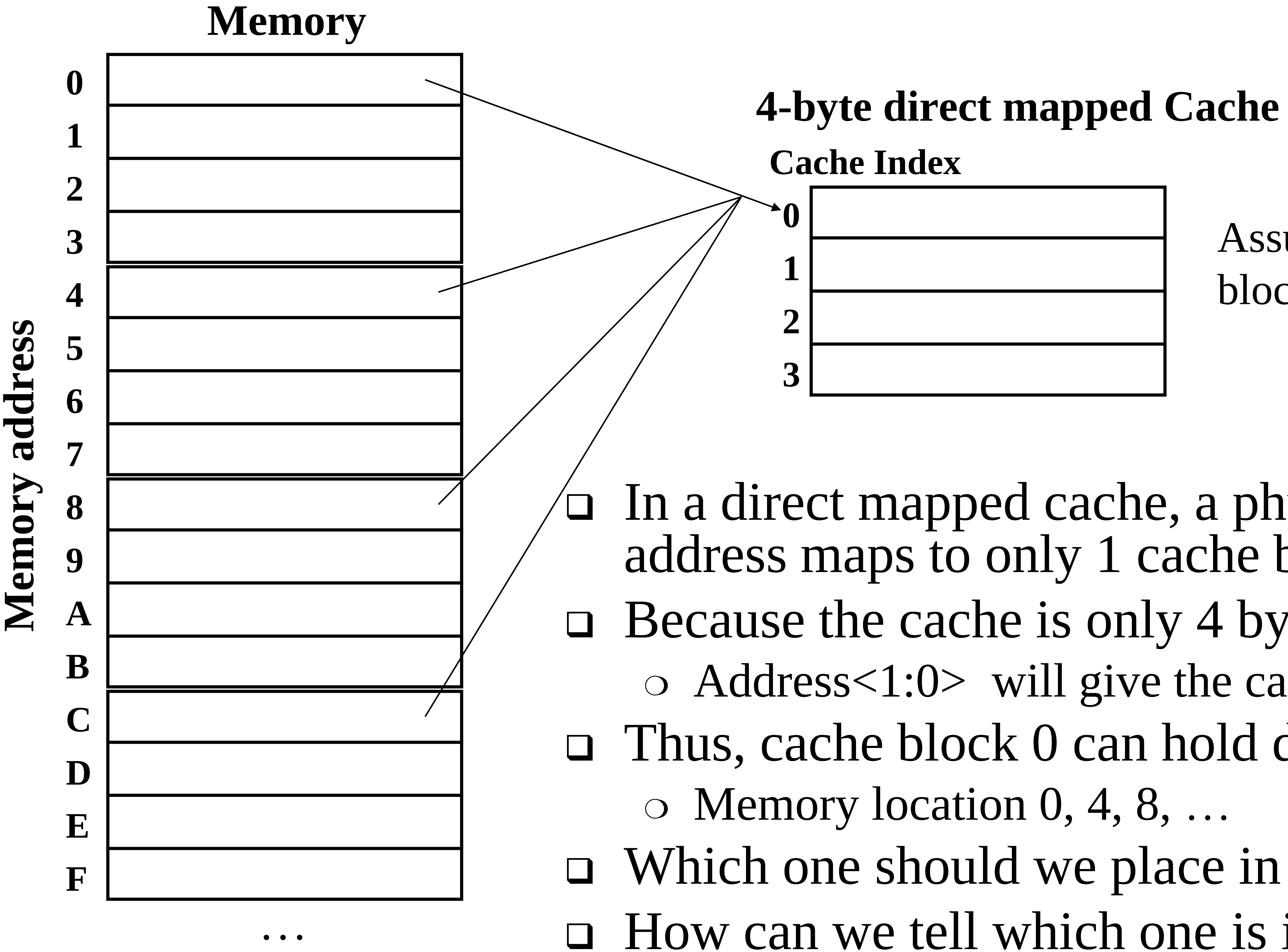
Questions for Memory Hierarchy

- Where can a block be placed in the faster level?
  - Know as *block placement*
- How is a block found if it is in the faster level?
  - Known as *block identification*
- Which block should be replaced on a miss?
  - Known as *block replacement*
- What happens on a write?
  - Known as *write strategy*

Cache Organization

- Caches are organized according to:
  - Block (line) size
  - Address mapping (i.e., how physical memory address map to the cache)
- Cache blocks contain multiple addressable “words”
  - Words have 1 to *n* bytes (typical 32-128 bytes)
  - Words in a cache block have consecutive addresses
- When a data address is read, the address will be mapped to a block in the cache
- Reading a data value will result in a whole cache block being replaced, if it is not already in cache

Consider a Direct Mapped Cache



Assume the cache block size is 1 byte

- In a direct mapped cache, a physical address maps to only 1 cache block
- Because the cache is only 4 bytes:
  - Address<1:0> will give the cache index
- Thus, cache block 0 can hold data from:
  - Memory location 0, 4, 8, …
- Which one should we place in the cache?
- How can we tell which one is in the cache?

Direct Mapped Cache

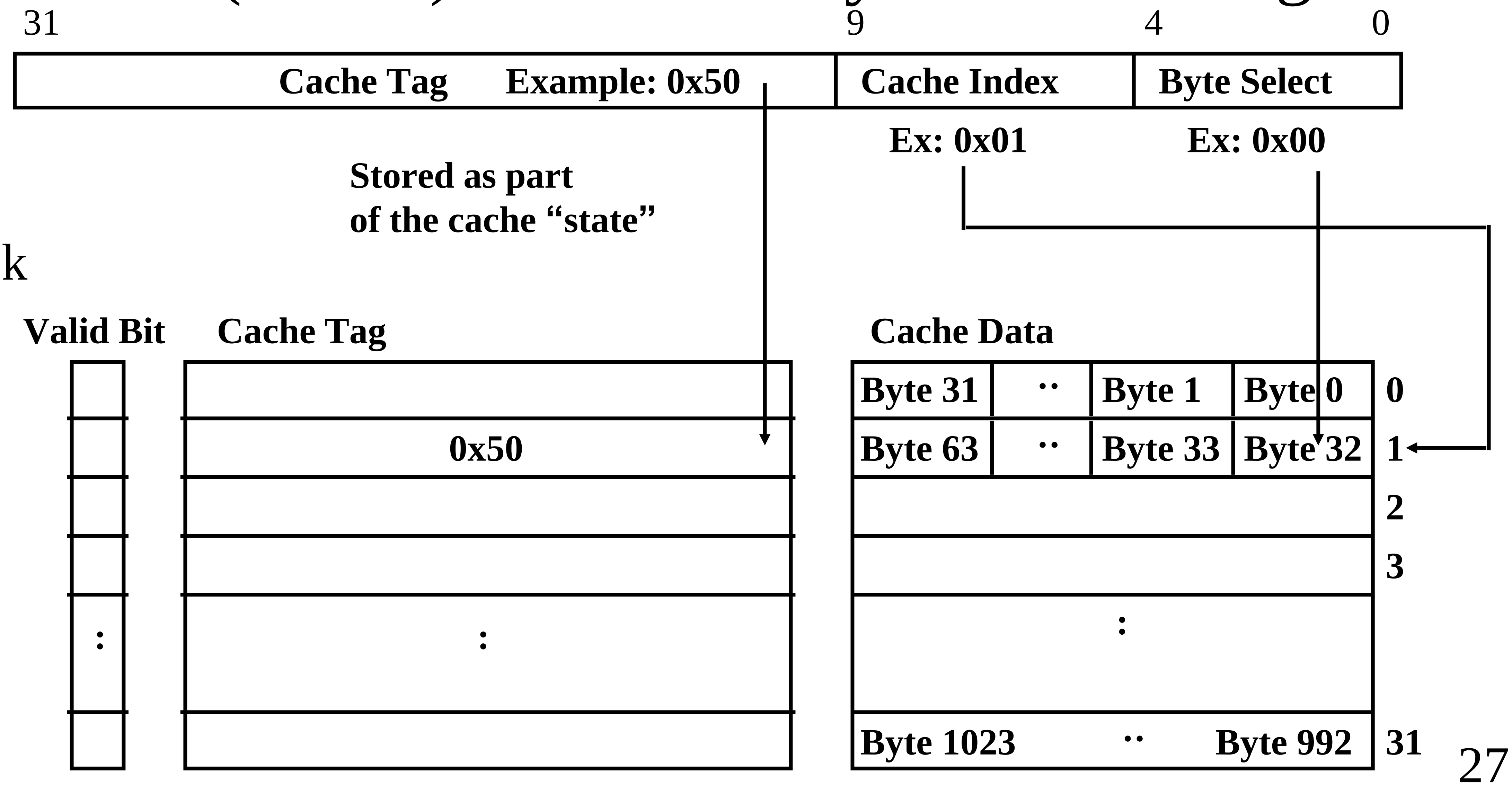
- Consider the following organization:
  - Cache size: 4 Mbytes (MB)
  - Cache line size 64 bytes (i.e., 64 bytes in a cache link)
  - # address bits: 64 bits (i.e., CPU uses 64-bit addresses)
  - Memory size: 1 Gbytes (GB)
- # cache lines = 4 MB / 64 B = 64 K
- # address bits to identify byte in cache line = 6
  - *Address<5…0>* used to access a byte in a cache line
- # address bits to identify cache line = 16
  - *Address<21…6>* identify which cache line is used
- 64 bits – 6 bits – 16 bits = 42 bits are used as the tag
  - *Address<63...22>* is stores as the tag in the cache

Direct Mapped Cache of Size 2^N bytes

- ❑ Suppose the address is 32 bits and block size = 2^M
  - ❍ Address<M-1…0> are used to select the byte in a block
  - ❍ Address <N-1…M> are used to select the cache line
  - ❍ Uppermost (32 - N) bits are always the cache tag

Cache parameters:

- • 1 KB size
- • 32 bytes/block
- • 32 blocks



31 9 4 0

Cache Tag Example: 0x50

Cache Index

Byte Select

Ex: 0x01

Ex: 0x00

Stored as part
of the cache "state"

Valid Bit

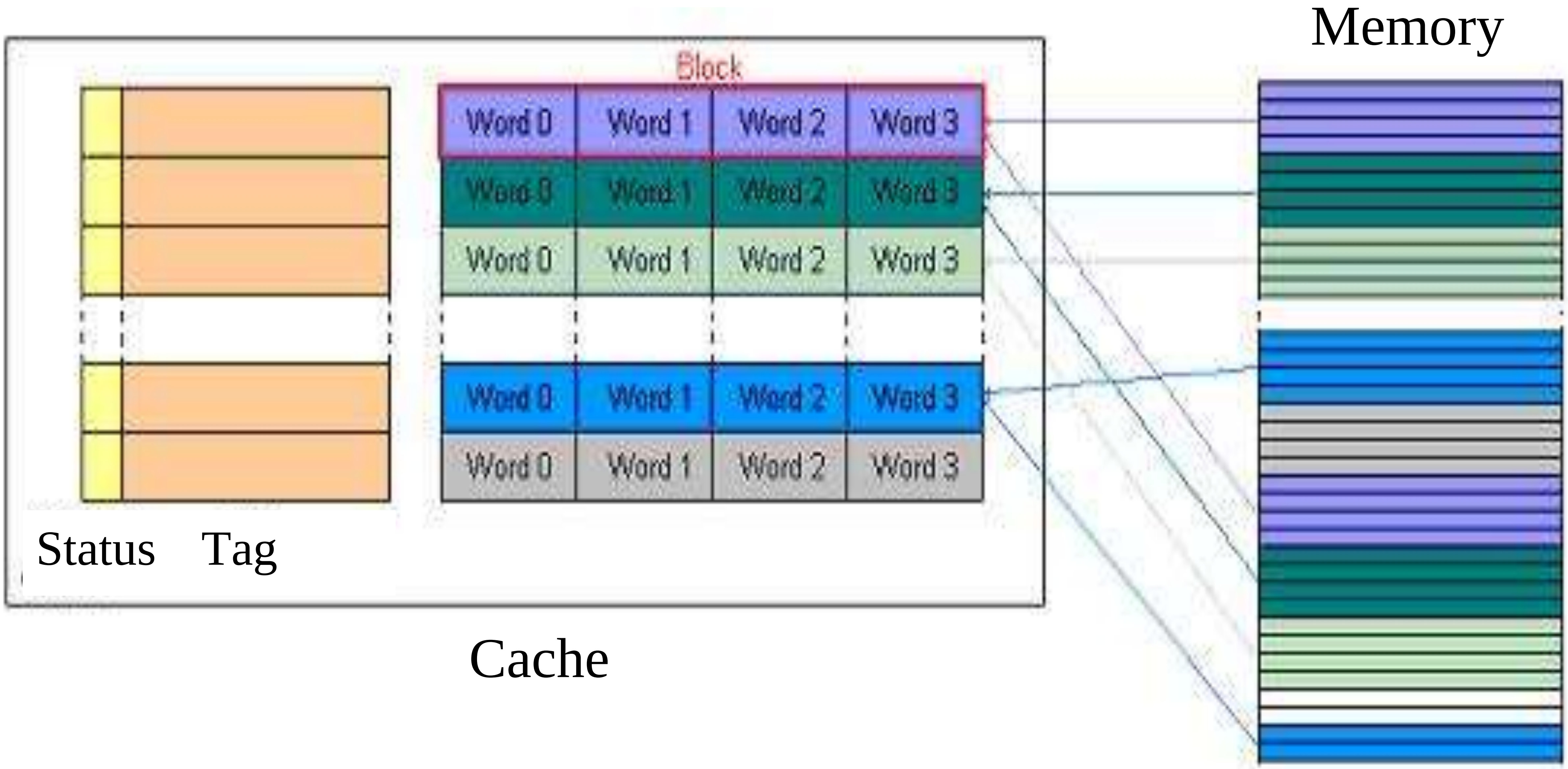
Cache Tag

Cache Data

Valid Bit	Cache Tag	Cache Data				
		Byte 31	··	Byte 1	Byte 0	0
	0x50	Byte 63	··	Byte 33	Byte 32	1
						2
						3
:	:	:				
		Byte 1023	··		Byte 992	31

Direct Mapped Cache (4 Words per Block)

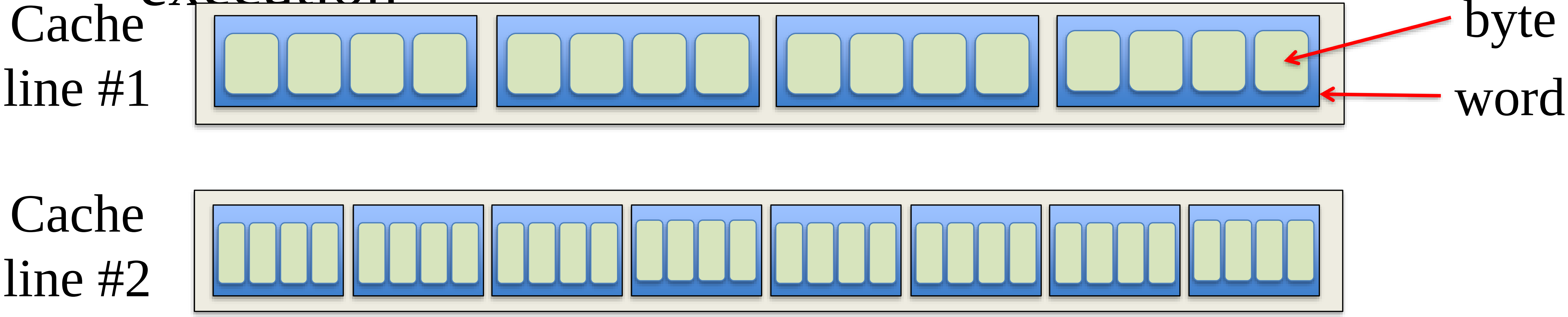
- Status bit indicates whether cache line is modified
- Tag contains address mapping information



