EITF20: Computer Architecture

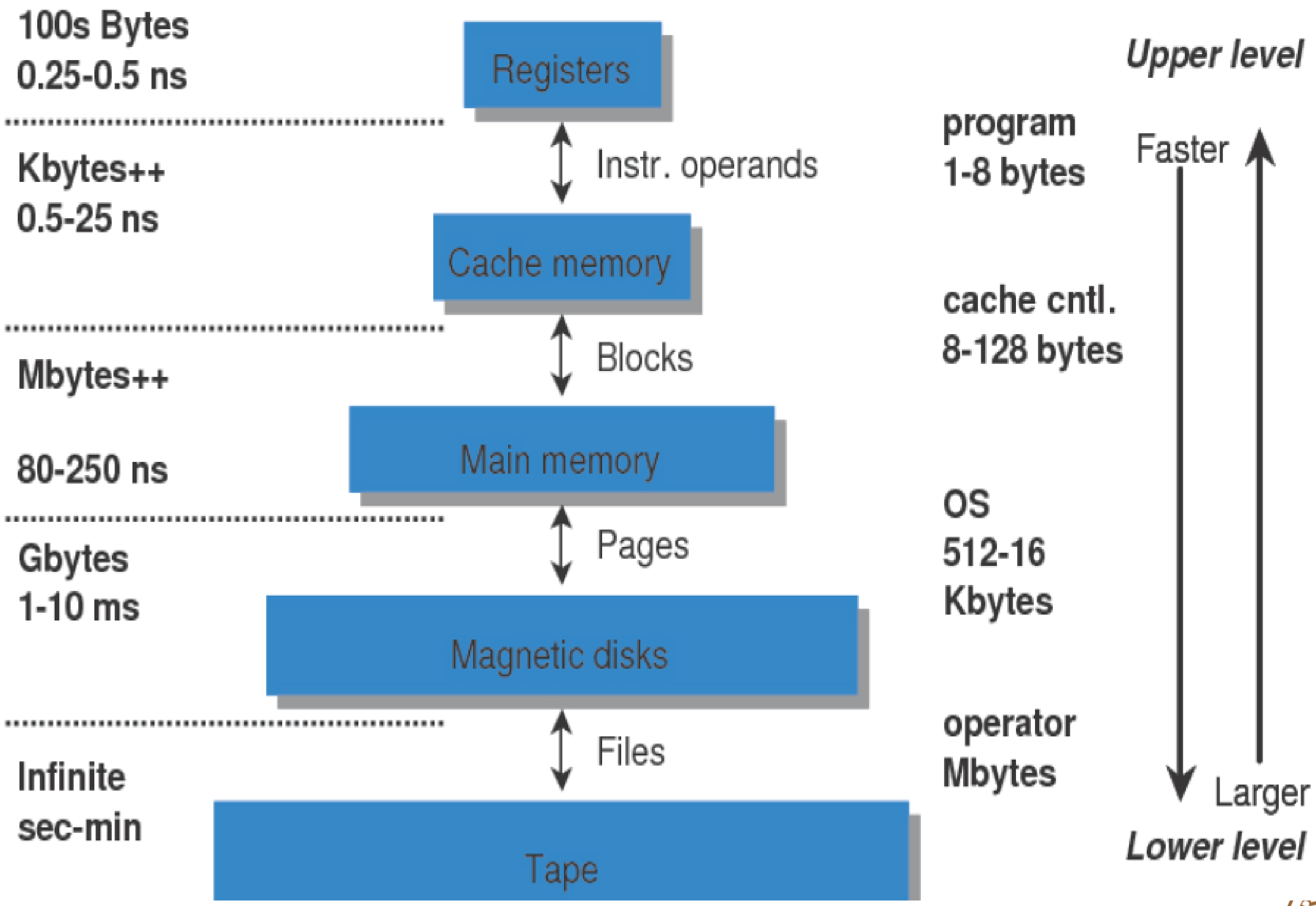
Part 5.1.1: Virtual Memory

Liang Liu
liang.liu@eit.lth.se

Outline

- Reiteration
- Cache optimization
- Virtual memory
- Case study AMD Opteron
- Summary

Memory hierarchy



Cache performance

$$\text{Execution Time} =$$

$$IC * (CPI_{execution} + \frac{\text{mem accesses}}{\text{instruction}} * \textbf{miss rate} * \textbf{miss penalty}) * \mathbf{T_C}$$

Three ways to increase performance:

- Reduce miss rate
- Reduce miss penalty
- Reduce hit time
- ... and increase bandwidth

remember:

Execution time is the only true measure!

Outline

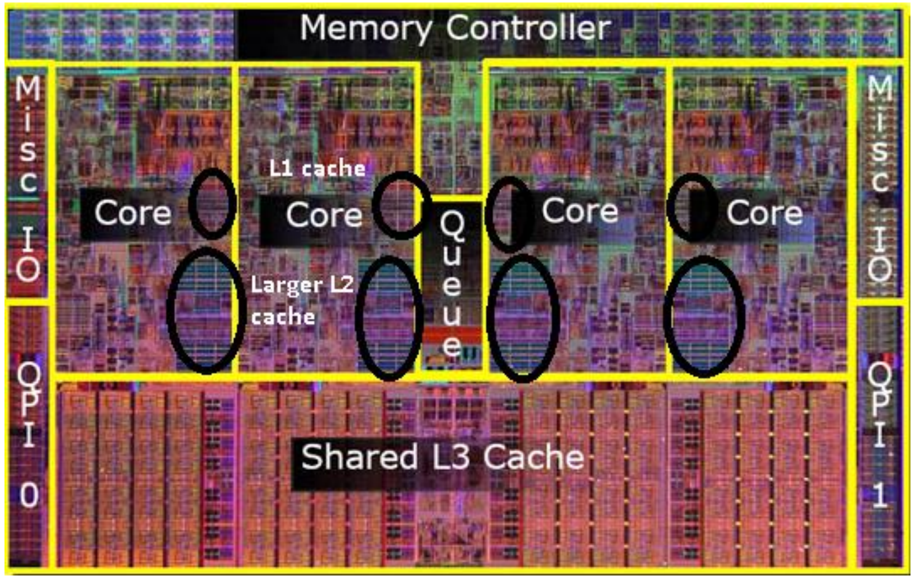
- Reiteration
- Cache performance optimization
- Bandwidth increase
- Reduce hit time
- **Reduce miss penalty**
- Reduce miss rate
- Summary

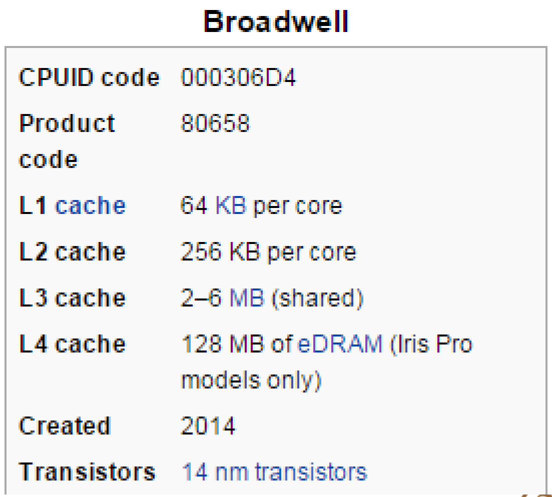
Cache optimizations

	Hit time	Band-width	Miss penalty	Miss rate	HW complexity
Simple	+			-	0
Addr. transl.	+				1
Way-predict	+				1
Trace	+				3
Pipelined	-	+			1
Banked		+			1
Nonblocking		+	+		3
Early start			+		2
Merging write			+		1
Multilevel			+		2
Read priority			+		1
Prefetch			+	+	2-3
Victim			+	+	2
Compiler				+	0
Larger block			-	+	0
Larger cache	-			+	1
Associativity	-			+	1

Reduce miss penalty 1: Multilevel caches

- **Use several levels of cache memory:**
  - The 1st level cache **fast and small** ⇒ match processing speed
  - 2nd level cache can be made much **larger and set-associative** to reduce capacity and conflict misses
  - ... and so on for 3rd and 4th level caches
- **On-chip or Off-chip?**
  - Today 4 levels on-chip





Broadwell

CPUID code	000306D4
Product code	80658
L1 cache	64 KB per core
L2 cache	256 KB per core
L3 cache	2–6 MB (shared)
L4 cache	128 MB of eDRAM (Iris Pro models only)
Created	2014
Transistors	14 nm transistors

Reduce miss penalty 1: Multilevel caches

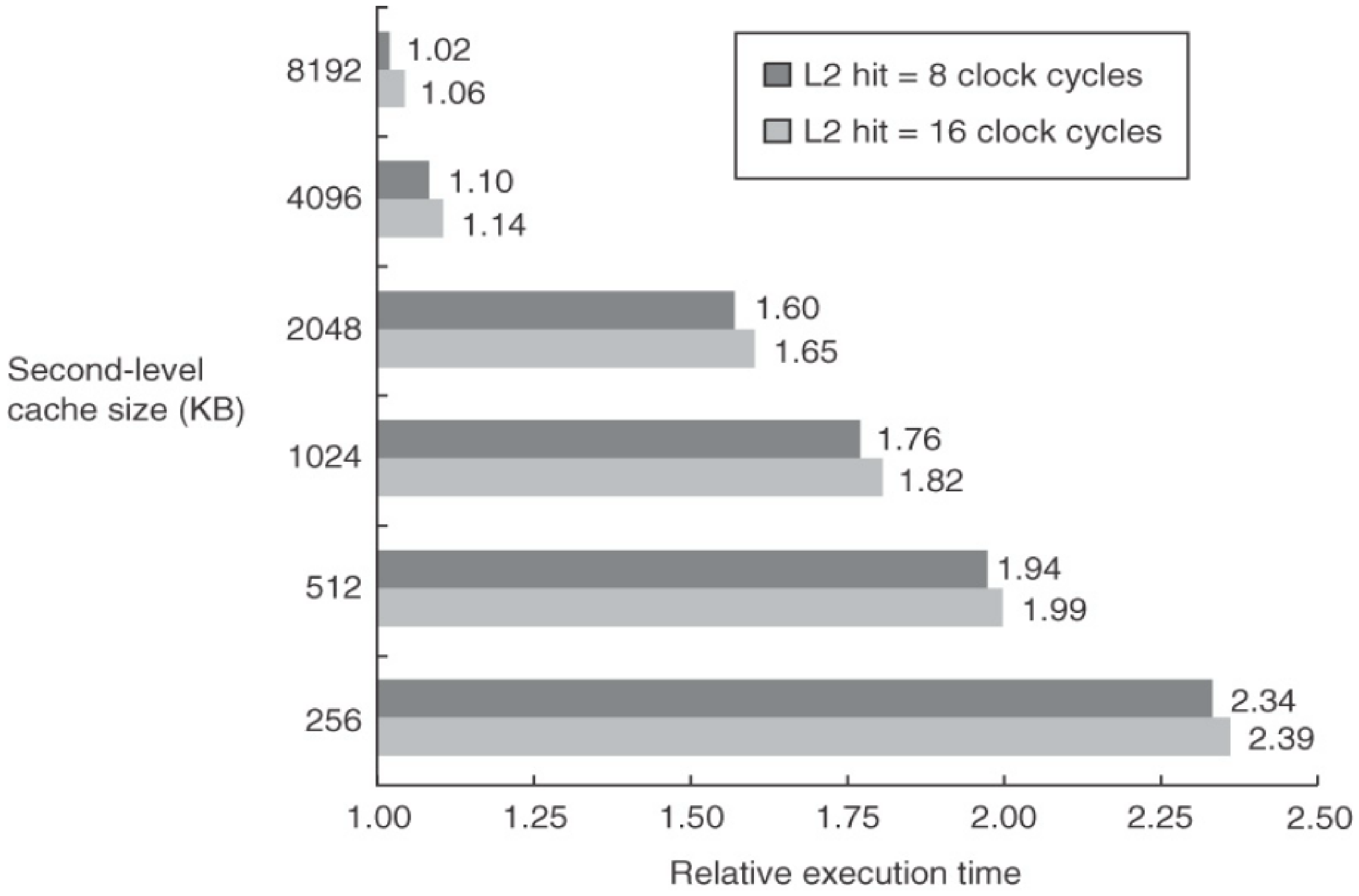
- **Use several levels of cache memory:**
  - The 1st level cache fast and small ⇒ match processing speed
  - 2nd level cache can be made much larger and set-associative to reduce capacity and conflict misses
  - ... and so on for 3rd and 4th level caches
- **On-chip or Off-chip?**
  - Today 4 levels on-chip

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$$

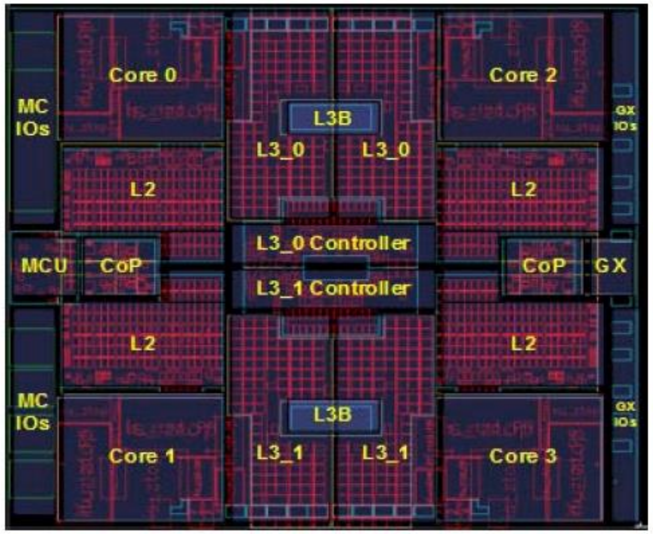
Multilevel caches: execution time



Multilevel caches: examples

		Cache		
CPU	**CP**	**L1**	**L2**	**L3**
	GHz	KB	KB	MB
FX-51	2.2	64+64	1024	-
Itanium 2	1.5	16+16	256	6
Pentium 4	3.2	12+8	512	-
(Pentium 4 EE)	3.2	12+8	512	2
Core i7	3.5	32+32	256	8
Phenom II	3	128	512	8
AMD Bulldozer	4	16+64	2048	8
IBM z196	5.2	64+128	1536	24

IBM z196



zEnterprise 196

Reduce miss penalty 2: Write buffers, Read priority

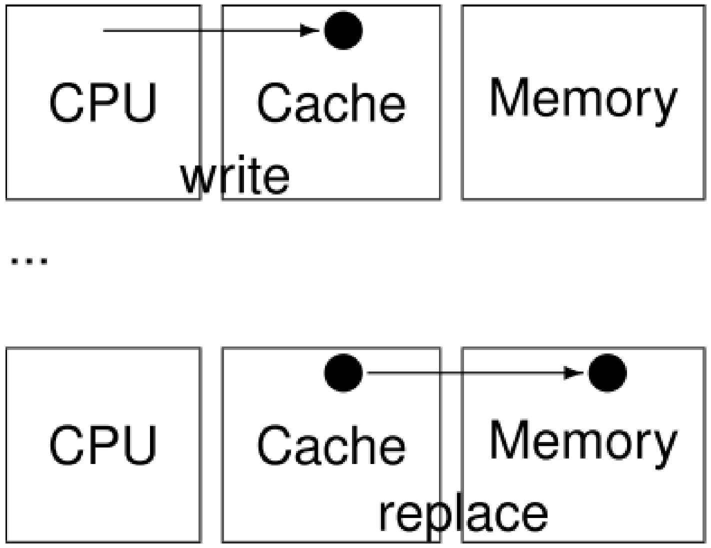
- **Write through:**
  - Using write buffers: RAW conflicts with reads on cache misses (first write is still in the buffer when the LW needs the value)

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

  - If simply wait for write buffer to empty might increase read miss penalty by 50% (old MIPS 1000)
  - Check write buffer contents before read; if no conflicts, let the memory access continue
  - Complicated cache control

Reduce miss penalty 2: Write buffers, Read priority

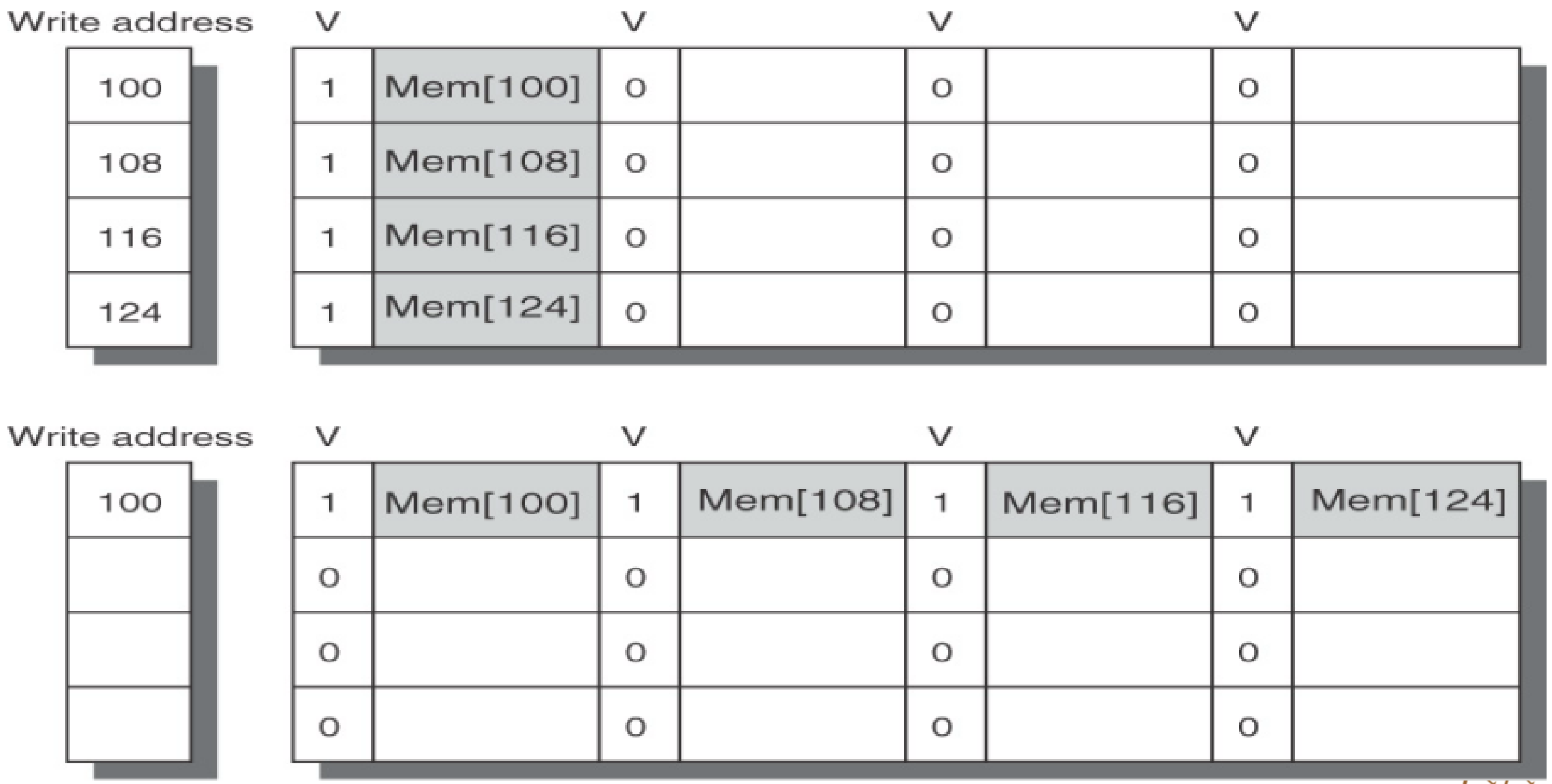
- **Write Back:**
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and then do the read (very long latency and stalls the processor)
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write
  - CPU stall less since restarts as soon as read completes



Reduce miss penalty 2: Write buffers, Read priority

Merging write buffers

- Multi-word writes more efficient to memory
- The Sun T1 (Niagara) processor, among many others, uses write merging



Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

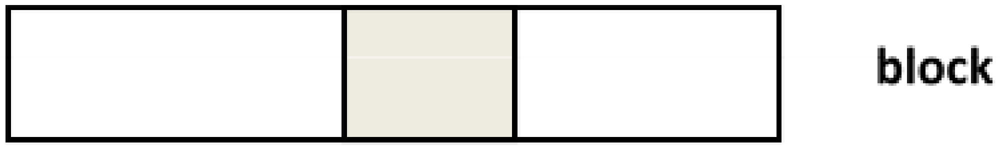
Reduce miss penalty 3: other tricks

Impatience

Don’t wait for full block before restarting CPU

- **Early restart –** fetch words in normal order but restart processor as soon as requested word has arrived
- **Critical word first –** fetch the requested word first. Overlap CPU execution with filling the rest of the cache block

Increases performance mainly with large block sizes.