Cache

Cache Lines

- ❑ Caches have multiple bytes (words) in a cache line
  - ❍ This supports spatial locality
  - ❍ A choice for a larger cache line says that you believe there is more spatial locality in your programs execution



Cache line #1

byte

word

Cache line #2

- ❑ Cache line #1 has 16 bytes (4x 4-byte words)
- ❑ Cache line #2 has 32 bytes (4x 4-byte words)

Cache Line Tradeoffs

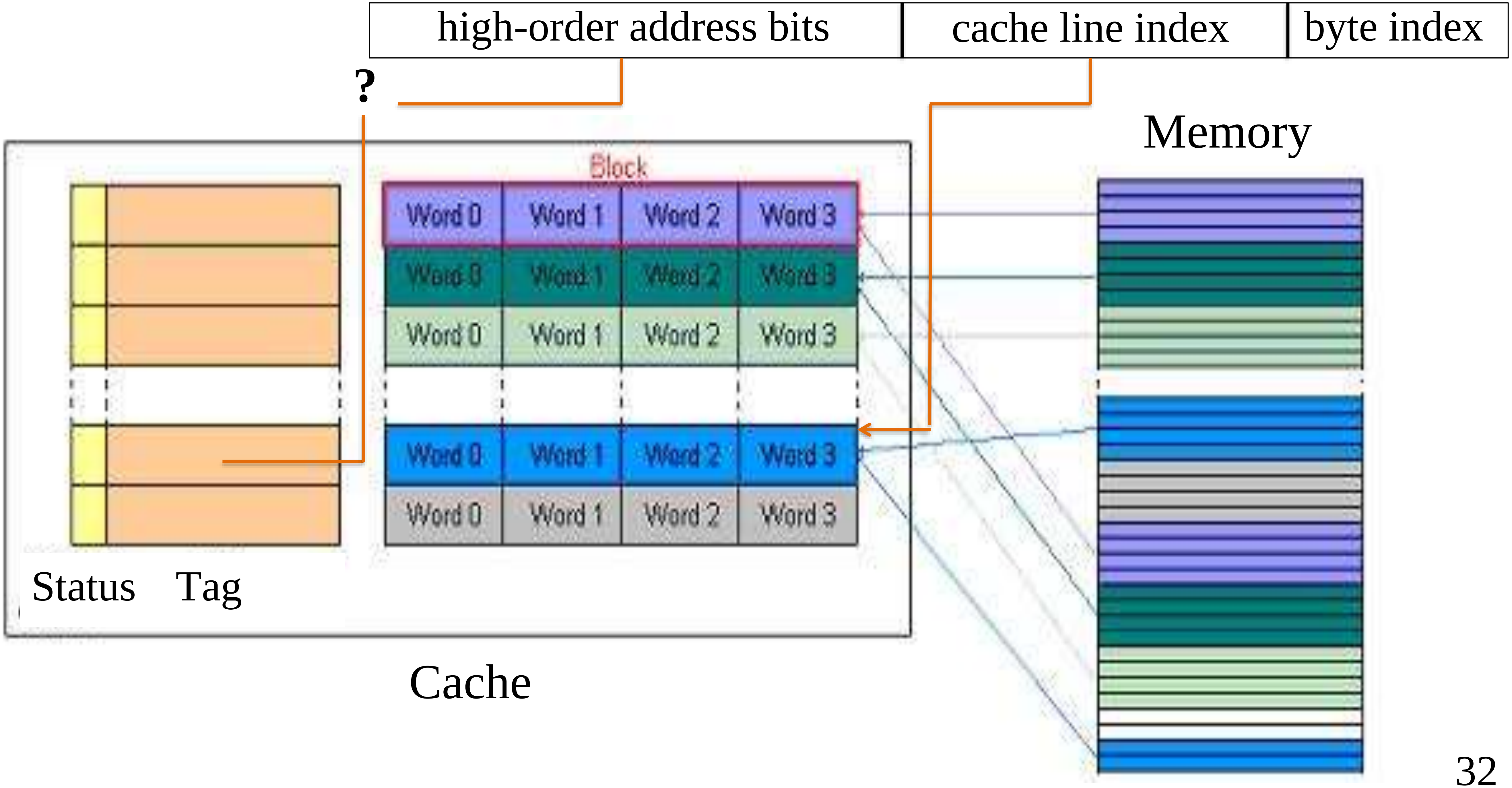
- When a memory reference is made (e.g., to a byte in memory), the memory system first checks to see if that byte is in the cache
  - If it is … great!
  - If it is not, the (current) cache line where the (new) byte will go must be replaced
  - If the (current) cache line has been modified, it must be written back to memory
  - Then, all bytes for the (new) cache line are read from memory
- Larger cache line (more bytes replaced)
  - + limits addressing hardware (addressing only lines)
  - + increases spatial locality opportunity (up to a point)
  - − increases memory traffic
  - − increases storage area (for same # lines

Taxonomy of Cache Misses

- A *cache miss* is when a memory reference is not found in the cache
- There are 3 kinds of cache misses
- *Cold miss*
  - Occurs when the memory location is first referenced
  - It is possible that a memory location that is close to this one was referenced and is in the cache line
- *Capacity miss*
  - Basically, it is when the “window” of spatial locality is greater than the cache size
- *Conflict miss*
  - This occurs when a memory location maps to a cache line that contains data from another memory region
  - This is easy to determine in a direct-mapped cache

Is a Memory Reference in the Cache?

- Need to compare the tag of the cache line with the high-order address bits in the memory reference



Direct Mapped Cache Problems (1)

- Consider the following code:

```
do I = 1, 1000
  s = s + A(i) * B(i)
enddo
```

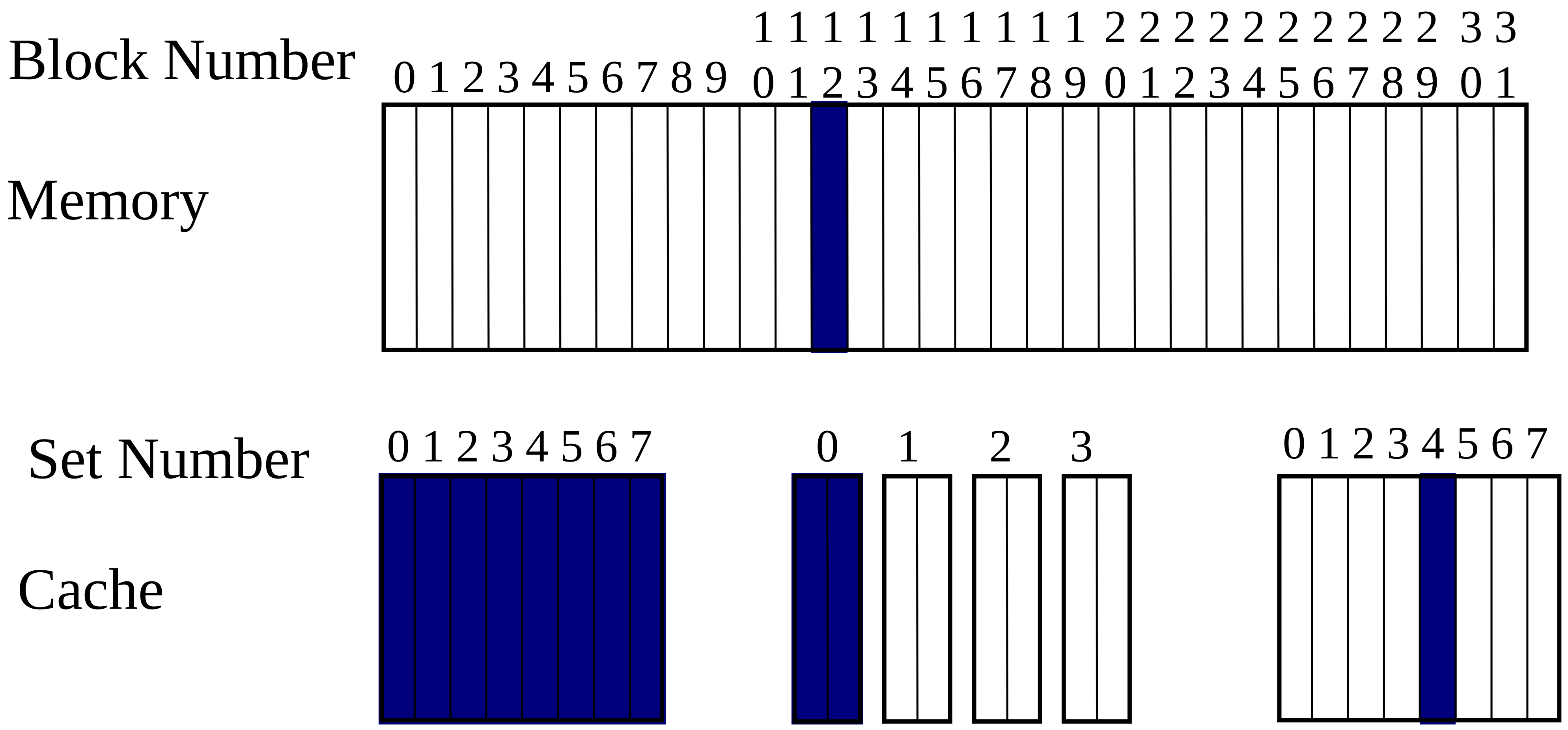
- Suppose we have the following cache parameters
  - Cache size: 2 KB
  - Line size: 32 B (can hold 4 double precision words)
- Suppose A() and B() have the following characteristics:
  - Arrays A() and B() are double precision (64 bits (8 bytes))
  - A(1) address is 2048 and B(1) address is 4096
  - Then A(1) and B(1) map to the same cache line
  - In fact, A(i) and B(i) map to the same cache line!

Direct Mapped Cache Problems (2)

Load fp1, A(1)	Miss	Bring A(1), A(2), A(3), A(4) in cache line 0	
Load fp2, B(1)	Miss	Bring B(1), B(2), B(3), B(4) in cache line 0	Evicts A(1), A(2), A(3), A(4)
Load fp1, A(2)	Miss	Bring back A(1), A(2), A(3), A(4) in cache line 0	Evicts B(1), B(2), B(3), B(4)
Load fp2, B(2)	Miss	Bring back B(1), B(2), B(3), B(4) in cache line 0	Evicts A(1), A(2), A(3), A(4)
Load fp1, A(3)	Miss	Bring back A(1), A(2), A(3), A(4) in cache line 0	Evicts B(1), B(2), B(3), B(4)
Load fp2, B(3)	Miss	Bring back B(1), B(2), B(3), B(4) in cache line 0	Evicts A(1), A(2), A(3), A(4)
Load fp1, A(4)	Miss	Bring back A(1), A(2), A(3), A(4) in cache line 0	Evicts B(1), B(2), B(3), B(4)

Set Associative Caches

- Organize cache blocks as sets for mapping



Fully Associative
anywhere

(2-way) Set Associative
anywhere in set 0
(12 mod 4)

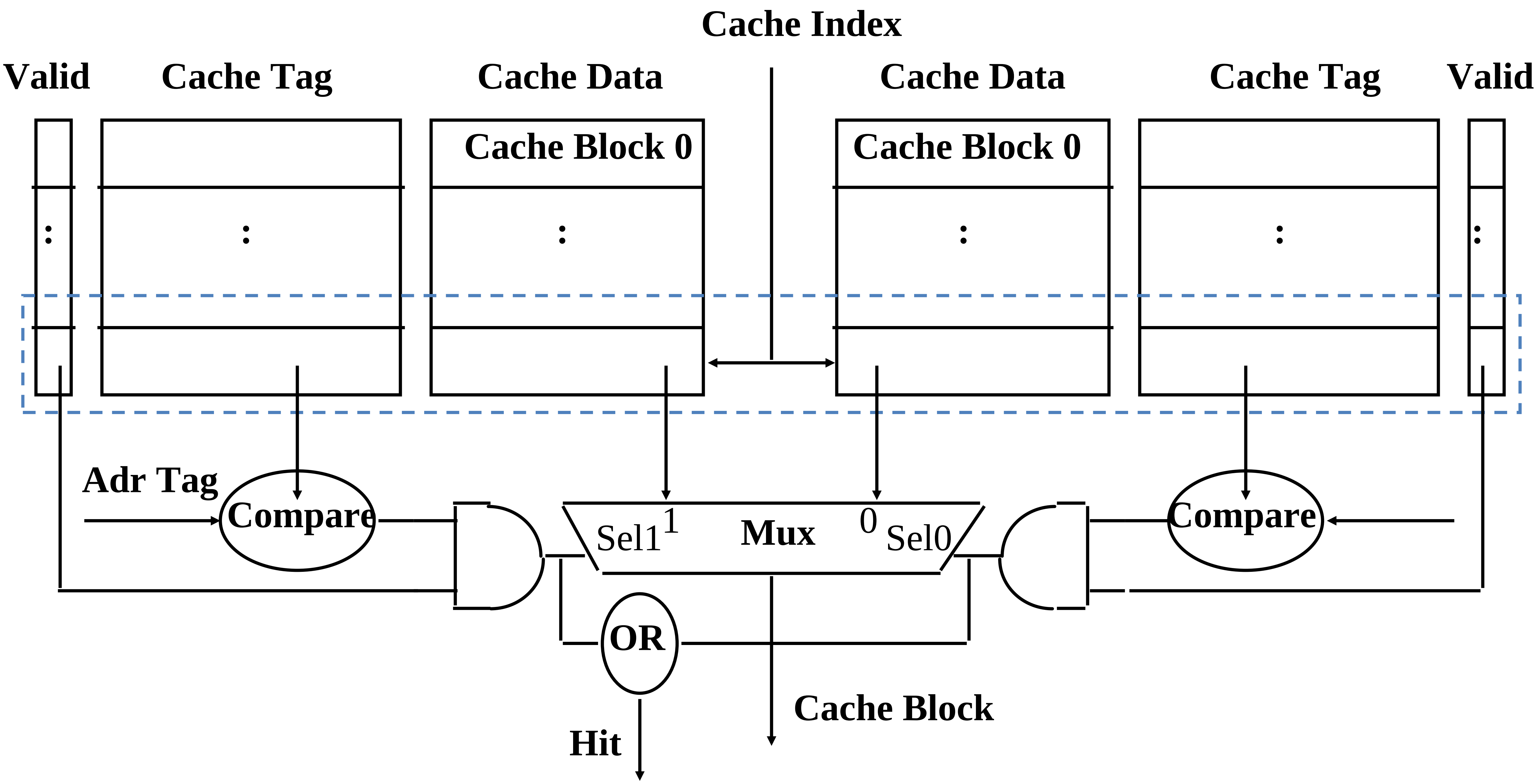
Direct Mapped
only into block 4
(12 mod 8)

block 12 can be placed →

2-way Set Associative Cache (1)

- *N-way set associative*
  - N entries for each cache index (set)
  - Equivalent to N direct mapped caches operating in parallel (N typically 2 to 4)
- Two-way set associative cache
  - Cache index selects a “set” from the cache
  - Two tags in the set are compared in parallel
  - Data is selected based on the tag result

2-way Set Associative Cache (2)