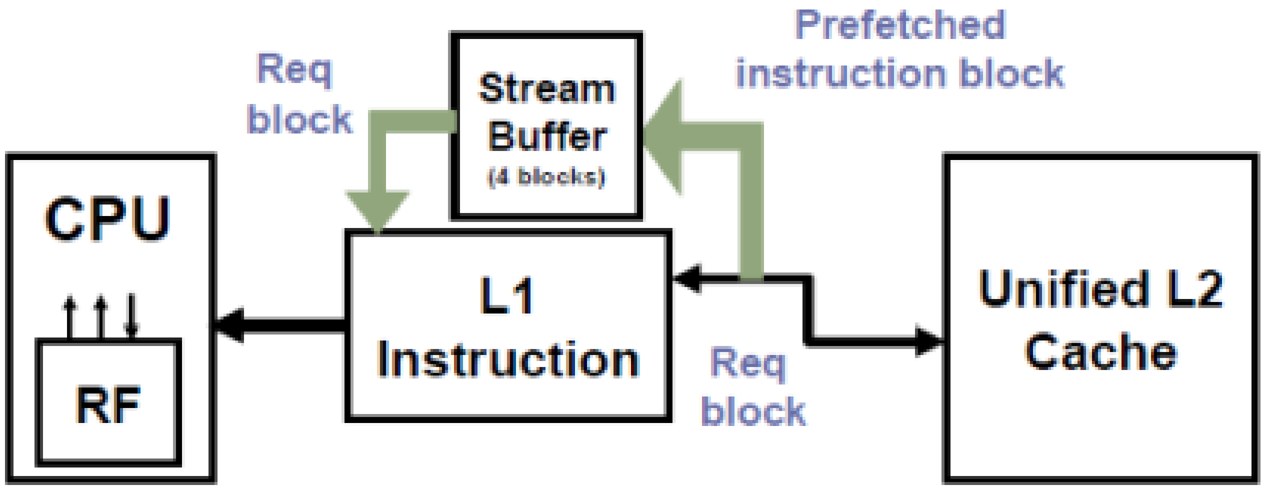
Reduce miss penalty 4: Non-blocking caches

- **Non-blocking cache ≡ lockup-free cache**
  - (+) Permit other cache operations to proceed when a miss has occurred
  - (+) May further lower the effective miss penalty if multiple misses can overlap
  - (-) The cache has to book-keep all outstanding references –Increases cache controller complexity
- **Good for out-of-order pipelined CPUs**
  - The presence of true data dependencies may limit performance
  - Requires pipelined or banked memory system (otherwise cannot support)

Reduce miss rate/penalty: hardware prefetching

Goal: overlap execution with speculative prefetching to cache

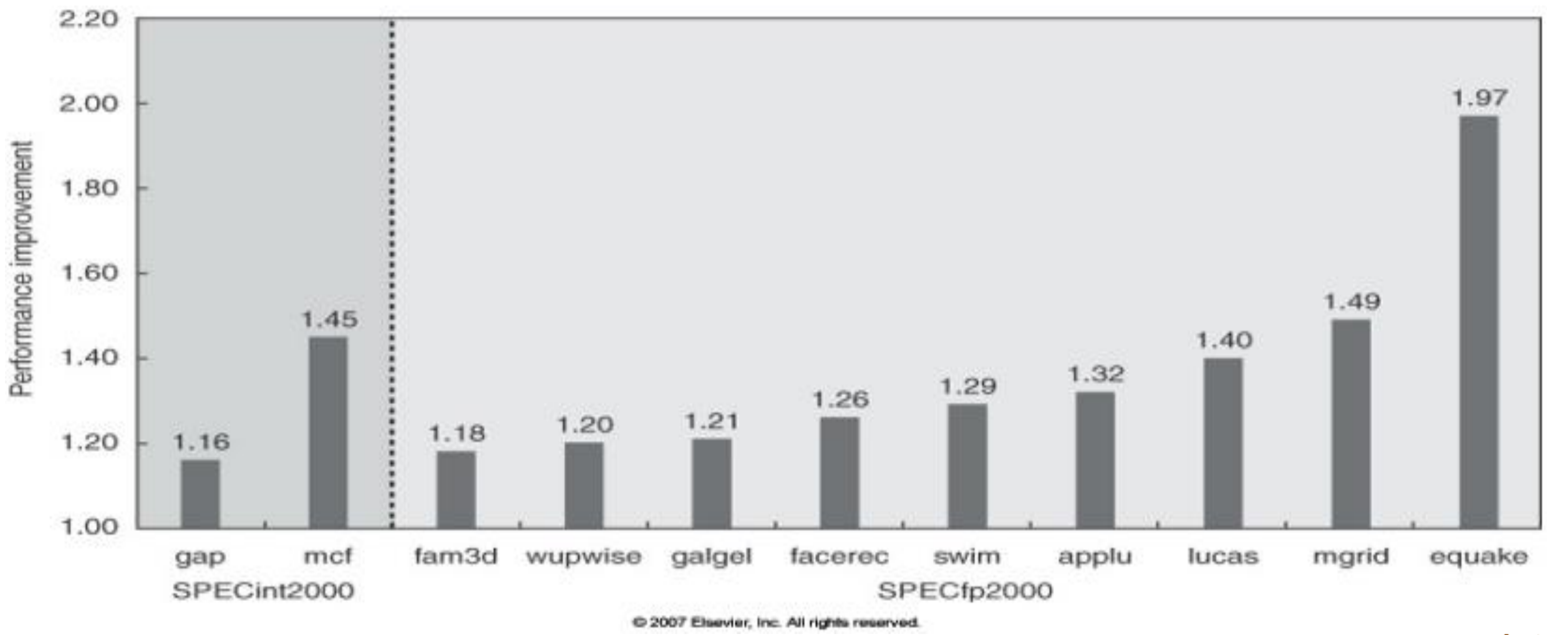
- **Hardware prefetching -** If there is a miss for block X, fetch also block X+1, X+2,... X+d
  - Instruction prefetching
    - **Alpha 21064 fetches 2 blocks on a miss (Intel i7 on L1 and L2)**
    - **Extra block placed in stream buffer or caches**
    - **On miss check stream buffer (highly possible is there)**
  - Works with data blocks too (generally better with I-Cache but depending on application)



Reduce miss rate/penalty: hardware prefetching

Goal: overlap execution with speculative prefetching to cache

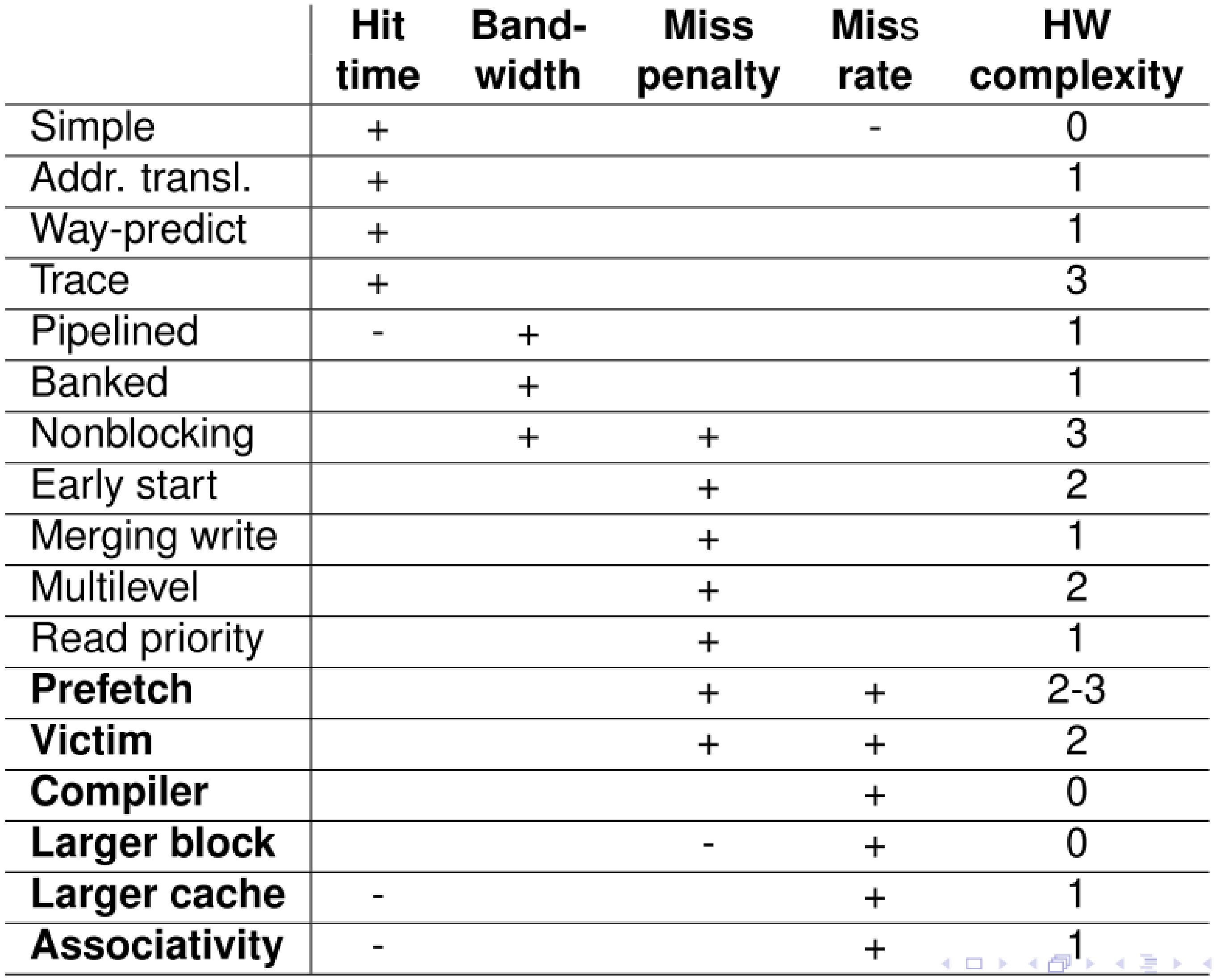
- **Potential issue**
  - Complicated cache control
  - Relies on **extra memory bandwidth** that can be used without penalty
  - Only useful if produce hit for next reference
  - May polute cache (useful data is replaced)



Outline

- Reiteration
- Cache performance optimization
- Bandwidth increase
- Reduce hit time
- Reduce miss penalty
- **Reduce miss rate**
- Summary

Cache optimizations



	Hit time	Band-width	Miss penalty	Miss rate	HW complexity
Simple	+			-	0
Addr. transl.	+				1
Way-predict	+				1
Trace	+				3
Pipelined	-	+			1
Banked		+			1
Nonblocking		+	+		3
Early start			+		2
Merging write			+		1
Multilevel			+		2
Read priority			+		1
Prefetch			+	+	2-3
Victim			+	+	2
Compiler				+	0
Larger block			-	+	0
Larger cache	-			+	1
Associativity	-			+	1

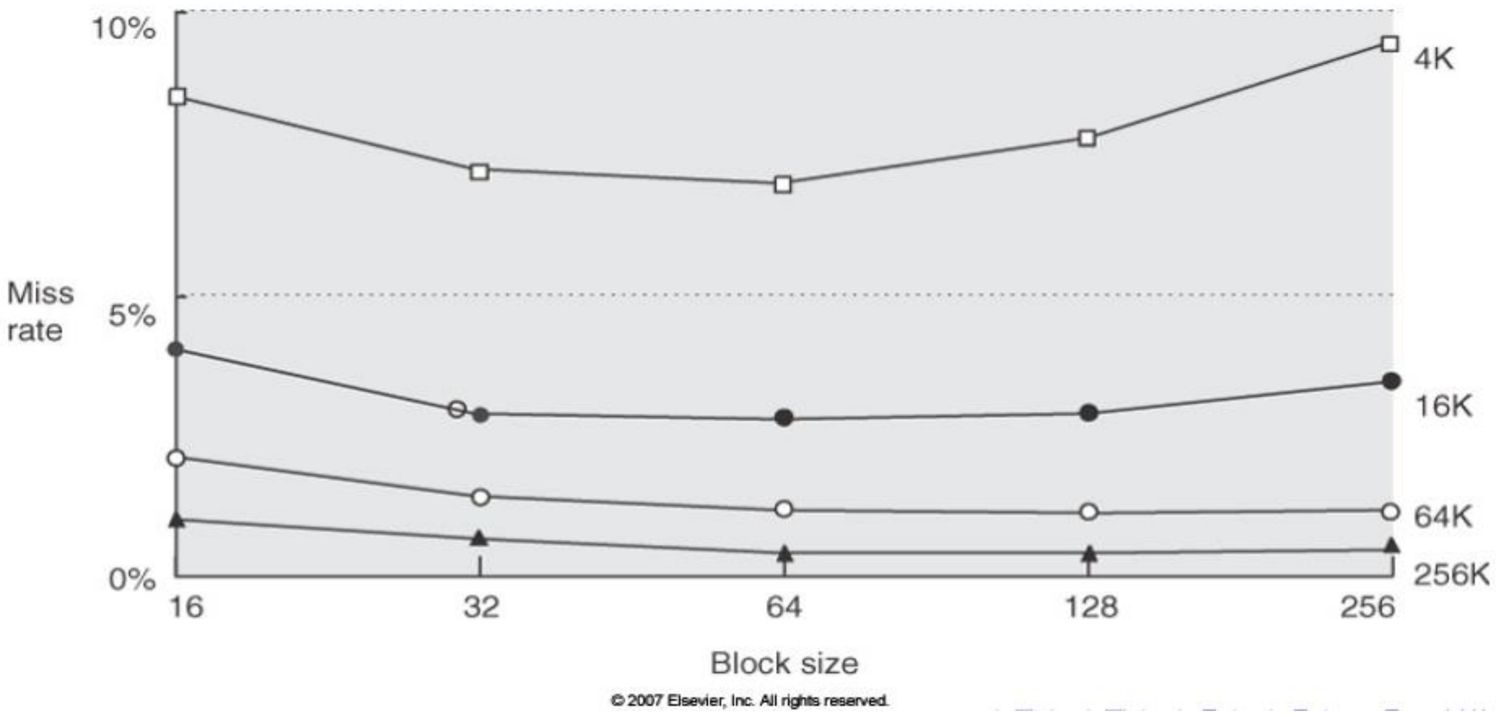
Reduce miss rate

- **The three C's:**
  - Compulsory – misses in an infinite cache
  - Capacity – misses in a fully associative cache
  - Conflict – misses in an N-way associative cache
- **How do we reduce the number of misses?**
  - Change cache size?
  - Change block size?
  - Change associativity?
  - Change compiler?
  - Other tricks!

Which of the three C's are affected?

Reduce misses 1: increase block size

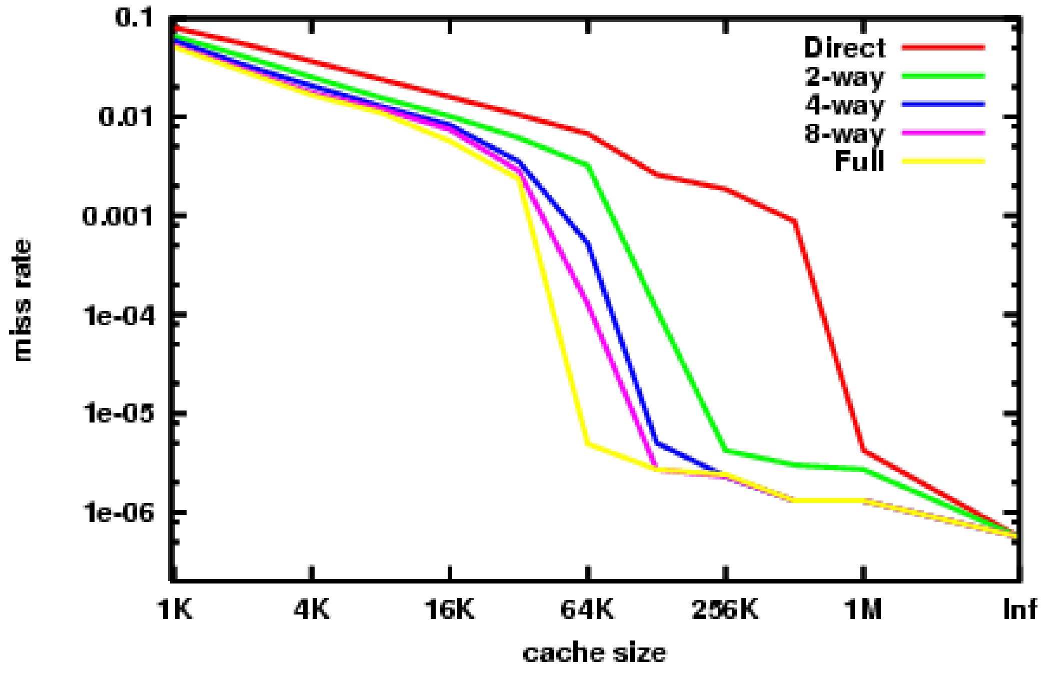
- Increased block size utilizes the spatial locality
- Too big blocks increases miss rate
- Big blocks also increases miss penalty



Beware - impact on average memory access time

Reduce misses 2: change associativity

Rule of thumb: A direct mapped cache of size N has the same miss rate as a 2-way set associative cache of size N/2



- **Hit time increases with increasing associativity**

Beware - impact on average memory access time

Reduce misses 3: Compiler optimizations

Basic idea: Reorganize code to improve locality

- **Merging Arrays**
  - Improve spatial locality by single array of compound elements vs. 2 arrays
- **Loop Interchange**
  - Change nesting of loops to access data in order stored in memory
- **Loop Fusion**
  - Combine two independent loops that have same looping and some variables overlap
- **Blocking**
  - Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

Reduce misses 3: Compiler optimizations

Loop Interchange

- Change nesting of loops to access data in order stored in memory
- If x[i][j] and x[i][j+1] are adjacent (**row major**)

```
/* Before */
for (k = 0; k < 100; k++)
  for (j = 0; j < 100; j++)
    for (i = 0; i < 5000; i++)
      x[i][j] = 2 * x[i][j];
```

```
/* After */
for (k = 0; k < 100; k++)
  for (i = 0; i < 5000; i++)
    for (j = 0; j < 100; j++)
      x[i][j] = 2 * x[i][j];
```

Depending on the storage of the matrix
Sequential accesses instead of striding through memory every 100 words

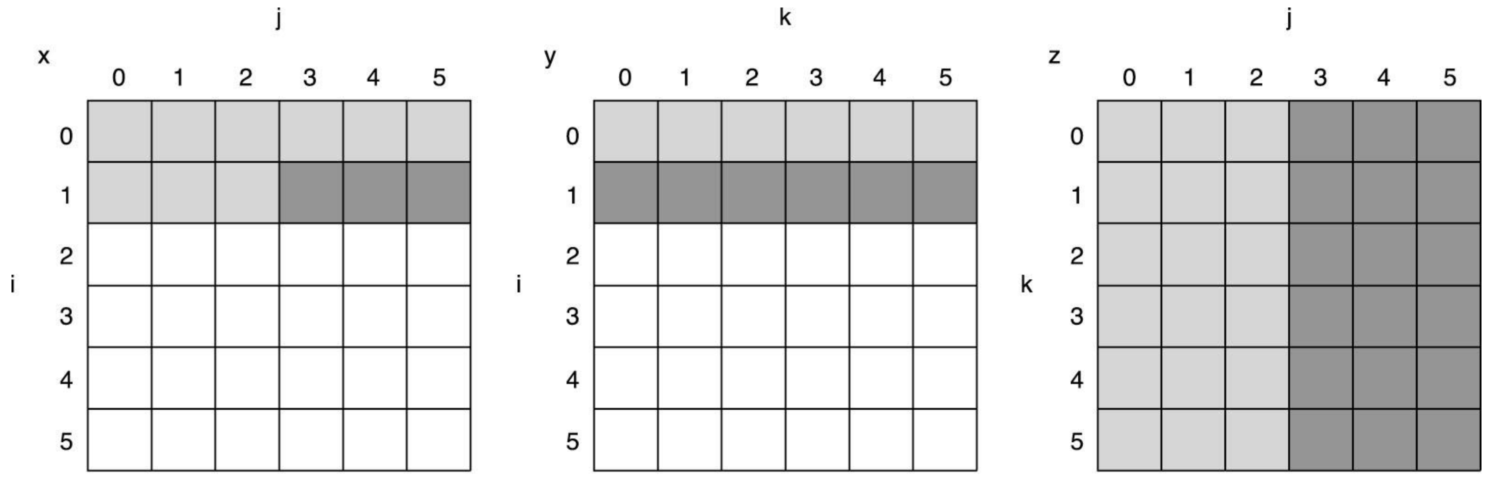
Reduce misses 3: Compiler optimizations

- Block (matrix multiplication)

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

Reduce misses 3: Compiler optimizations

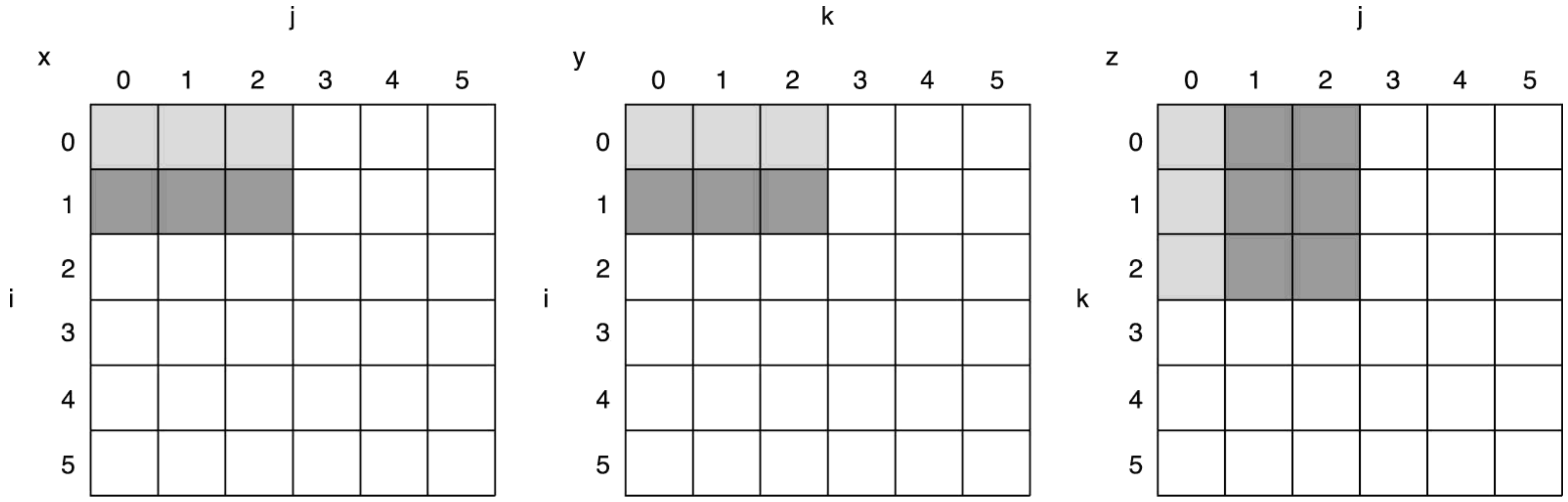
- White means not touched yet
- Light gray means touched a while ago
- Dark gray means newer accesses