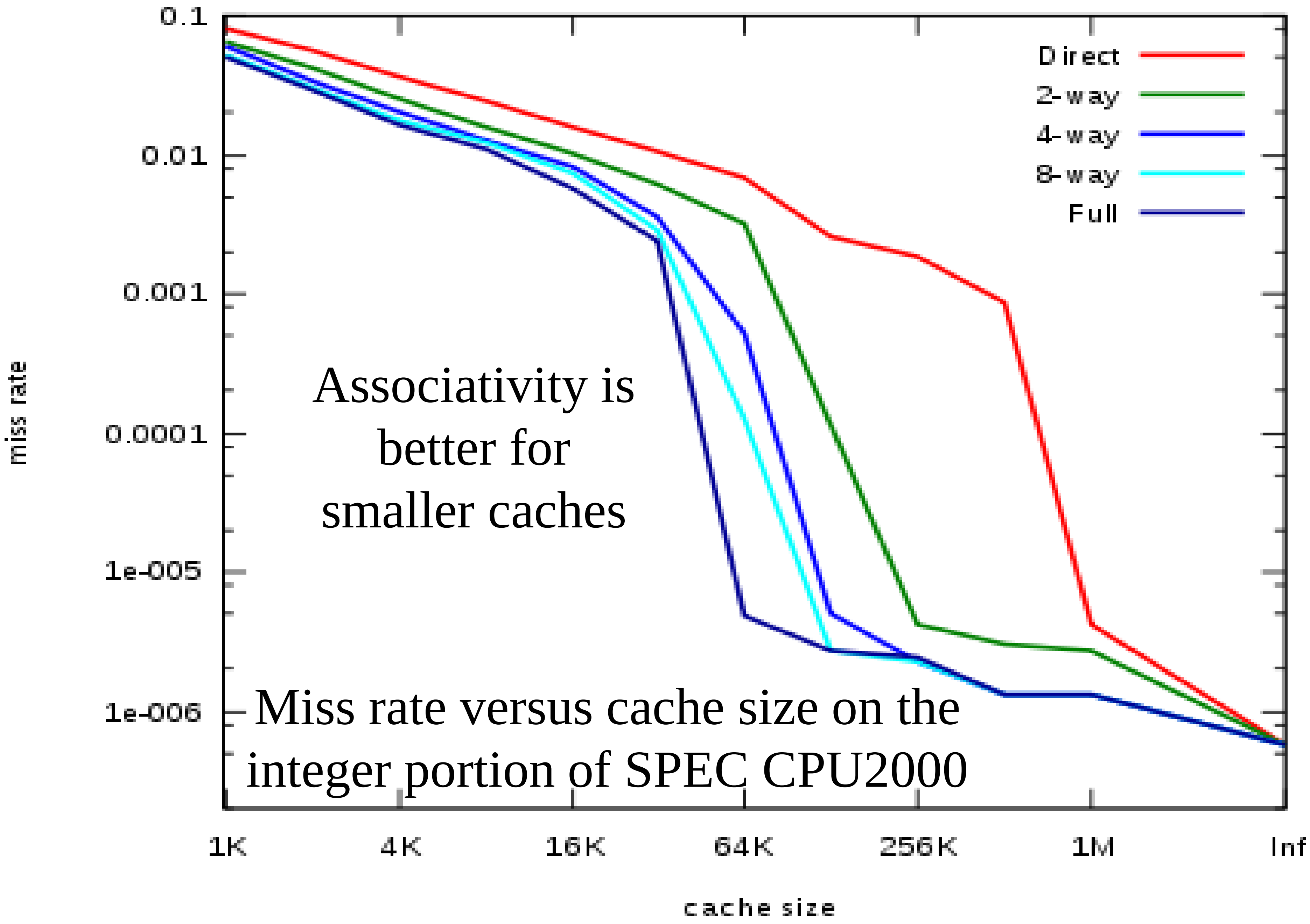


N-way Set Associative versus Direct Mapped

- Advantages
  - Better cache efficiency
  - Miss rate only increases IFF spatial locality increases
- Disadvantages
  - More complex hardware design
    - N comparators versus 1 (for parallel lookup and comparison)
    - extra MUX delay for getting the data
    - more is used
  - Data comes AFTER hit/miss (before with direct mapped)
- Nowadays all caches are associate
  - General observation is that the farther the cache is from the CPU, the higher the associativity degree should be

Performance Gains of Associative Cahces

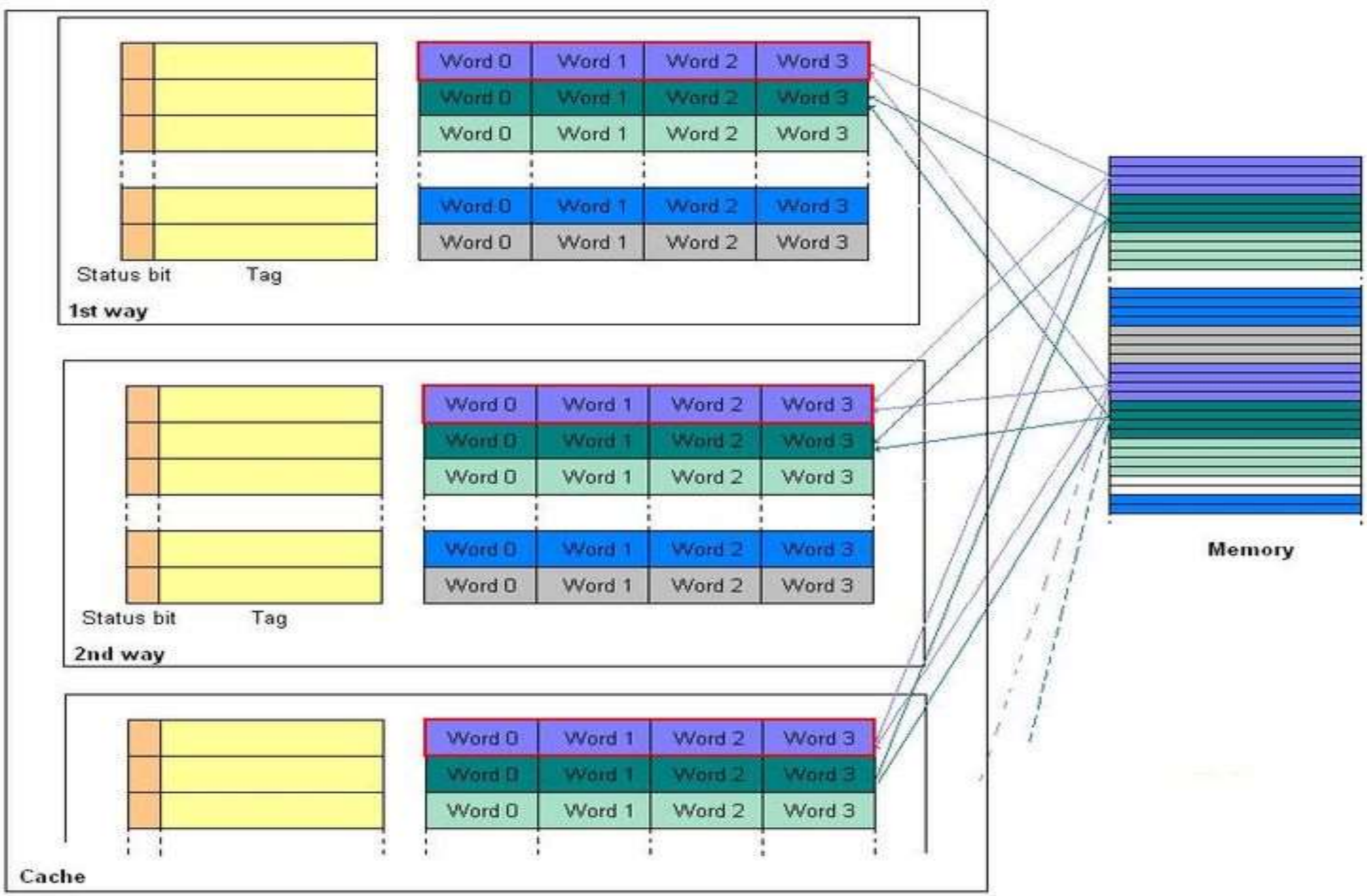
- Compare direct mapped and N-way set associated cache architectures for different cache sizes



Associativity is better for smaller caches

Miss rate versus cache size on the integer portion of SPEC CPU2000

4-way Set Associative Cache



Which block should be replaced on a miss?

- Only 1 choice for a direct mapped cache
- For set associative cache, there are 3 methods
    - *Random* : too expensive in practice
    - *First-in First-out (FIFO)*
    - *Least recently used (LRM)*
- FIFO
    - Easier to implement
- LRU
    - LRU state must be updated on every access
    - Only really feasibly for small set size (e.g., 2-way)
    - Can use pseudo LRU binary tree for larger set sizes
    - A variant is not LRU which is FIFO with exception for most recently used block or blocks

Performance Compared to Random

Associativity	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Fully Associative Caches

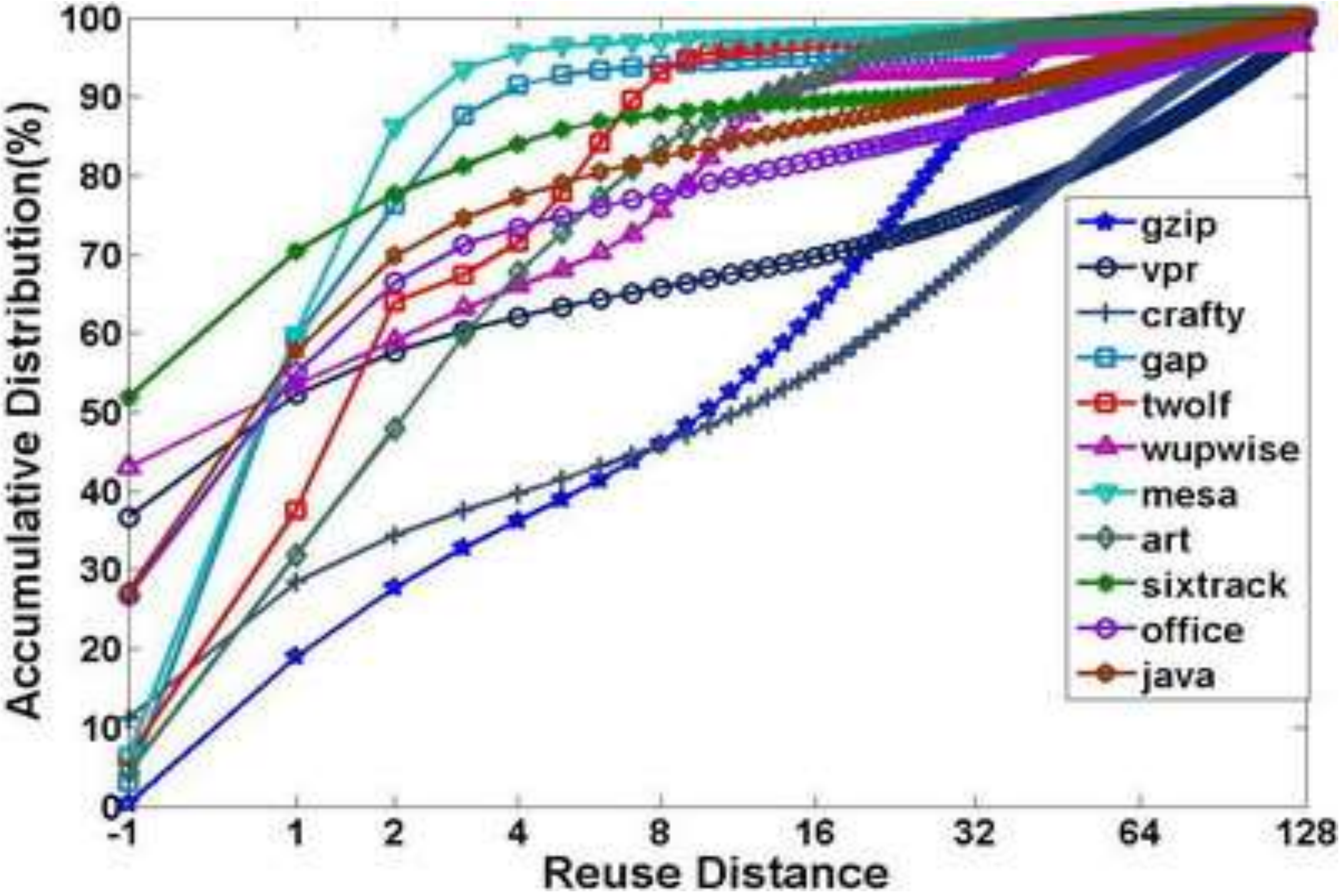
- In general, associative caches attempt to give more opportunities for temporal locality
  - Thus, can lead to better cache efficiency
  - However, there is increasing complexity
- Look-up cost increases with cache size
  - A large cache is more effective
- Look-up in has an high latency
  - Appropriate for higher levels in the memory hierarchy where some cycles can be lost
- Fully associative caches can effectively retain temporal locality windows, but at a cost
  - Cache replacement is the most important factor
- Concept of *stack reuse distance*

Stack Reuse Distance

- How can we quantify temporal locality?
- For any memory reference, build a stack and histogram
- If the reference is the first access to that trace element
  - Set Δ = ∞
  - Increment S(∞) and push the reference on the stack
- If a trace element has been previously referenced, the reference will be already on the stack
  - Let Δ be the distance from the top of the stack to the position at which the reference is found
  - Increment S(Δ)
  - Move the reference to the top of the stack
  - All references between top and Δ are pushed down one position
- Calculate metrics of stack reuse distance

Quantifying Temporal Locality

- Using the stack reuse distance method, we can measure the temporal locality of an application



Compiler and Manual Optimizations

- Restructuring code affects data block access sequence and can lead to improved performance
  - Group data accesses together to improve spatial locality
  - Re-order data access to improve temporal locality
- Use prefetch mechanisms
  - Bring in data items in advanced of them being needed
  - Identified via compiler flags, pragmas, or intrinsics
- Prevent data from entering the cache
  - Useful for variables that will be only accessed once
  - Example: non-temporal (NT) stores on x86
    - data not kept in cache results in 2 to 3 times improvement
- Point to data that should be evicted first

Code Generation for Better Spatial Locality

-
  ```
  do I = 1, 1000
    s = s + A(i) * B(i)
  enddo
  ```

Load fp1, A(1)	Miss	Bring A(1), A(2), A(3), A(4) in cache line 0	
Load fp2, A(2)	HIT		
Load fp3, A(3)	HIT		
Load fp4, A(4)	HIT		
Load fp5, B(1)	Miss	Bring B(1), B(2), B(3), B(4) in cache line 0	Evicts A(1), A(2), A(3), A(4)
Load fp6, B(2)	HIT		
Load fp7, B(3)	HIT		

Code Transformations: Loop Interchange

- Loop interchange
  - Applied to multi-dimension loops
  - Example: assume the array A is stored column-wise

```
DO I = 1, 100
  DO J = 1, 100
    … = A(i,j)
  ENDDO
ENDDO
```

```
DO J = 1, 100
  DO I = 1, 100
    … = A(i,j)
  ENDDO
ENDDO
```

achieves better data locality

- Change array organization
  - Suppose loops can not be interchanged

```
DO I = 1, 100
  DO J = 1, 100
    … = A(i,j)
  ENDDO
ENDDO
```

```
B = transpose(A)
DO I = 1, 100
  DO J = 1, 100
    … = B(i,j)
  ENDDO
ENDDO
```

achieves better data locality

Code Transformations: Array Reshaping

- Change the array in ways to promote good locality
  - Might involve creating temporary arrays
  - Example: assume the array A is stored column-wise

```
A(1000,1000)
DO
…
 DO J=1,100
  DO I=1,100
   ... = A(I,J)
  ENDDO
 ENDDO
…
ENDDO
```

```
TMP(100,100) = A(1:100,1:100)
DO
…
 DO J=1,100
  DO I=1,100
   … = TMP(I,J)
  ENDDO
 ENDDO
…
ENDDO
```

All of TMP is in contiguous memory

achieves better data locality

Other Code Transformations

- Loop fusion
  - Combining 2 or more loops together
- Loop blocking / tiling
  - Accessing the data in the loop in a blocked / tiled way
- C / Fortran optimizations
- Avoidance of false sharing
- *Array of Structures* (AoS)
- *Structure of Arrays* (SoA)

Code Transformations: Loop Fusion

- Combine 2 or more loops together

```
DO I=1,100
  B(I) = C(I) + D(I)
ENDDO
DO I=1,100
  F(I) = C(I) - D(I)
ENDDO
```

```
DO I=1,100
  B(I) = C(I) + D(I)
  F(I) = C(I) - D(I)
ENDDO
```

achieves better data locality

C(I) and D(I) are already in the cache here

Cache Review

- Five key issues
- Location
  - Where goes the memory block ?
- Identification
  - How to find the block of memory ?
- Replacement
  - How to make room for new blocks?