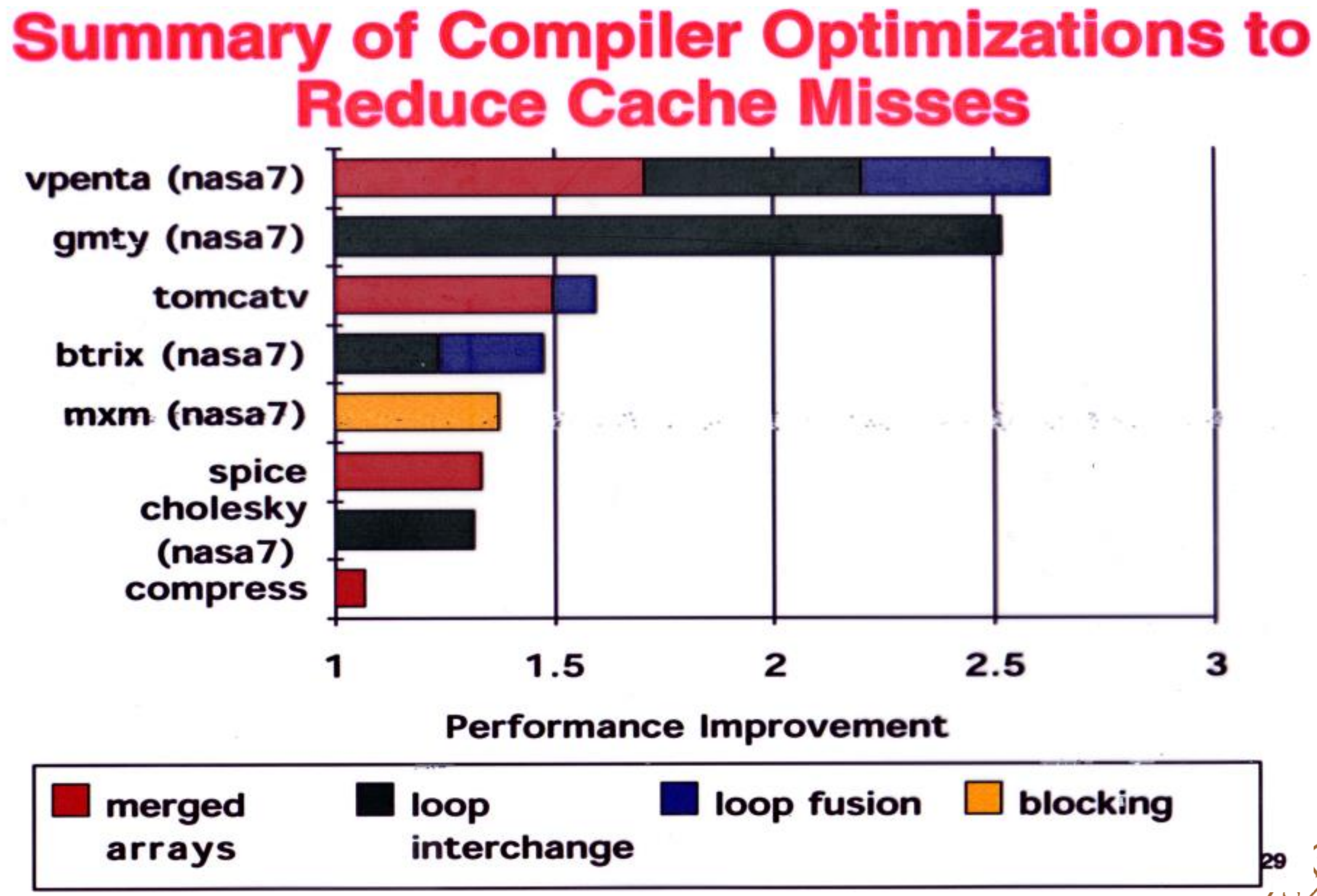
Reduce misses 3: Compiler optimizations

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i++)
      for (j = jj; j < min(jj+B-1,N); j++) {
         r = 0;
         for (k = kk; k < min(kk+B-1,N); k++)
           r = r + y[i][k]*z[k][j];
         x[i][j] = x[i][j] + r;
      }
```

Reduce misses 3: Compiler optimizations

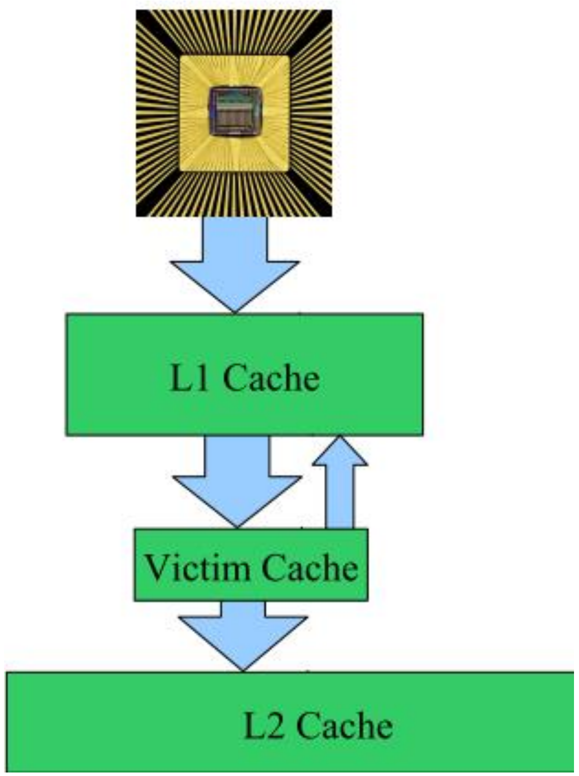


Reduce misses 3: Compiler optimizations



Reduce misses 4: Victim cache

How to combine fast hit time of direct mapped yet still avoid conflict misses?



- **Victim cache operation**
  - On a miss in L1, we check the Victim Cache
  - If the block is there, then bring it into L1 and swap the ejected value into the victim cache
  - If not, fetch the block from the lower levels
- **Norman Jouppi,1990**
  - a 4-entry victim cache removed 25% of conflict misses for a 4 Kbyte direct mapped cache
- **Used in AMD Athlon, HP and Alpha machines**

Outline

- Reiteration
- Cache performance optimization
- Bandwidth increase
- Reduce hit time
- Reduce miss penalty
- Reduce miss rate
- **Summary**

Cache performance

$$\text{Execution Time} =$$

$$IC * (CPI_{execution} + \frac{\text{mem accesses}}{\text{instruction}} * \textbf{miss rate} * \textbf{miss penalty}) * \mathbf{T_C}$$

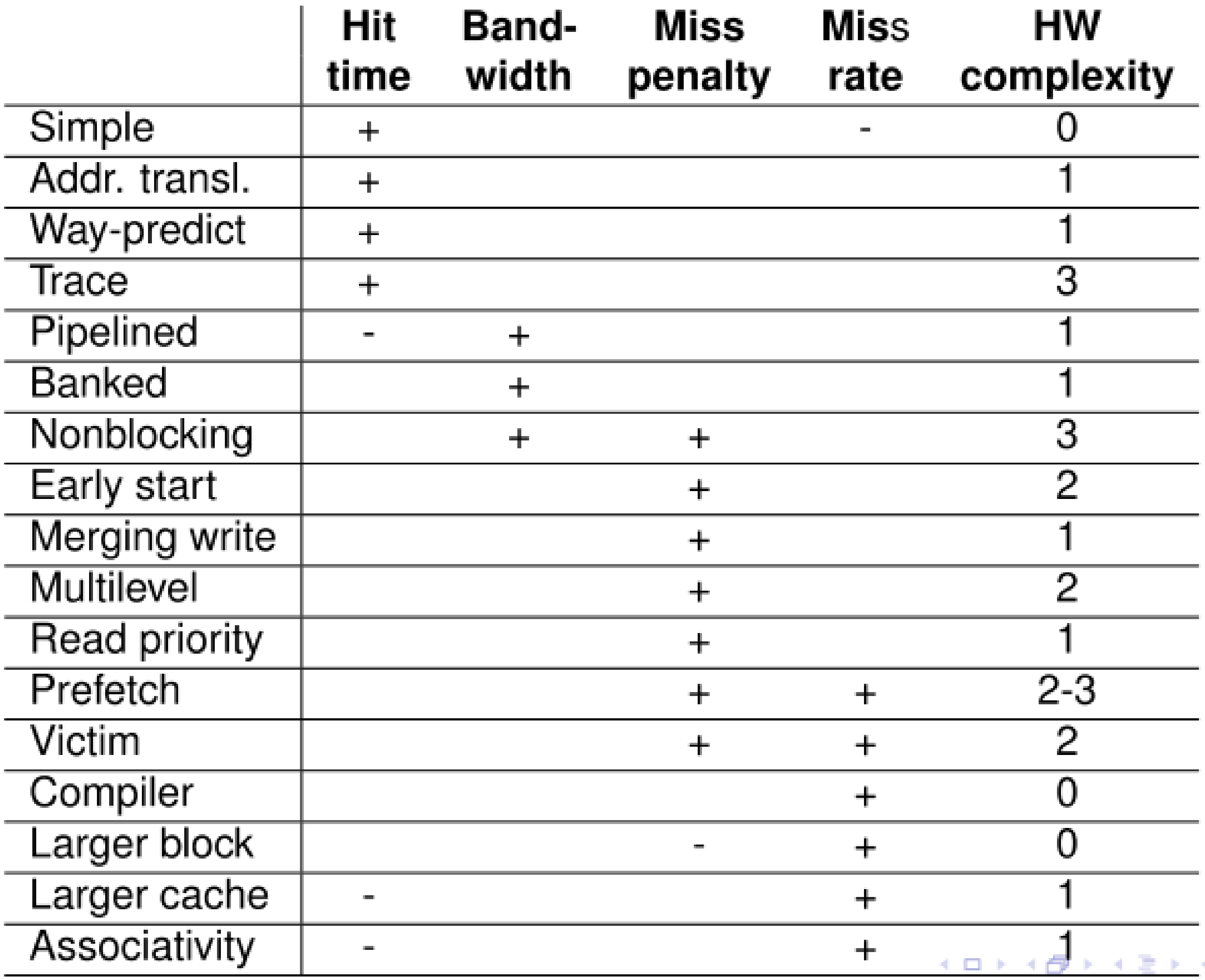
Three ways to increase performance:

- Reduce miss rate
- Reduce miss penalty
- Reduce hit time
- ... and increase bandwidth

remember:

Execution time is the only true measure!

Cache optimization

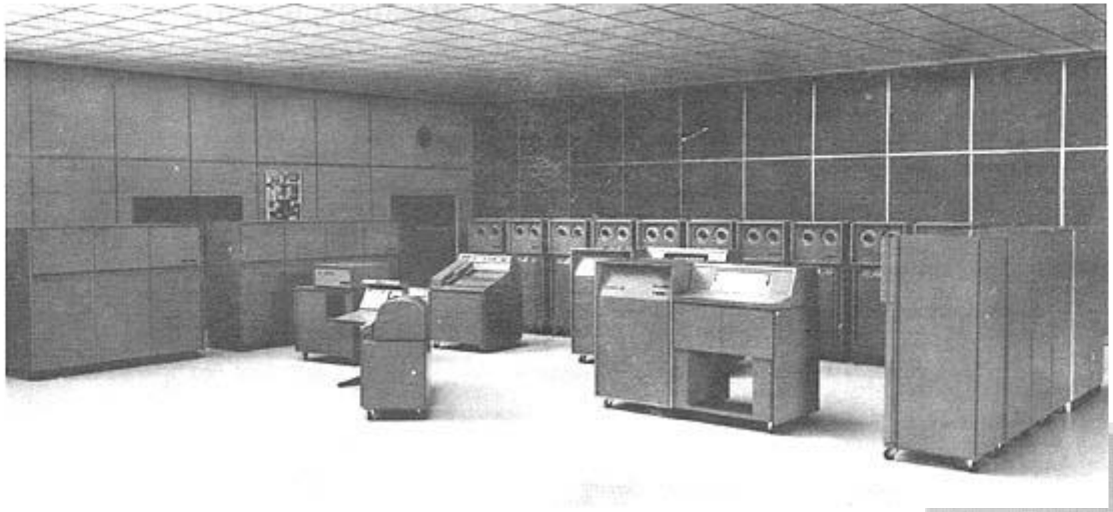


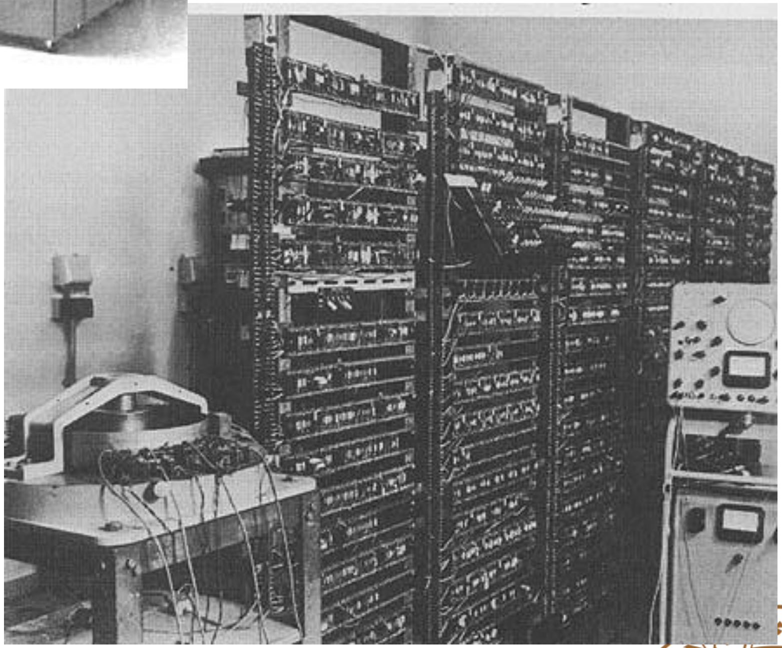
	Hit time	Band-width	Miss penalty	Miss rate	HW complexity
Simple	+			-	0
Addr. transl.	+				1
Way-predict	+				1
Trace	+				3
Pipelined	-	+			1
Banked		+			1
Nonblocking		+	+		3
Early start			+		2
Merging write			+		1
Multilevel			+		2
Read priority			+		1
Prefetch			+	+	2-3
Victim			+	+	2
Compiler				+	0
Larger block			-	+	0
Larger cache	-			+	1
Associativity	-			+	1

Outline

- Reiteration
- Virtual memory
- Case study AMD Opteron
- Summary

Virtual memory





Above: The Burrough B5000 computer. The first commercial machine with virtual memory (1961).

Right: First experimental virtual memory. The Manchester Atlas computer, which had virtual memory backed on a magnetic drum.

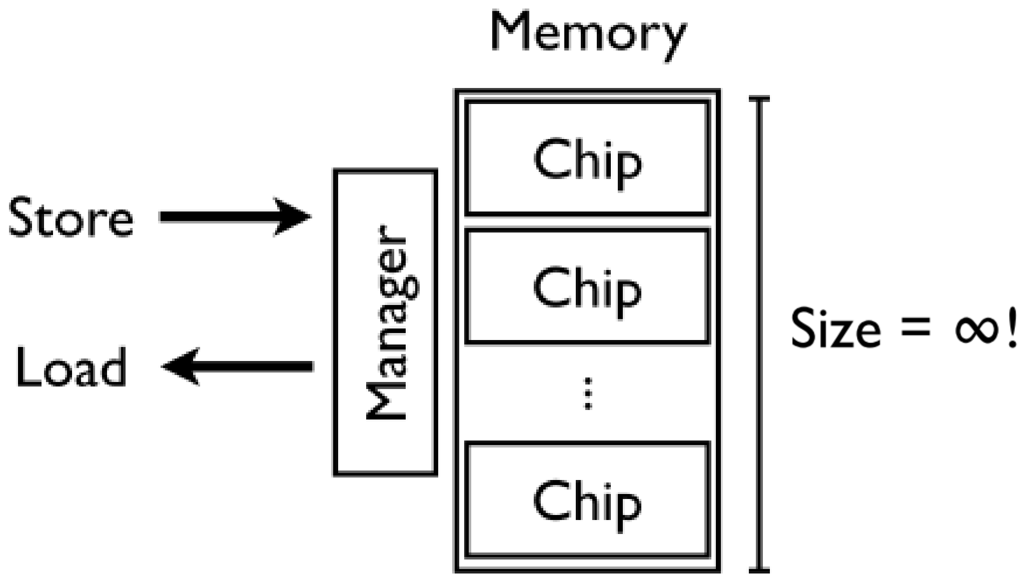
Memory hierarchy tricks

Use two "magic" tricks

- Make a slow memory seem faster
  (Without making it smaller)

cache memory

Memory tricks (techniques)

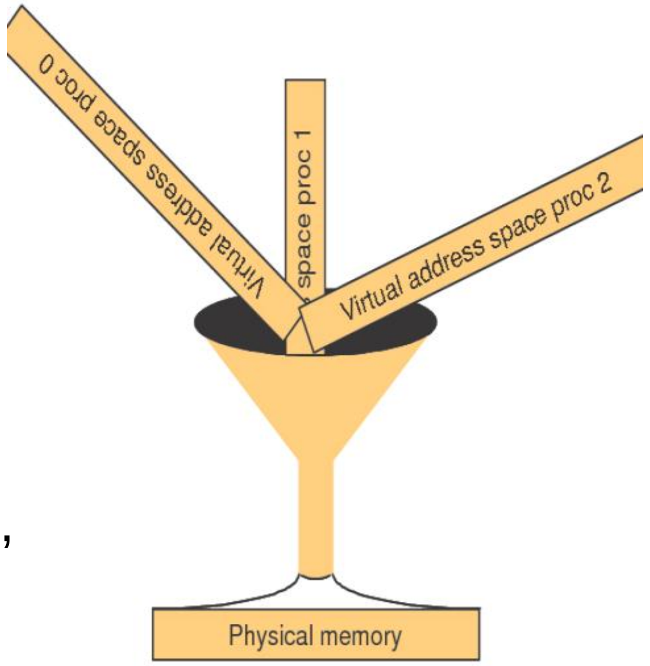


“An engineer is a man who can do for a **dime** what any other may do for a **dollar**”
— Anonymous

“An engineer is a man who can do for **16G** what any other may do for **infinite**.”
— Anonymous

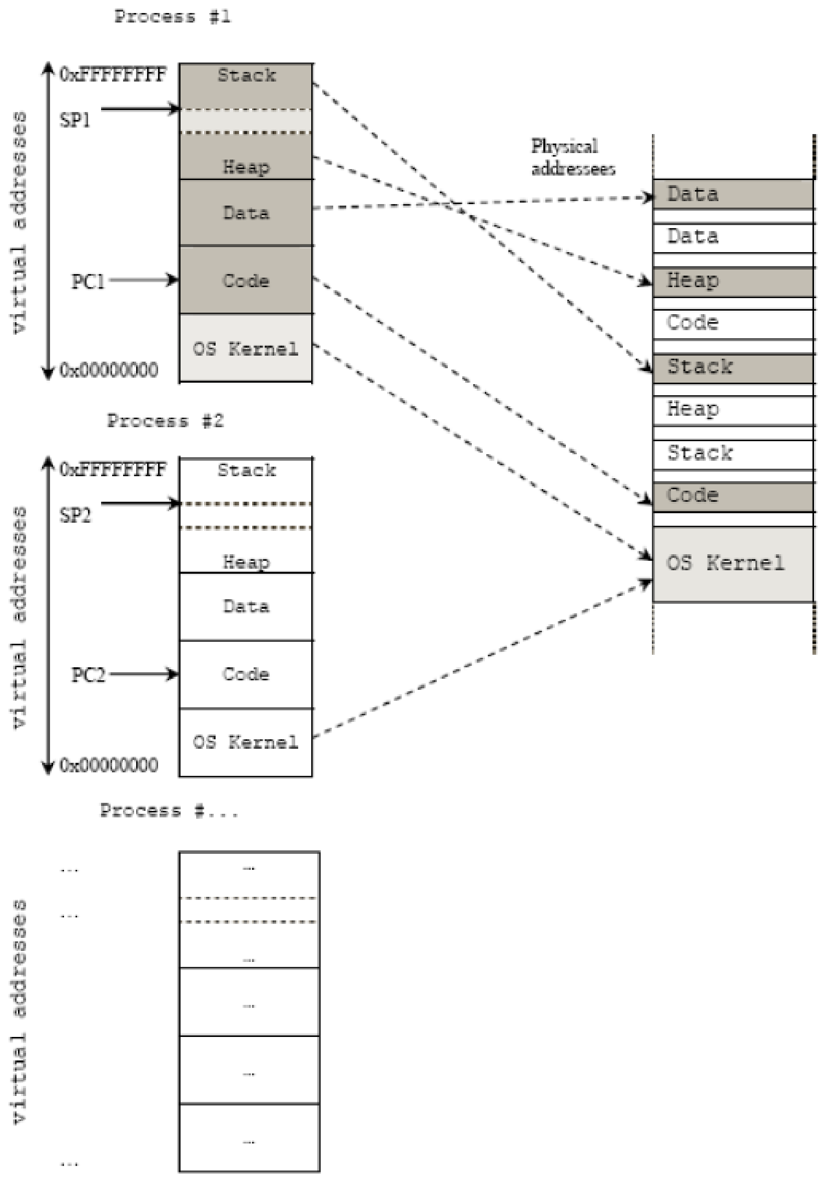
OS Processes/Virtual memory

- Run several programs at the same time
- Each having the full address space available (but only use part of it)
- Sharing physical memory among processes
- Program uses virtual memory address
- Virtual address is translated to physical address
- Should be completely transparent to program, minimal performance impact


Virtual address space proc 0
Virtual address space proc 1
Virtual address space proc 2
Physical memory

Was invented to solve the overlays problem

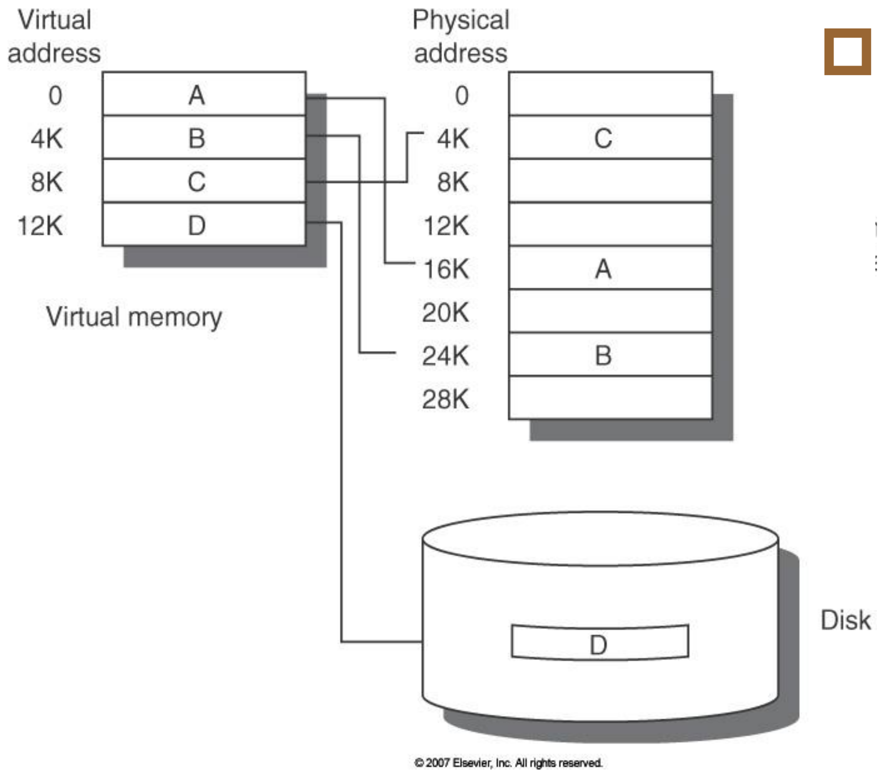
Process address spaces



Virtual memory benifits

- **Using physical memory efficiently**
  - Allowing software to address more than physical memory
  - Enables programs to begin before loading fully (some implementations)
  - Programmers used to use overlays and manually control loading/unloading (if the program size is larger than mem size)
- **Using physical memory simply**
  - Virtual memory simplifies memory management
  - Programmer can think in terms of a large, linear address space
- **Using physical memory safely**
  - Virtual memory protests process' address spaces