Discussed already

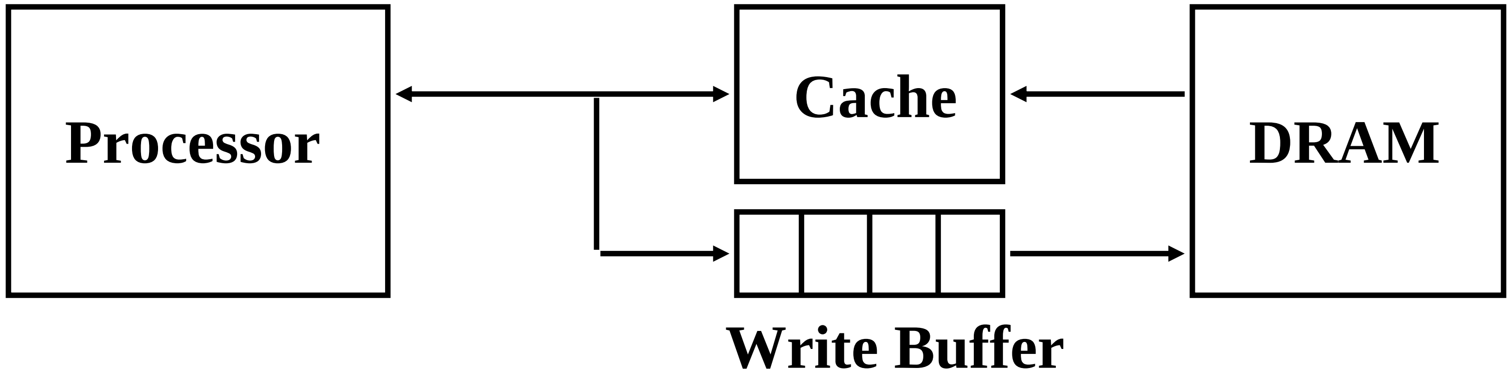
- Policy
  - How to propagate changes on cached data ?
- When to fetch memory block

What happens on a write?

- When a cache block is updated, the result must be written back to the (slower) memory at some point
- *Write through (WT)*
  - When the block in the cache is updated, a write occurs immediately to the memory
- *Write back (WB)*
  - Write occurs when the cache block is replaced
  - Need to keep a bit to indicate if the block is *clean* or *dirty*
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes to same location

Write Buffer for Write Through Cache

- WT always combined with write buffers so that you do not have to wait for write completion



- A write buffer is needed between cache and memory
  - 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
  - Typical number of entries: 4
  - Store frequency << 1 / DRAM write cycle

Write Policy Choices

- Cache hit:
    - *write through*: write both cache and memory
        - generally higher traffic but simplifies cache coherence
    - *write back*: write cache only (not memory)
        - memory is written only when the entry is evicted
    - a *dirty b*it per block can further reduce the traffic
        - avoid to write back blocks which have not been modified
- Cache miss:
    - *no write allocate*: only write to main memory
        - do not bring data in because you are writing to it
    - *write allocate* (aka *fetch on write*): fetch into cache
- Usual combinations:
    - write through and no write allocate
    - write back with write allocate

Dealing with Memory Dependencies (1)

- Consider the following instructions

  st 0(r2), fp2

  …

  ld fp3, 0(r3)

- Do not know the address in r2
- How to resolve whether have memory dependency?
- First step: partial address comparison
  - If the low order bits match, then the second load cannot bypass
- Second step: complete the address comparison
  - If full addresses matched: load can get directly the value from fp2
    - save one memory access
    - most of the time such code pattern is due to a compiler mistake

Dealing with Memory Dependencies (2)

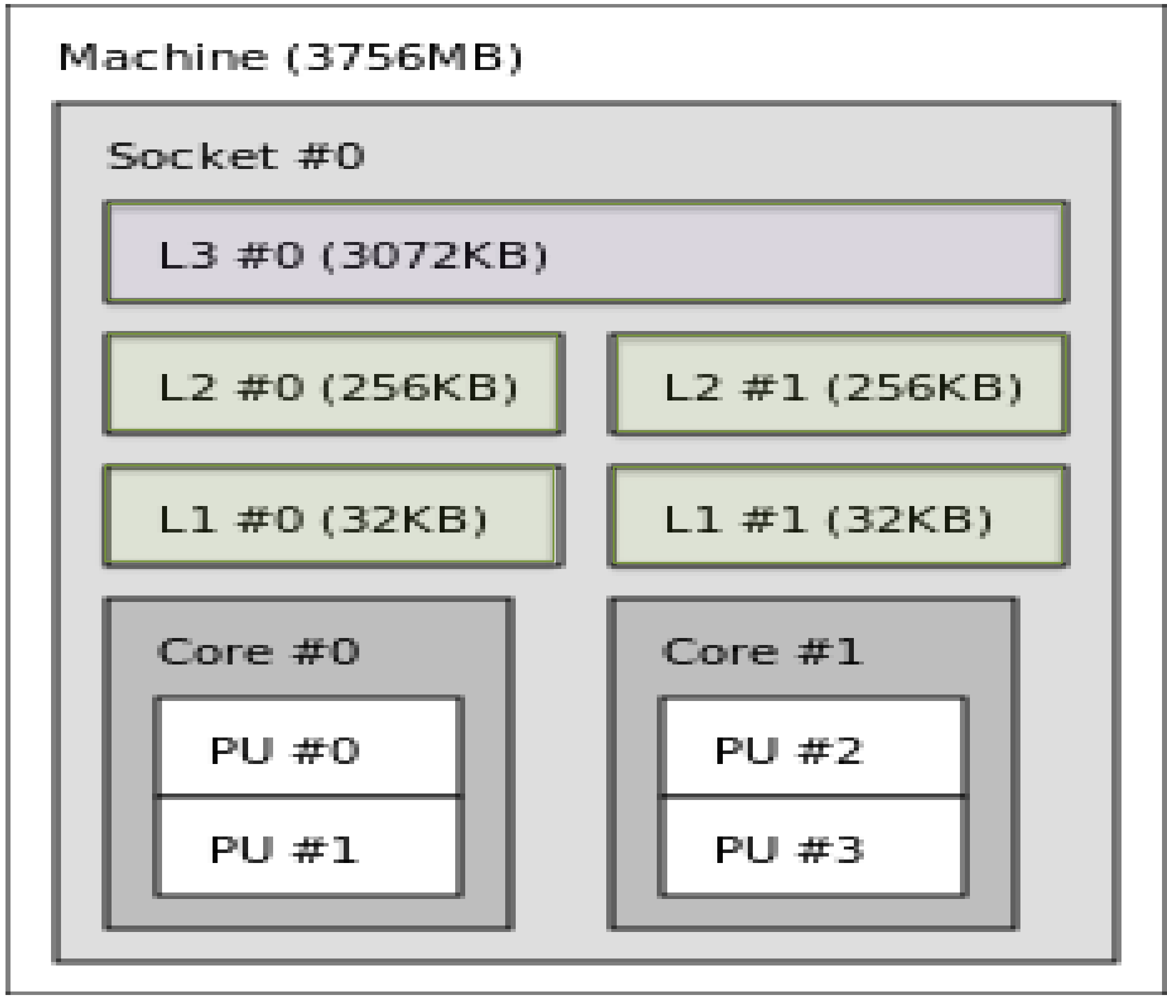
- Consider the following instructions

```
st 0(r2), fp2
…
st 0(r3), fp3
```

- Most architecture perform stores in order (no bypass)
  - Data coherency reasons
  - No delay on dependent instructions
- Pending store instructions are kept in a write buffer
- On advanced architecture, can save memory bandwidth with coalescing
  - Requires full addresses comparison: if 2 stores match coalesced them into a single instruction

State of the Art Cache Memory

- Modern cache hierarchies are complex
- Multiple cache levels
  - From L1 to L3, size and latency increase
  - Private and shared
- Different strategies between the various levels:
  - L1 can be write through and L2 write back
  - BW on chip is cheaper than BW off
- L1 will have much lower degree of associativity than L2 and L3
- Finding out about cache
  - lstopo
    - apt-get install hwloc
  - likwid-topology
  - cat /proc/cpuinfo



private shared

Multilevel Caches and Inclusion Policy

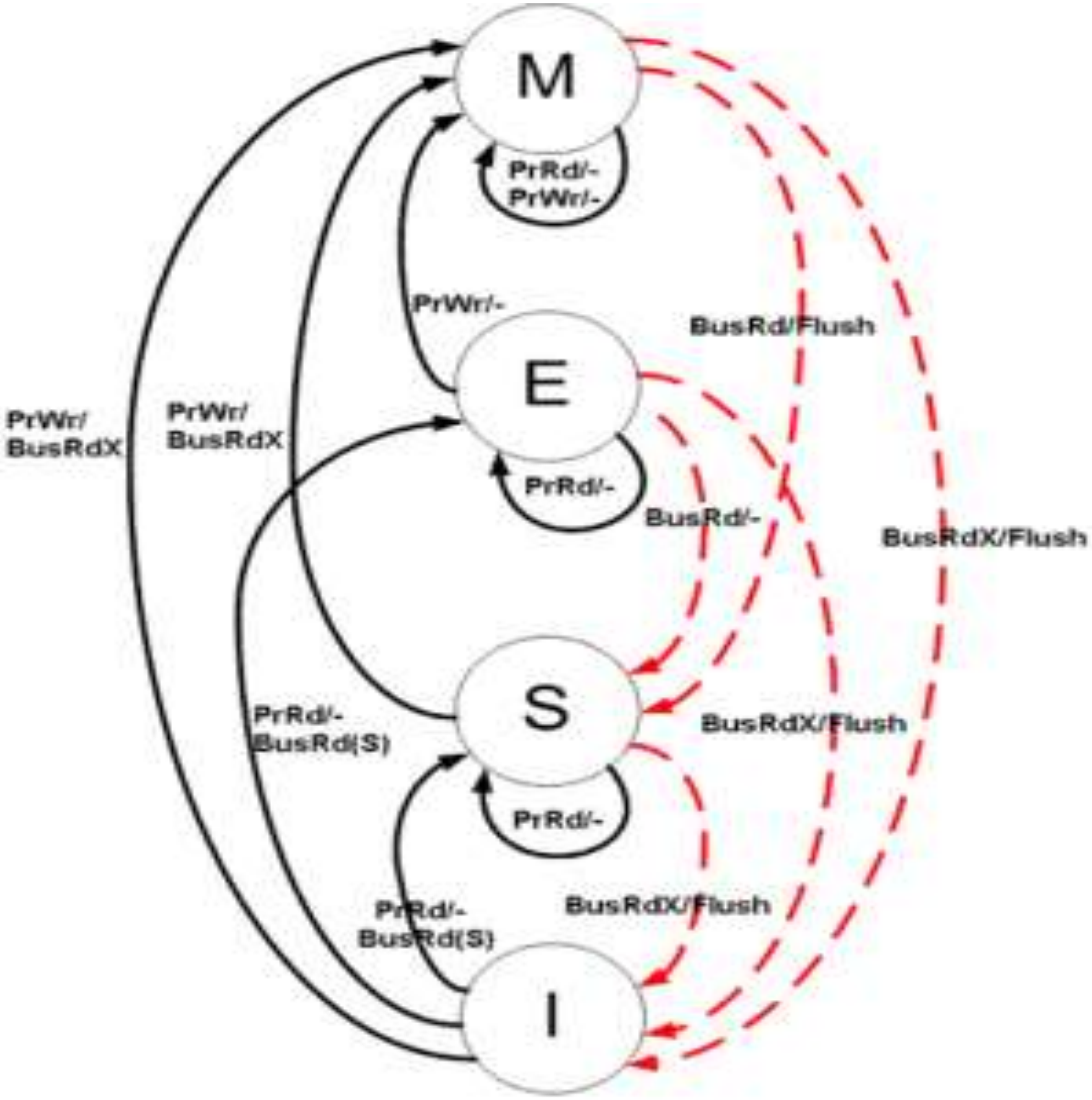
- *Inclusive* multilevel caches
  - Outer cache contains copies of all data from inner cache
  - External access needs only to check outer cache
  - Most common case
  - Does not make efficient use of cache space
  - Not so easy to implement due to replacement policies
- *Exclusive* multilevel caches
  - Inner cache may hold data not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache
  - Makes a better use of cache space at the expense of more complex coherency scheme

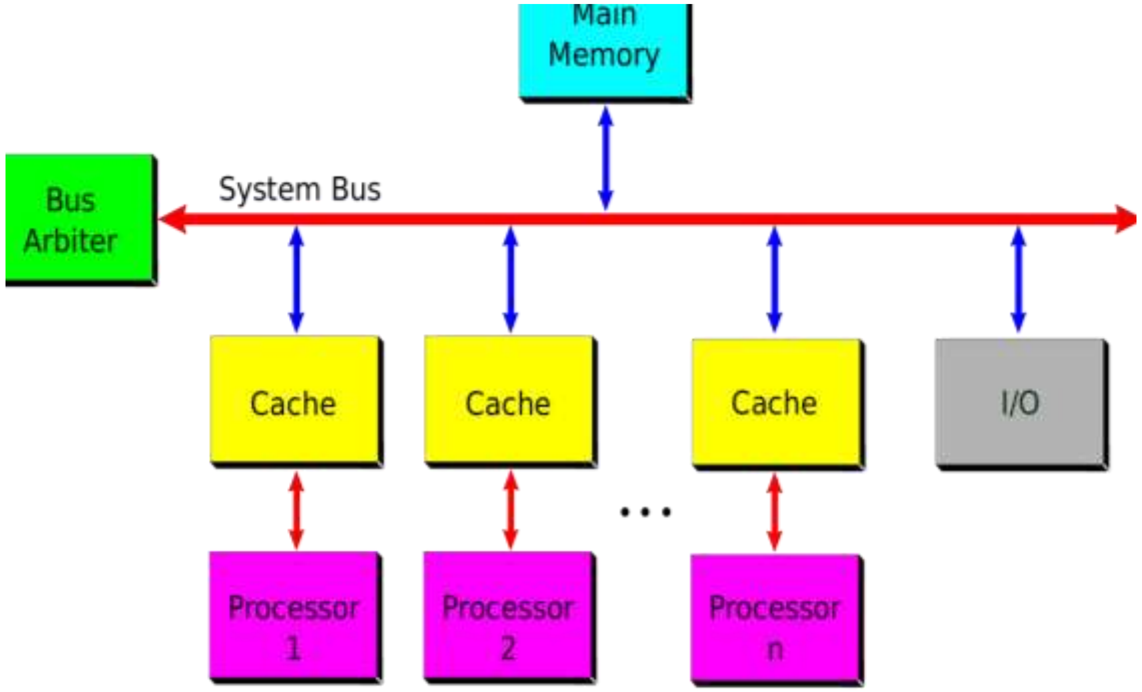
Multicore and Multisocket

- Almost all processors today have multiple cores
- Inclusion and exclusion policies will help manage the L1, L2, and L3 caches between the cores in a processors
- Shared memory computer systems can be built to use multiple sockets (processors) sharing memory
  - Threads and processes share a single address space
  - Raises issues of memory access uniformity
  - Raises more difficult issues of *cache coherency*
- *Uniform Memory Access (UMA)*
  - Latency to memory is the same for all processors
- *Non-Uniform Memory Access (NUMA)*
  - Main memory is distributed across sockets
  - Memory local to a socket has lower latency
  - Can make cache coherency more difficult