  - Processes cannot interfere with each other, because they operate in different address space (or limited mem space)
  - User processes cannot access priviledged information

Virtual memory concept

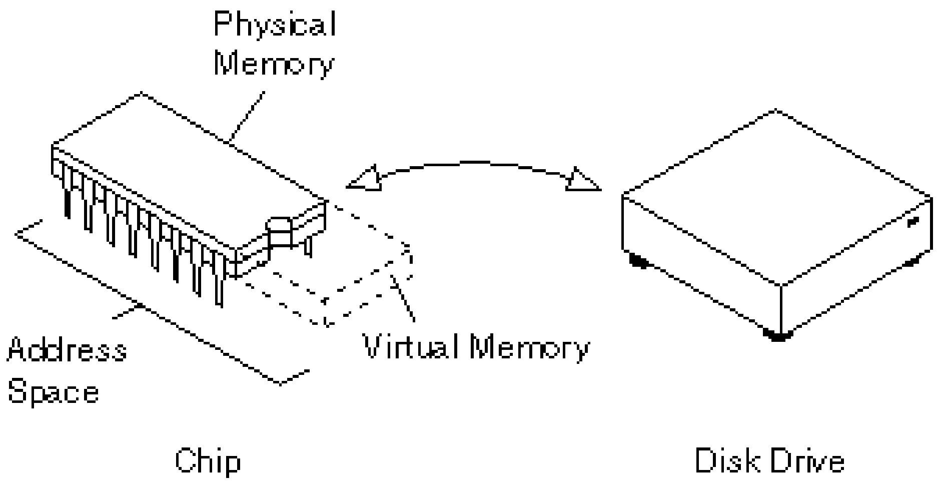


- **Is part of memory hierarchy**
  - The virtual address space is divided into **pages** (blocks in Cache)
  - The physical address space is divided into **page frames**
  - A miss is called a **page fault**
  - Pages not in main memory are stored on **disk**

- **The CPU uses *virtual addresses***
- **We need an *address translation* (memory mapping) mechanism**

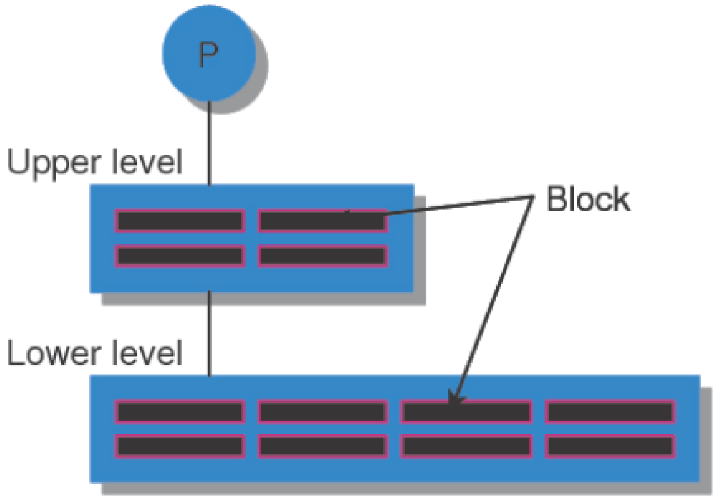
“Virtual”

- **Why “virtual”?**
  - If you think it’s there, and it’s there... it’s real
  - If you think it’s not there, and it’s not there... it’s non-existent
  - If you think it’s not there, and it’s there... it’s transparent
  - If you think it’s there, and it’s not there... it’s imaginary
- **Virtual memory is imaginary memory**
  - It gives you the illusion of memory that’s not physically there

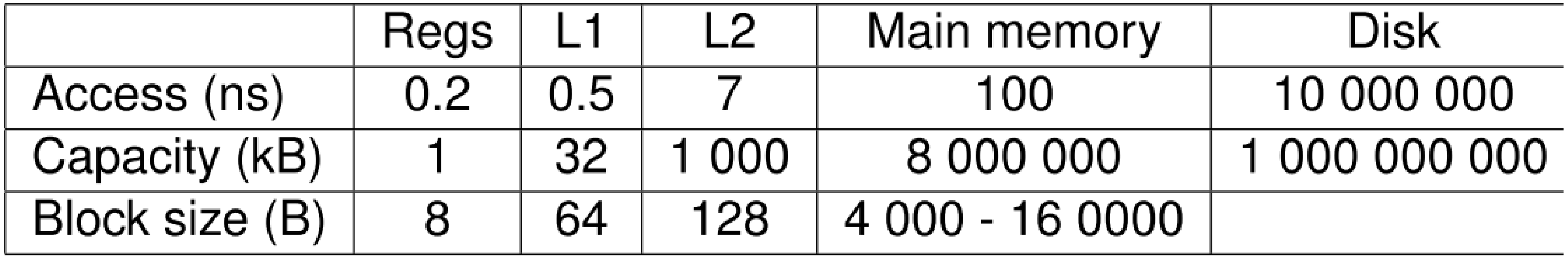


4 memory hierarchy questions

- Q1: Where can a block be placed in the upper level? (Block placement)
- Q2: How is a block found if it is in the upper level? (Block identification)
- Q3: Which block should be replaced on a miss? (Block replacement)
- Q4: What happens on a write? (Write strategy)



Virtual memory parameters



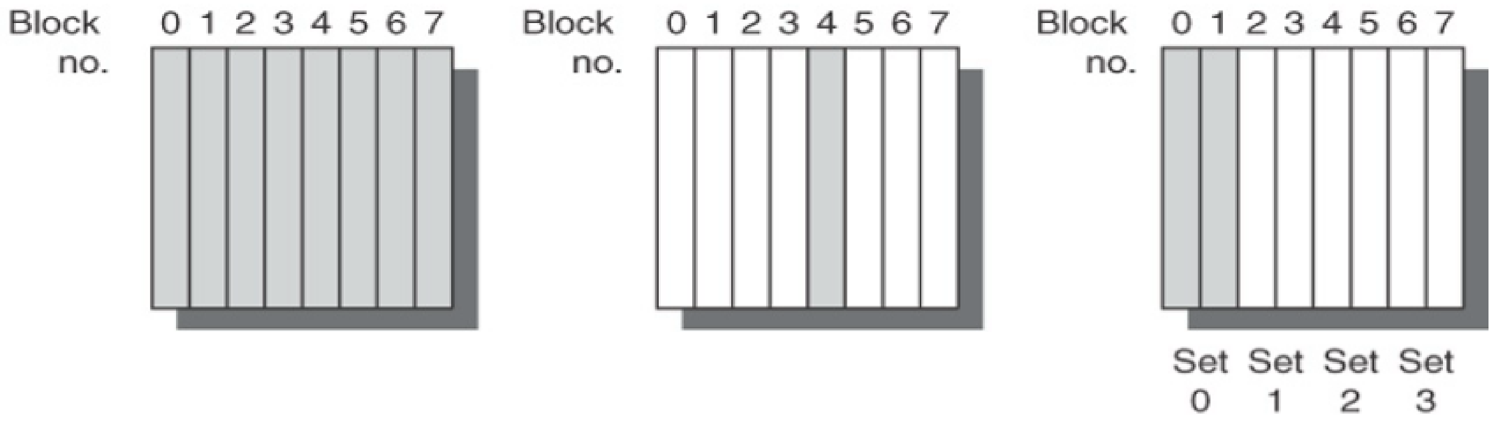
	Regs	L1	L2	Main memory	Disk
Access (ns)	0.2	0.5	7	100	10 000 000
Capacity (kB)	1	32	1 000	8 000 000	1 000 000 000
Block size (B)	8	64	128	4 000 - 16 0000	

- **Size of VM determined by no of address bits**
  - 32-bits: ~4,000,000,000 (four billion) bytes (4GB)
  - 64-bits: ~16,000,000,000,000,000,000 (sixteen quintillion) bytes

Page placement

- **Where can a page be placed in main memory?**
  - Cache access: ~ ns
  - Memory access: ~ 100 ns