Just a Glimpse of Cache Coherency (1)

- Machines with multiple caches used by multi-threaded programs must address cache coherency

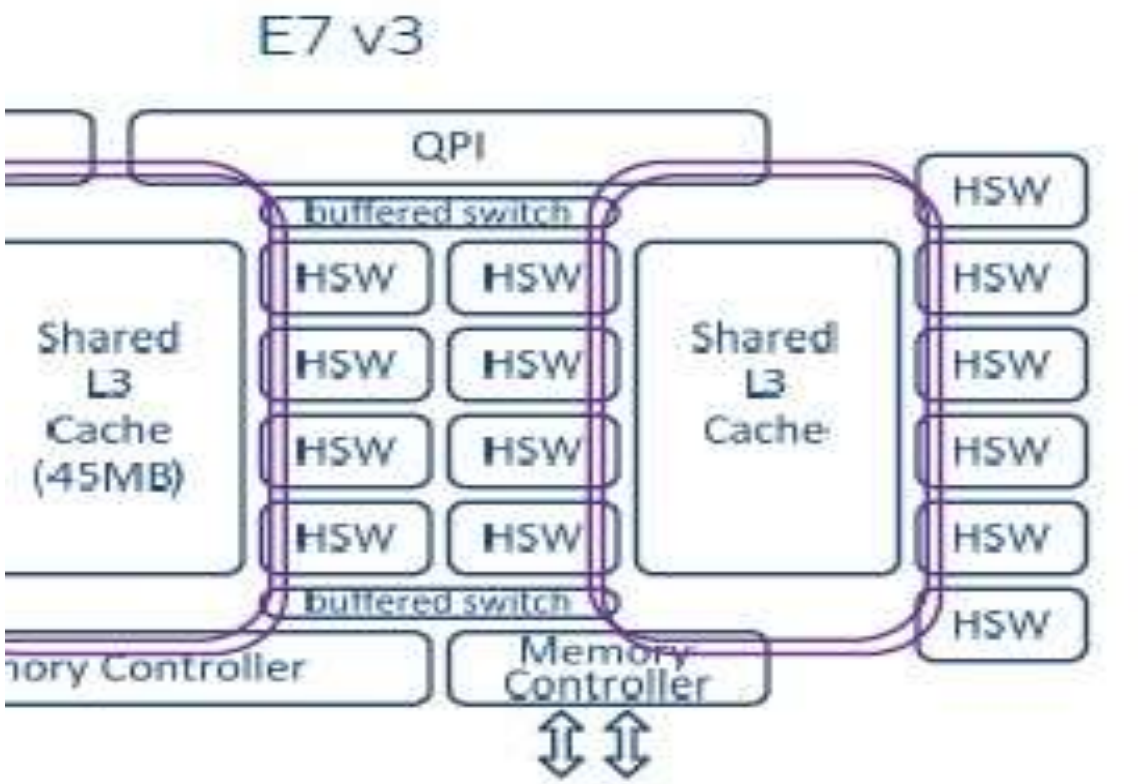


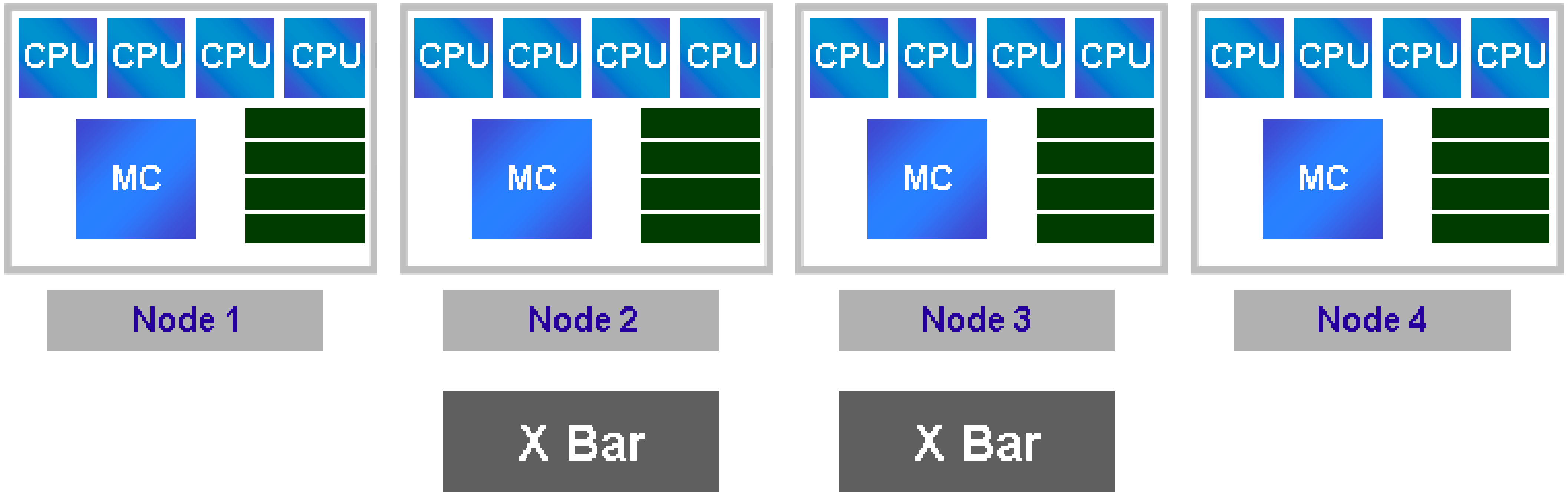


Modified Exclusive Shared Invalid (MESI) protocol is built into modern processors

Just a Glimpse of Cache Coherency (2)

- MESIF
  - Optimization of the MESI protocol for multisocket
  - F state specialized version of the S state
- Marks the cache as responsible for copy propagation





Advanced Cache Features

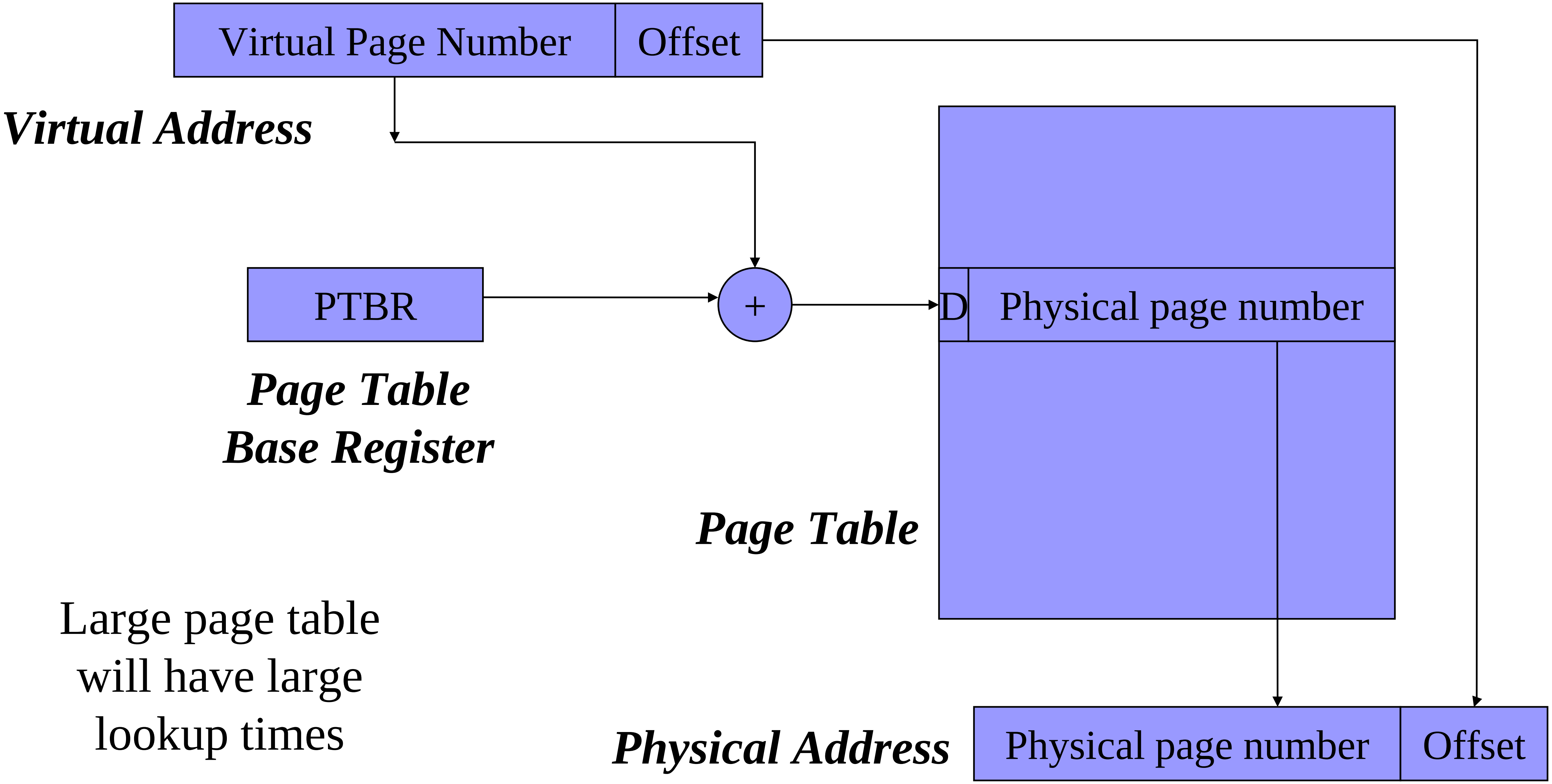
- Shared memory addressing and cache coherency are challenges for multisocket, multicore systems
- Further complicated by virtual memory
- Need to look at other memory address translation hardware and how caches are integrated
- Refresher on virtual memory
  - What is virtual memory?
  - Virtual address translation
  - Translation lookaside buffer

Virtual Memory

- Processes are given a virtual address space
  - All logical addresses issued by the processor are virtual
  - Must be mapped to physical adddresses
- OS is in charge of maintaining correspondence between virtual and real (physical) memory addresses
  - It does so through address translation
  - Partitions the virtual address space into pages
  - Translation between virtual and physical page stored in the page table
- Virtual memory was introduced as an alternative to overlays
  - Provides protection for the same price (thread vs process)
  - Allows multiple processes to share limited physical memory
  - Reduced the complexity of programming
- Virtual memory only became possible with hardware
  - Translation of virtual addresses to physical address efficiently
  - Becomes more complicated with caches and cache coherence

Virtual Address Translation

- Page size and the # virtual address bits determines the size of the page table and # bits of offset



Virtual Page Number

Offset

Virtual Address

PTBR

Page Table Base Register

+

D

Physical page number

Page Table

Physical Address

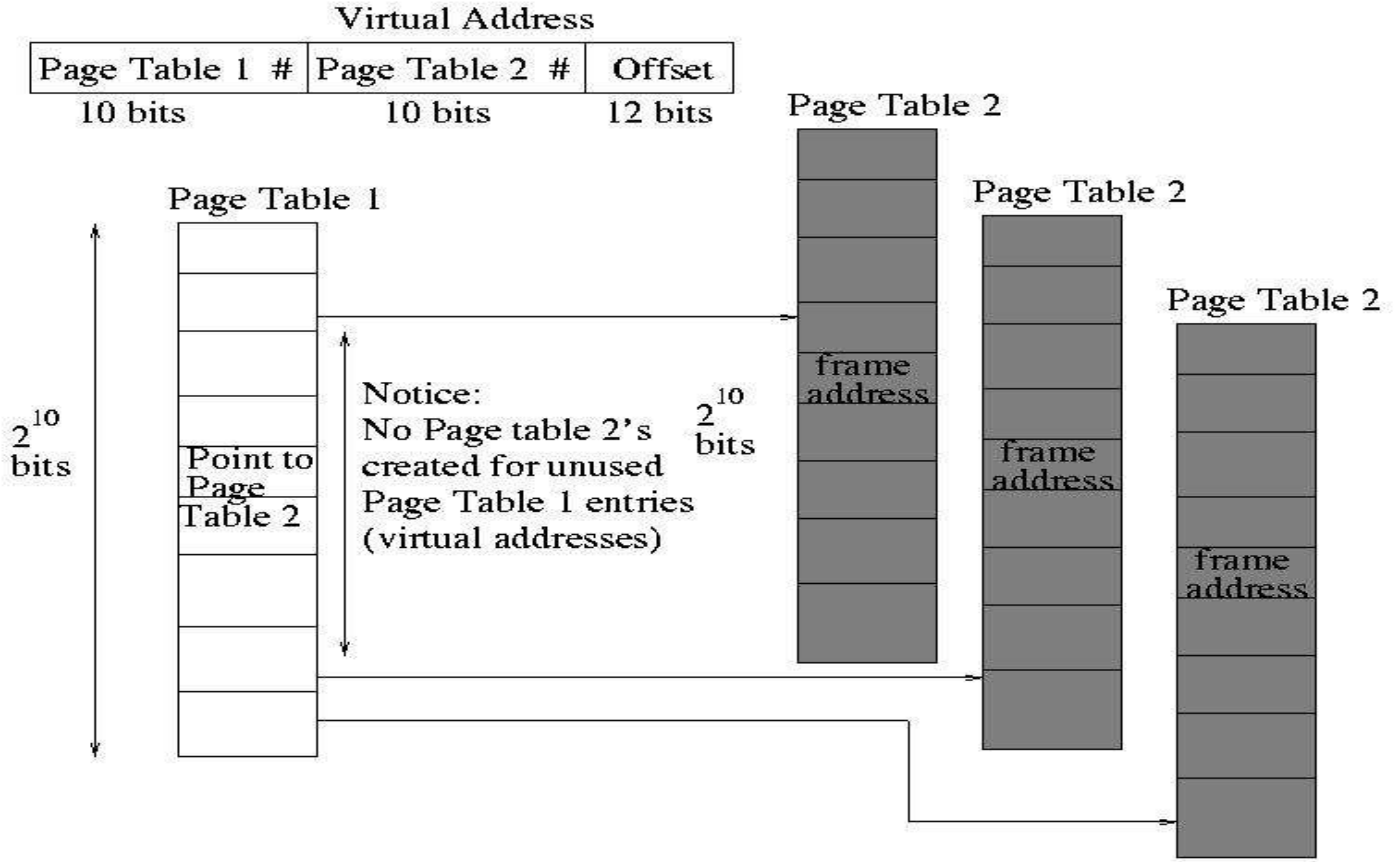
Physical page number

Offset

Large page table
will have large
lookup times

Multilevel Page Tables

- Multiple indexed page tables are used when a single page table gets too large



Page Faults

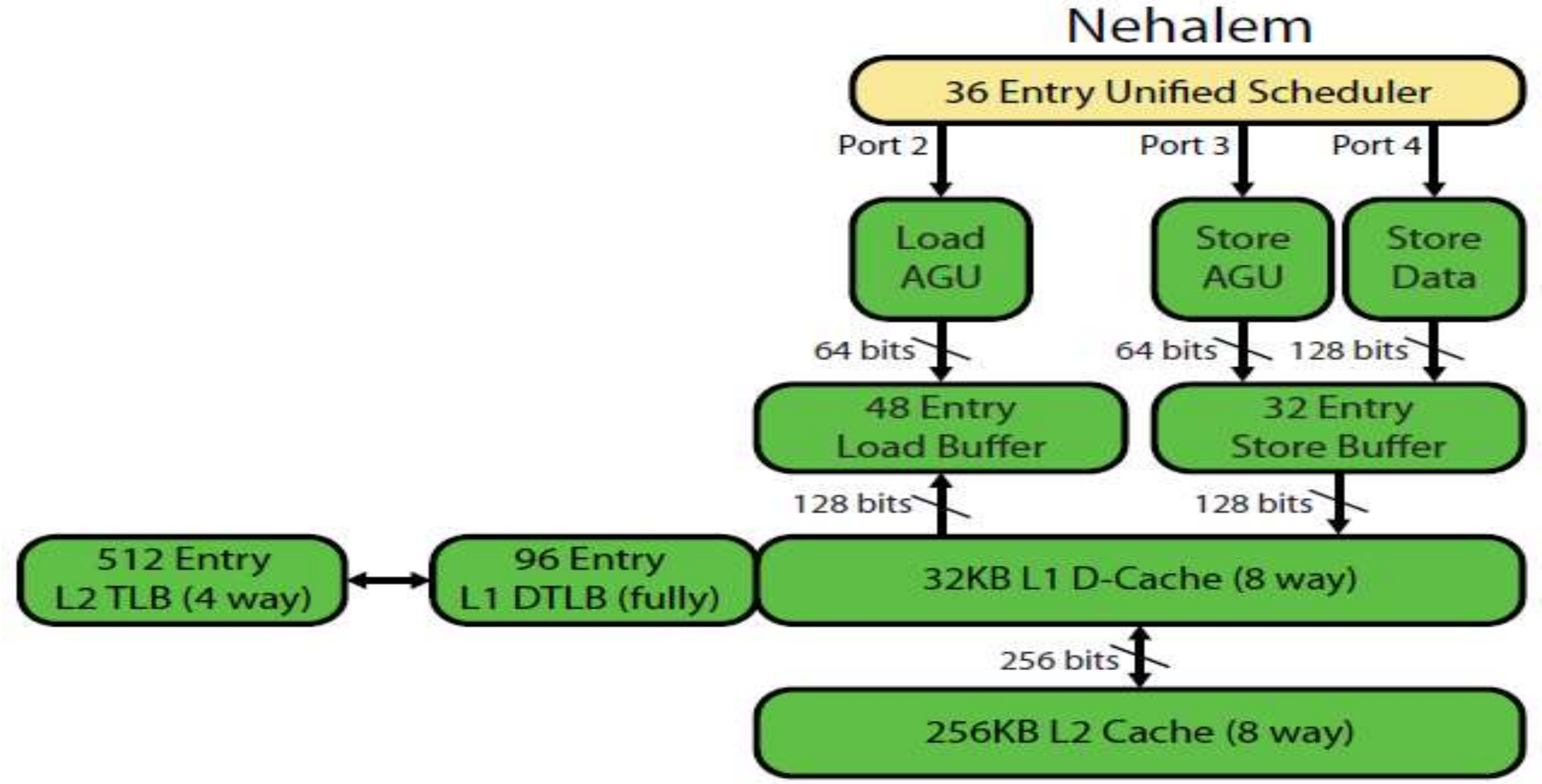
- Virtual memory systems are used to manage the limit amount of physical memory
- A *page fault* occurs when a virtual page is not found in main memory
    - Call the OS (exception)
    - Invoke OS page fault handler
    - Find a physical page (eventually, evict)
    - Switch to other task ready to run (context switch)
- Page faults are extremely costly and should be avoided at all costs

Computing Physical Addresses Fast

- Finding the actual physical address for a virtual memory reference is on the critical path
- Efficient instruction processing depends on it
- Need to determine physical address of pages in memory very fast
- Use a translation lookaside buffer (TLB)
  - Special “cache” of recently used virtual pages descriptors
  - Allow quick mapping to physical page locations
  - Cache the last virtual address translation result
- In general very few entries in the TLB
  - For instancce, 256 to 512
- TLB misses are costly
  - Can have multiple levels of TLB to increase coverage
  - Or go to page tables, which might be in cache

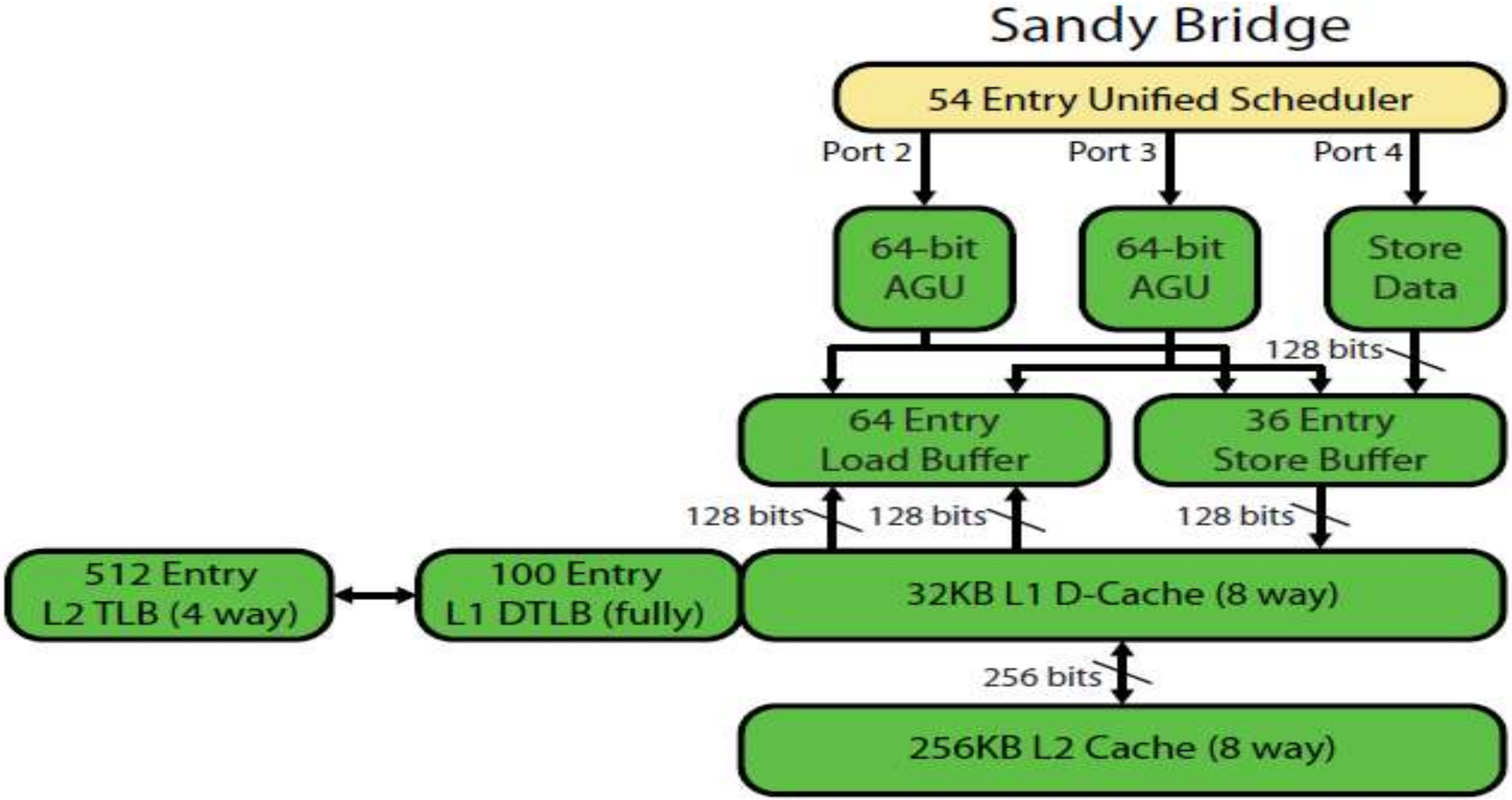
Examples of TLB and Cache Hierarchy (1)

- Intel Nehalem processor



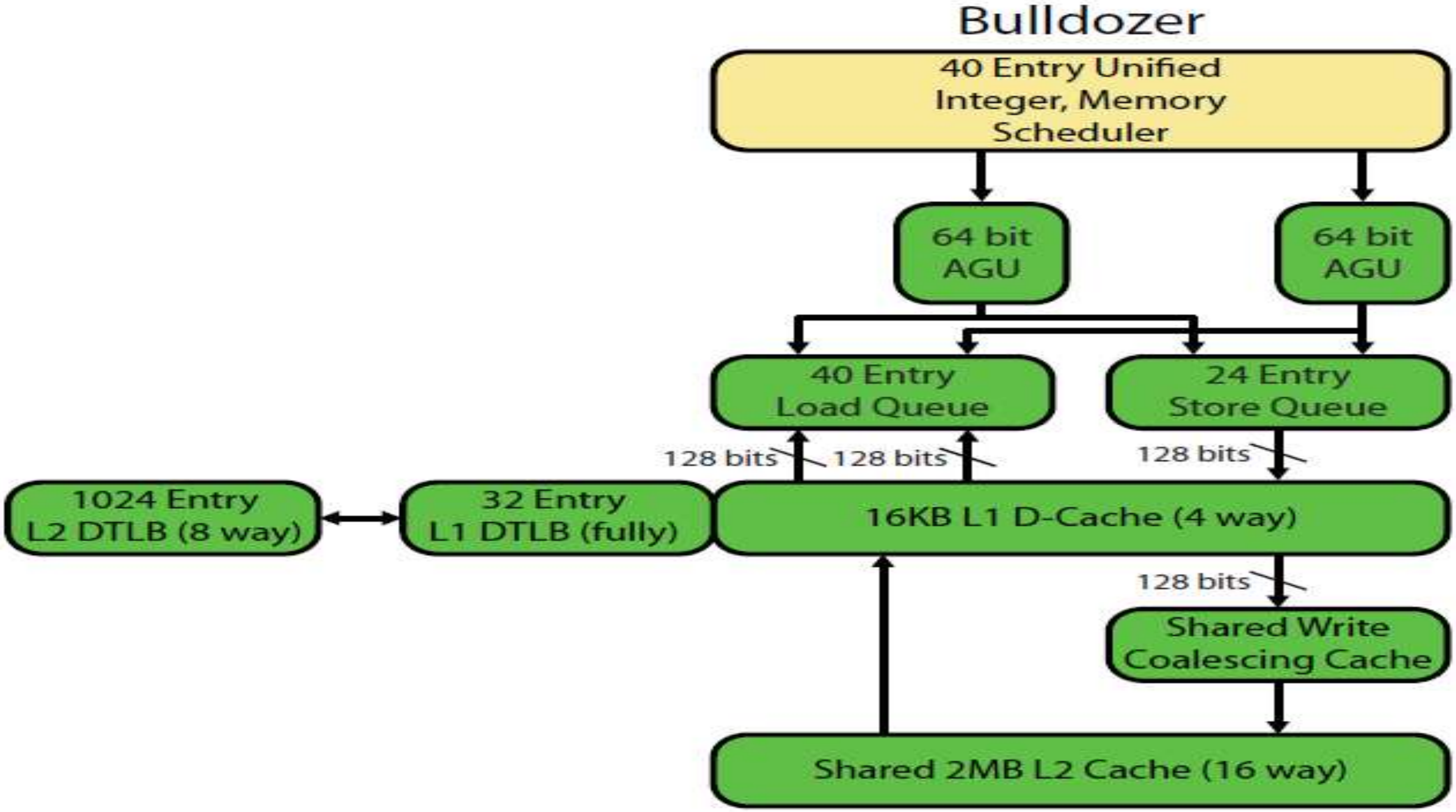
Examples of TLB and Cache Hierarchy (2)

- Intel Sandy Bridge processor



Examples of TLB and Cache Hierarchy (3)

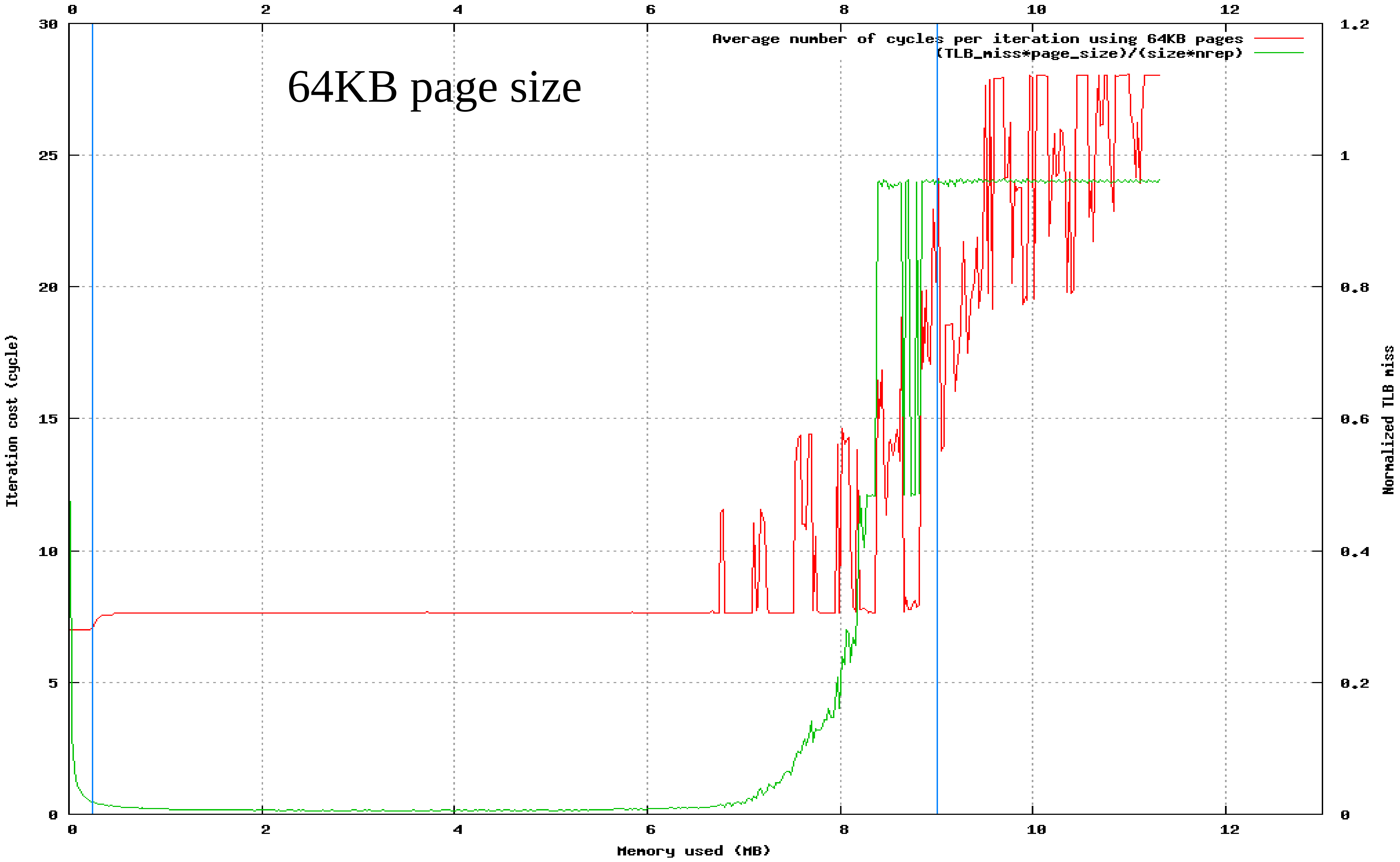
- AMD Bulldozer processor



Virtual Addresses and Cache

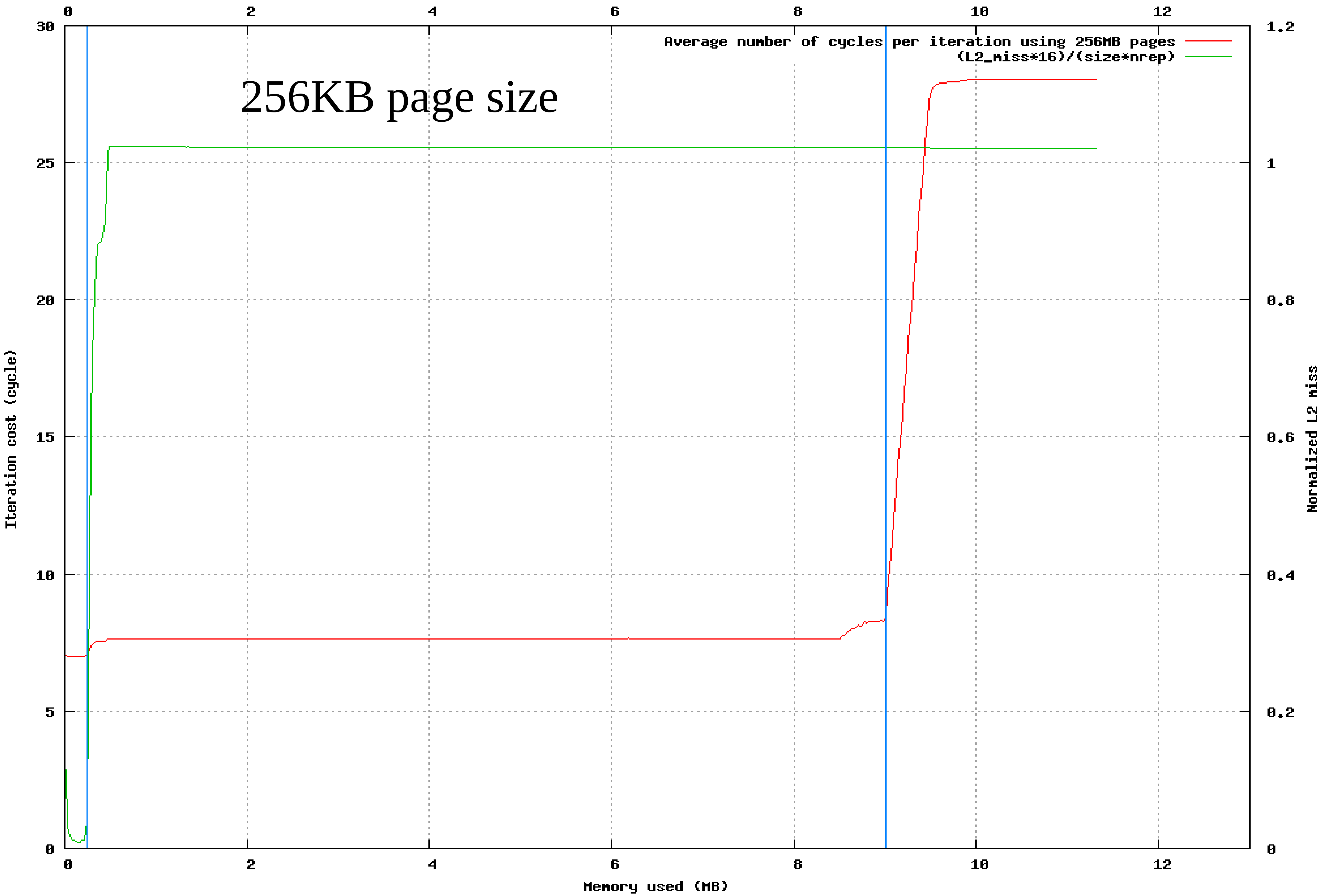
- Is a real or virtual address used to determine the cache slot?
- *PIPT* caches : physical address for index and tag
  - [+] simple, solve aliasing problem
  - [-] slow : TLB access prior to cache look-up
    - potential TLB miss + access to memory prior to know if the data is cached!
- *VIVT* caches : virtual address for both index and tag
  - [+] Fast look-up (no TLB access) prior to cache
  - [-] Aliasing problem : multiple virtual addresses pointing to identical same physical address
    - consume multiple cache lines for the same memory address (coherence issue)
  - [-] Homonym problems : same virtual address maps to multiple physical addresses
    - cache flush after context switch
- *PIVT* caches : physical address for index, virtual address for tag

Performance on Itanium with 9MB



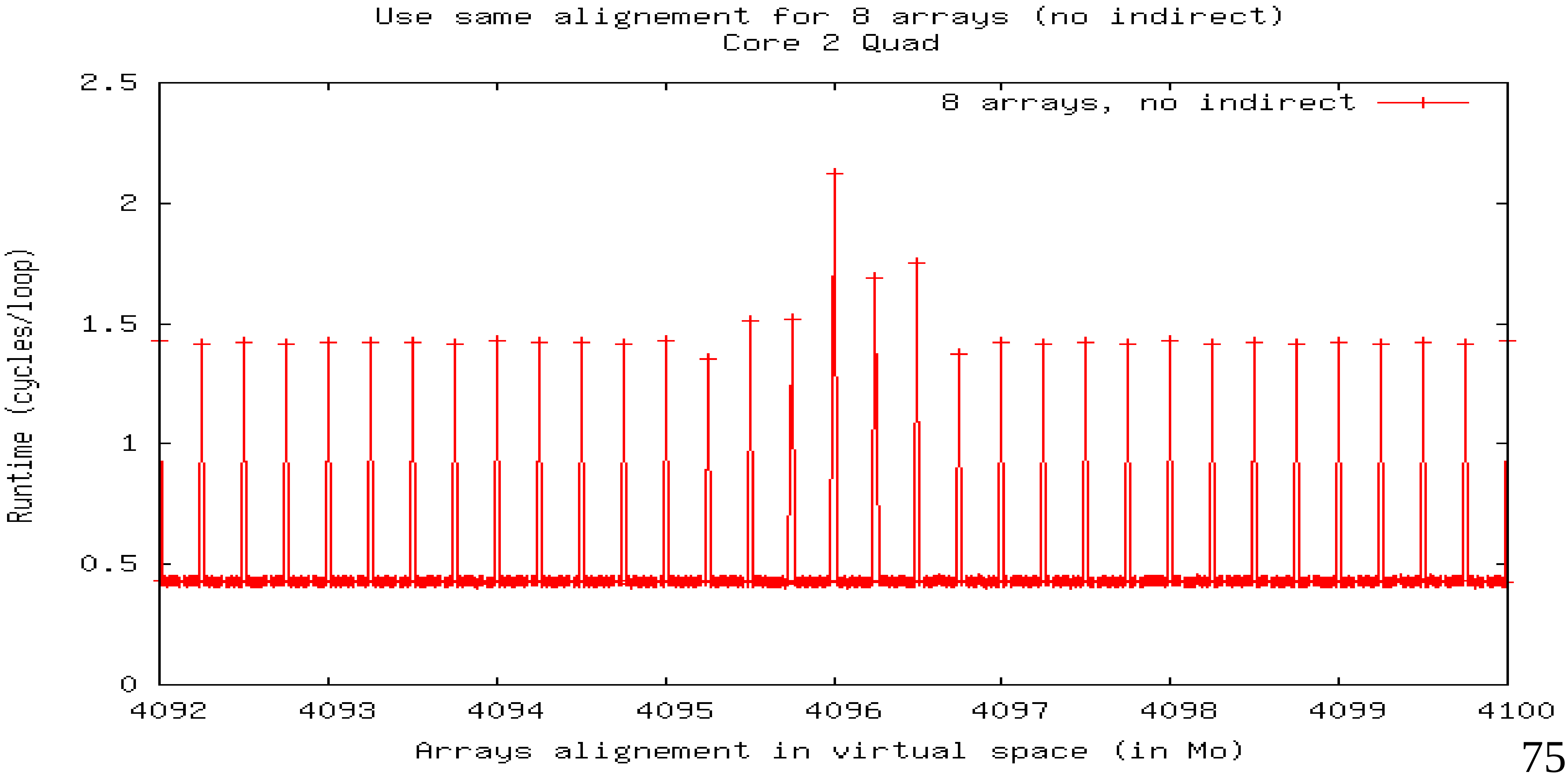
64KB page size

Performance with Large Pages


256KB page size

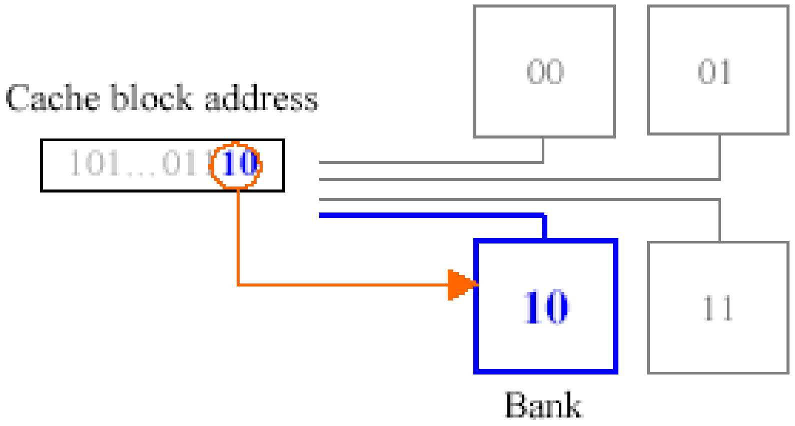
Affect of System Calls on Caches

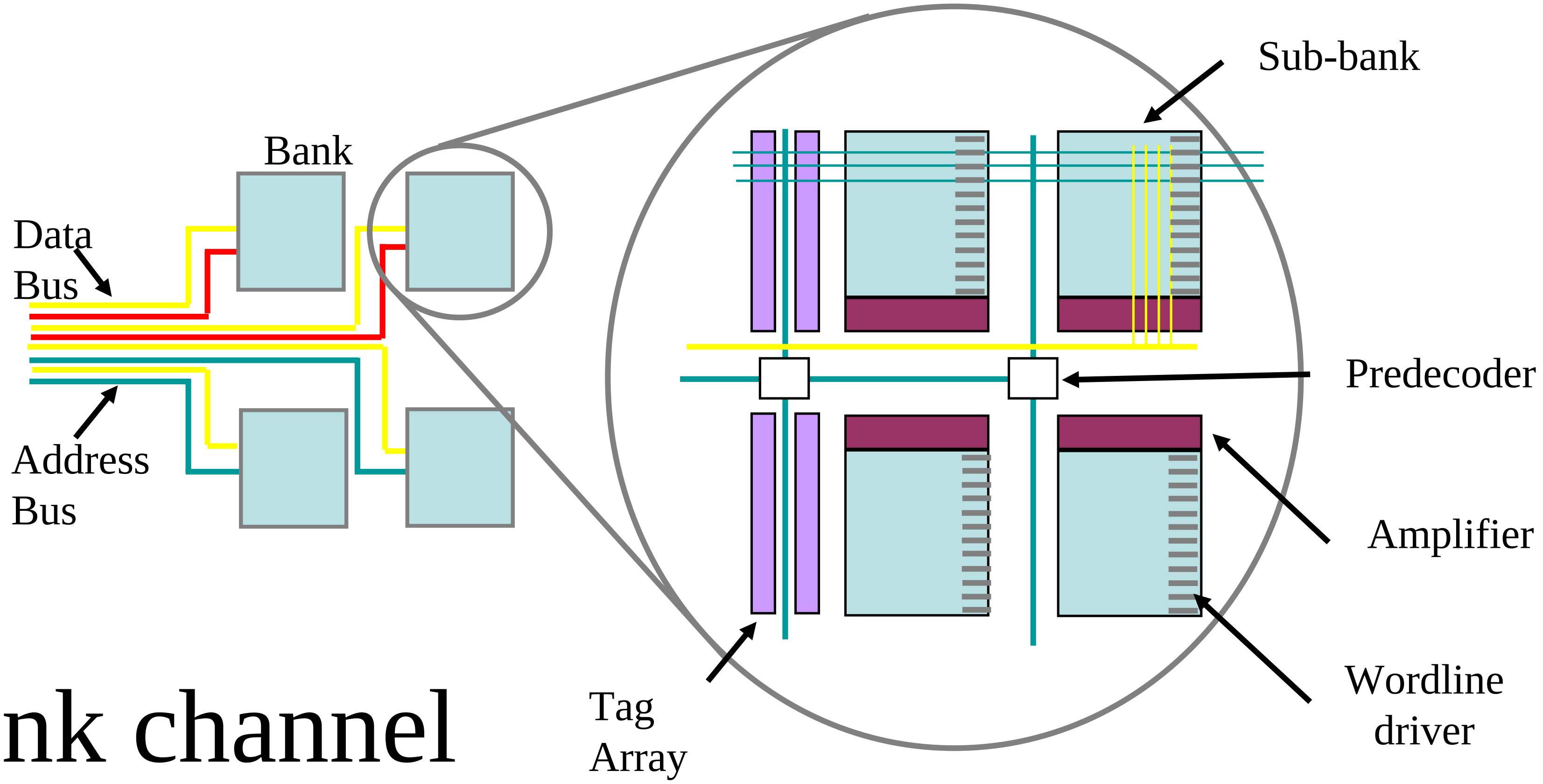
- Cache associativity can be killed in case of resonance with memory allocator
  - All arrays starts at address 0 modulo 2048



Beyond Associativity – Static NUCA(1)

- Non-Uniform Cache Access (NUCA)

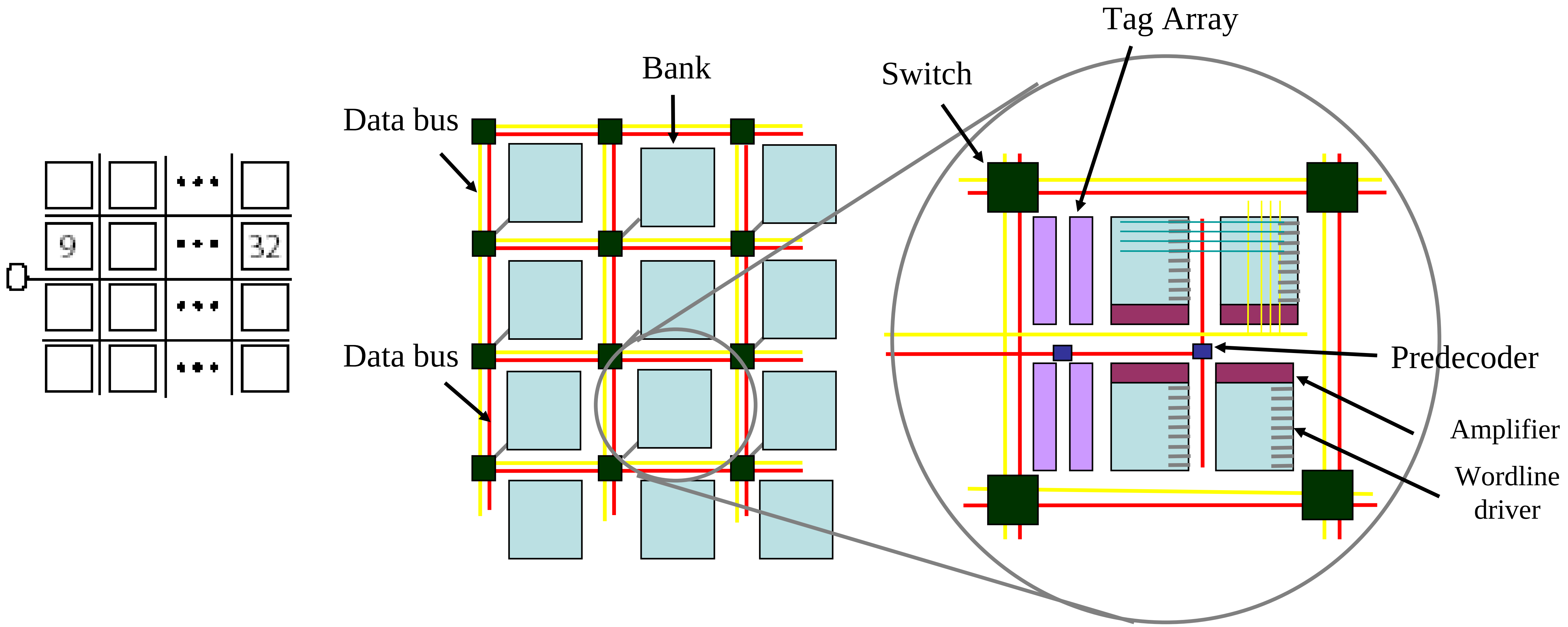




- Use private per-bank channel
- Each bank has its distinct access latency
- Statically decide data location for its given address
- Average access latency = 34.2 cycles
- Wire overhead is an issue (20.9% in this design)

Courtesy of Prof. Hsien-Hsin S. Lee

Beyond Associativity – Static NUCA (2)



- ❑ 2D switched network alleviates wire area overhead
- ❑ Average access latency =24.2 cycles
- ❑ Wire overhead = 5.9%

Courtesy of Prof. Hsien-Hsin S. Lee

Beyond Associativity – Dynamic NUCA

- Data can dynamically migrate
  - Move frequently used cache lines closer to CPU
- Use associativity!
  - Each data is eligible in any of X-way (e.g,. 4-way)
  - Pick the way closest to the core is the data is hot

Courtesy of Prof. Hsien-Hsin S. Lee

Cache Summary

- Larger size
  - + Decrease probability of capacity and conflict miss
  - – Increase cache hit latency
- Higher associativity
  - + Decrease probability of capacity and conflict miss
  - – Increase cache hit latency
- Larger cache block (line)
  - + Decrease compulsory and capacity miss (reload)
  - – Increase conflict misses and cost of misses

Little's Law

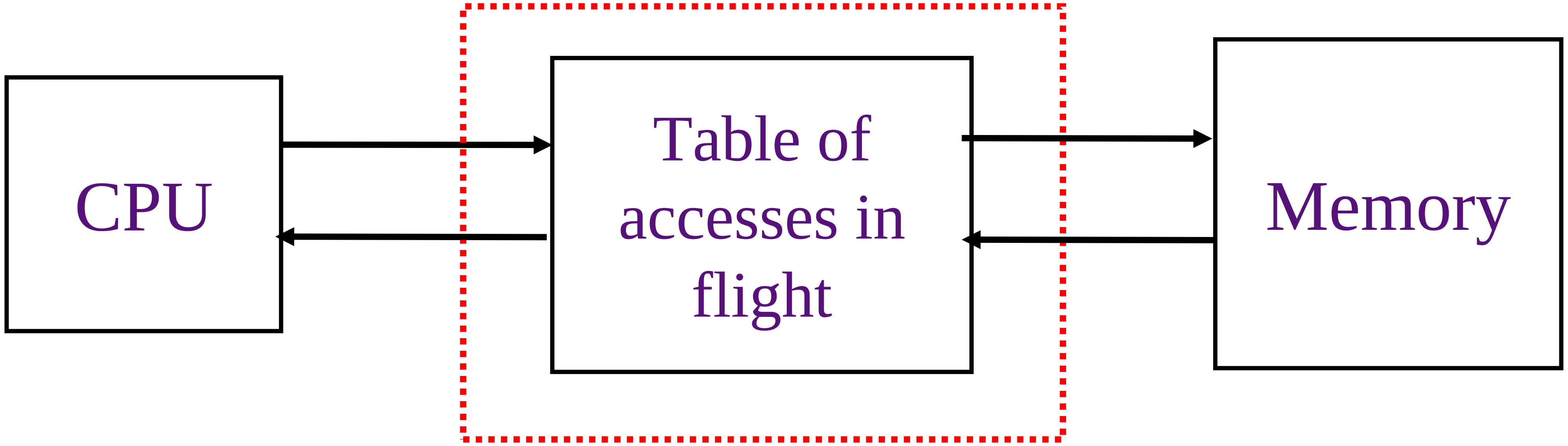
- Little's Law looks at the relationship between:
  - *Throughput* of a system (work completed per second)
  - *Capacity* of the system to do work (concurrency)
  - *Latency* required for each work item in the system

$$Throughput = Capacity \,/\, Latency$$

- Little's Law can be applied anywhere in a system
- Capacity could be different things
  - Instructions, memory references, …
- We are interested in increasing throughput
  - Could increase capacity
  - Could reduce latency
- Notion of "hiding latency"

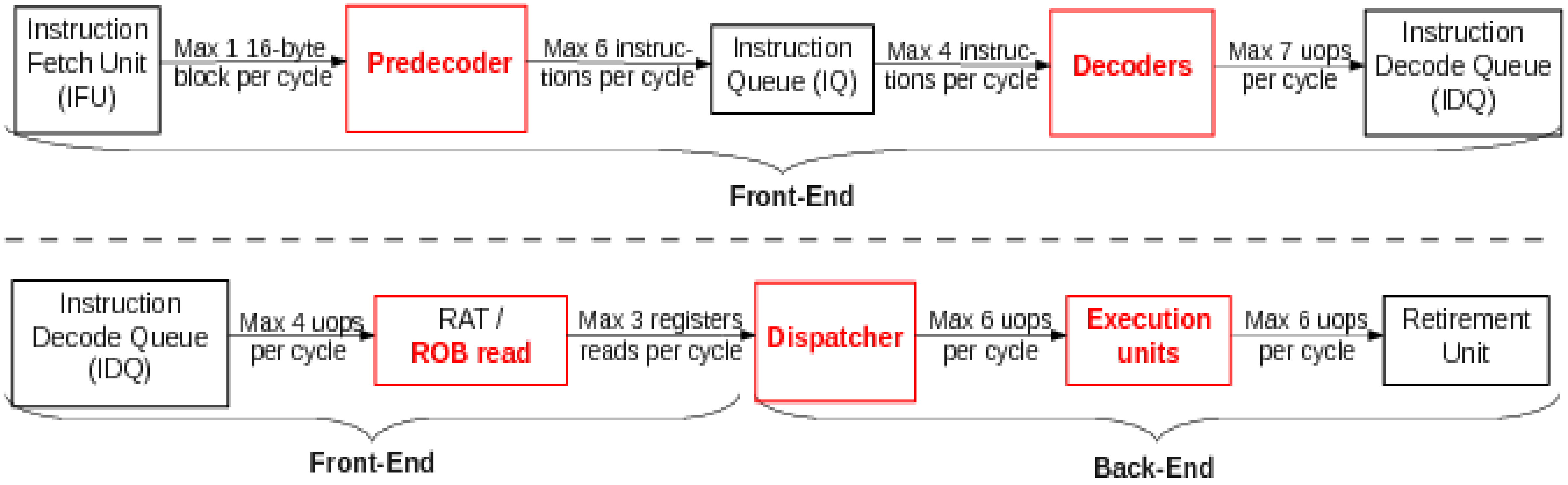
Little's Law and Memory

- Think about # outstanding memory references
- Think about # cycles per memory reference



- Example:
  - Assume infinite bandwidth memory system
  - Each memory reference takes 100 cycles
  - How many memory references need to be outstanding to sustained a throughput of 1.2 memory references / instruction?
- Think about pipelining the memory system
  - # memory references in flight = 1.2 * 100 = 120 table entries

Pipeline Modeling: Core2


Instruction Fetch Unit (IFU)
Max 1 16-byte block per cycle
Predecoder
Max 6 instruc-tions per cycle
Instruction Queue (IQ)
Max 4 instruc-tions per cycle
Decoders
Max 7 uops per cycle
Instruction Decode Queue (IDQ)
Front-End
Instruction Decode Queue (IDQ)
Max 4 uops per cycle
RAT / ROB read
Max 3 registers reads per cycle
Dispatcher
Max 6 uops per cycle
Execution units
Max 6 uops per cycle
Retirement Unit
Front-End
Back-End

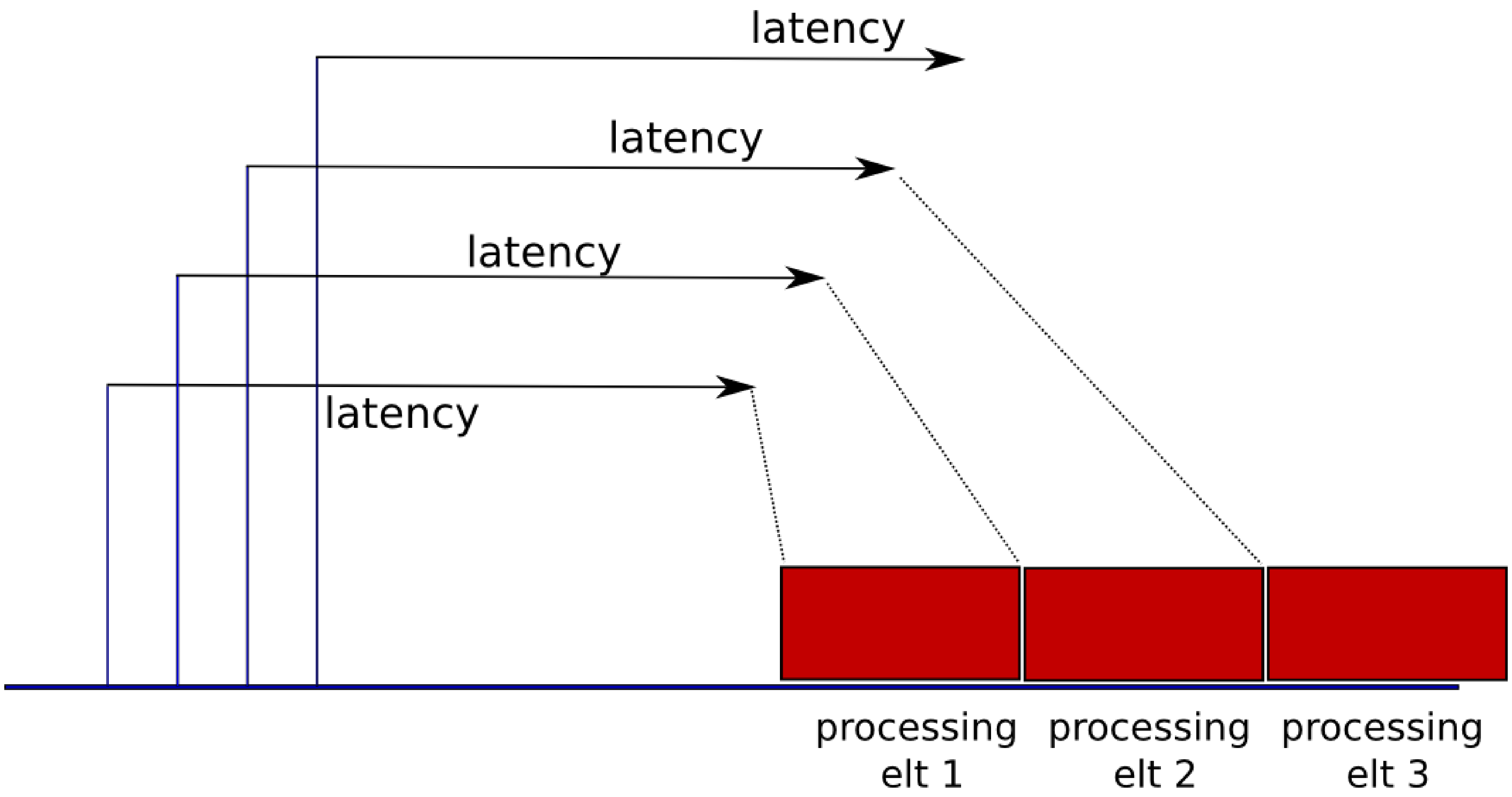
Courtesy of Souad Koliai

Reducing Miss Probability

- Large lines exploit spatial locality
  - Bet on a linear access
  - More complex patterns yet structured patterns occurs.
  - Is it possible to use the past to predict the future ?
  - To anticipate data, to prefetch them ?

Hardware Prefetching (1)

- Prefetching is the idea of requesting memory in advance of when the data is needed
  - It is a way to hide latency of memory referencing
- Need hardware to implement prefetch instructions



Hardware Prefetching (2)

- Need compiler help to find opportunities for prefecting
  - Can detect patterns in the stream of memory accesses
- Main advantage is in tolerating latency
  - It does not decrease latency!
- Prefetching is speculative
  - If prefetch memory references that are beyond a control point in the program, risk wasted resources
  - Prefetching can affect cache usage and space
  - It requires additional memory bandwidth

Hardware Prefetching (3)

- How to detect patterns ?
  - Eavesdrops all addresses accessed
  - Detect constant stride
- Keep track of last accessed addresses
  - Cheap MSHR already provided
  - @, @+1, @+2 easy to detect
- If constant stride detected then allocate a prefetch stream
  - Number of prefetch streams has a criteria

Hardware Prefetching (4)

- More complex patterns ?
  - Stride computation easy to overlap due to long latency memory access
- High number of instructions in-flight
  - ~100 memory instructions on Sandybridge
- Interleave stream makes patterns detection difficult
  - Use IP as an filter
- IP base hardware prefetcher
  - Decrease risk of mis-prediction
  - Decrease pattern detection latency

Hardware Prefetching

- Sandybridge has multiple hardware prefetchers
  - Stream behavior may differ depending on cache level
  - Implementation trade-off (silicon cost, latency) differ
- DCU (Data Cache Unit) prefetcher from L2 to L1
  - Stream prefetch
  - Ascending order of consecutive cache lines
- IP-Based (IPP) HW prefetchers from memory to L2
  - IP based
  - Detect stride of up to 2K
  - Run-ahead distance: 8 lines

What's next?

- 3D stacked cache layer
  - KNL / Knight Hill
- 3D X point (NVM/ SCM)
- Unified memory space
  - GPU