  - Disk access: ~ 10, 000, 000 ns

$\Longrightarrow$ **HIGH miss penalty**



Page placement

- **Where can a page be placed in main memory?**
  - Cache access: ~ ns
  - Memory access: ~ 100 ns
  - Disk access: ~ 10, 000, 000 ns

$\Longrightarrow$ **HIGH miss penalty**

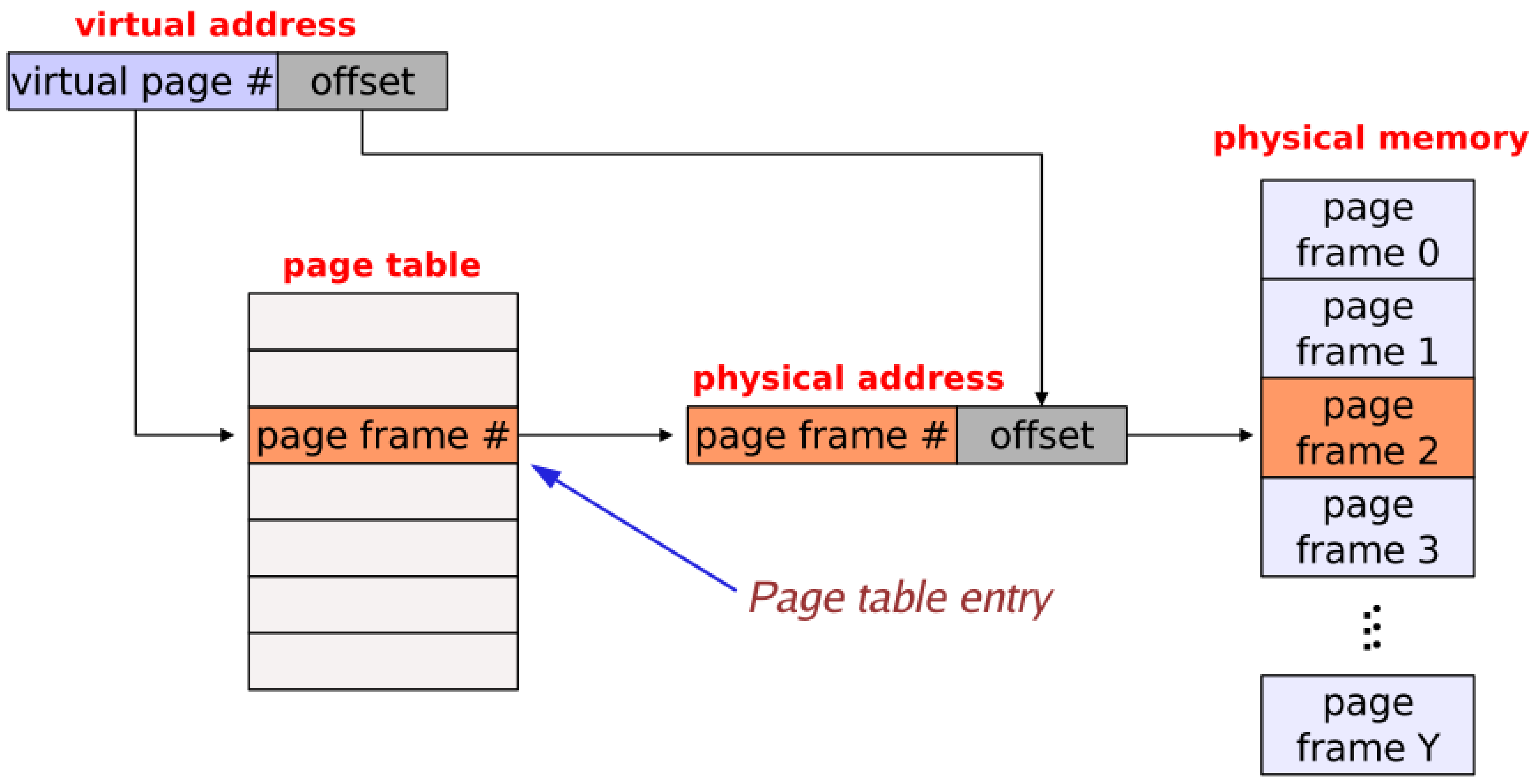
- **The high miss penalty makes it**
  - Necessary to **minimize miss rate**
  - Possible to use software solutions to implement a **fully associative address** mapping

Page identification

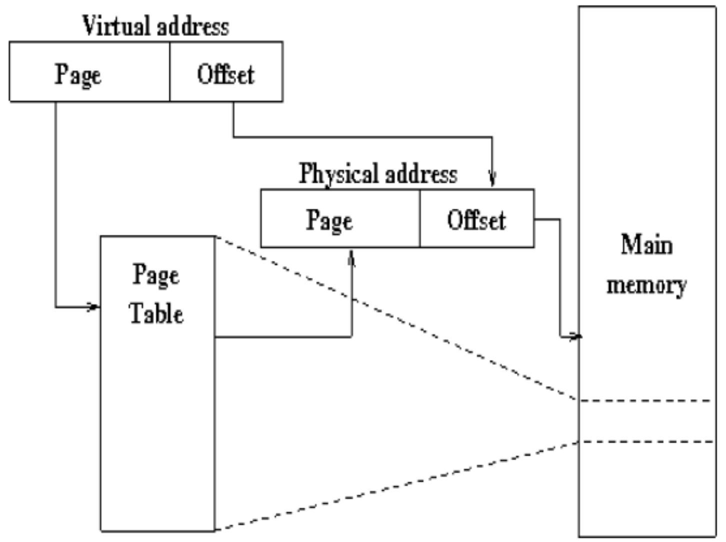
- **Assume**
  - 4GB VM composed of 2^{20} 4KB pages
  - 64MB DRAM main memory composed of 16384 (2^{14}) page frames (of same size)
- **Only those pages (of the 2^{20}) that are not empty actually exist**
  - Each is either in main memory or on disk
  - Can be located with **two** mappings (implemented with tables)

Virtual address	**= (virtual page number,**	**page offset)**
VA	**= (VPN,**	**offset)**
32 bits	**= (20 bits +**	**12 bits)**
Physical address	**= (real page number,**	**page offset)**
PA	**= (RPN,**	**offset)**
26 bits	**= (14 bits +**	**12 bits)**

Page identification



Page identification: address mapping



- **Contains Real Page Number**
- **Miscellaneous control information**
  - valid bit,
  - dirty bit,
  - replacement information,
  - access control

- **4Byte per page table entry**
  - Page table will have

    $2^{20}*4=2^{22}=4$MByte
  - Generally stored in the main memory

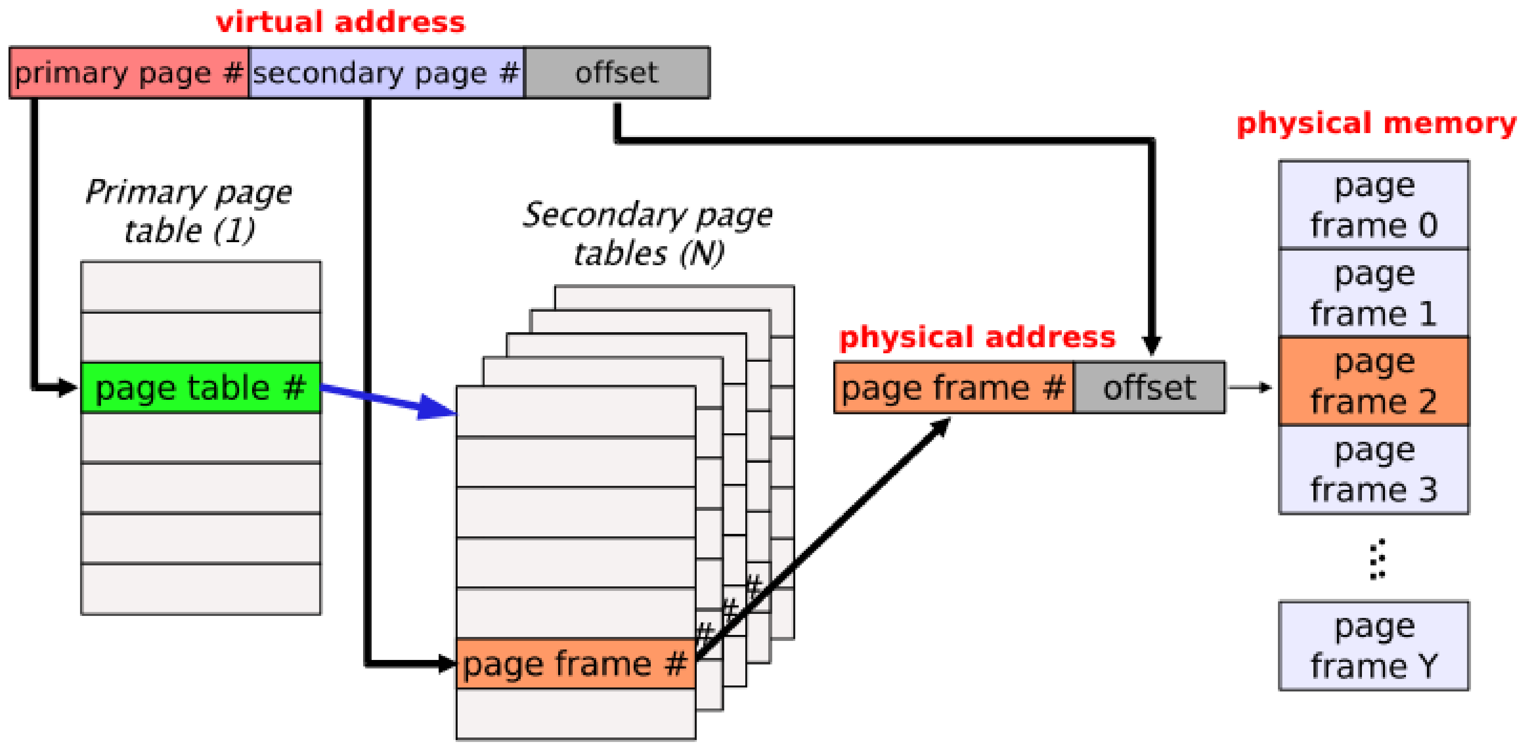
- **64 bit virtual address,16 KB pages:**

  $2^{64}/2^{14}*4=2^{52}=2^{12}$TByte

- **One page table per program (100 program?)**
- **Solutions**
  - Multi–level page table
  - Inverted page table

Multi-level PT

- **Problem:**
  - Can't hold all of the page tables in memory
  - 1-Level Page Table can only be stored in memory (PA is needed)
- **Solution: Page the page tables!**
  - Allow portions of the page tables to be kept in memory at one time



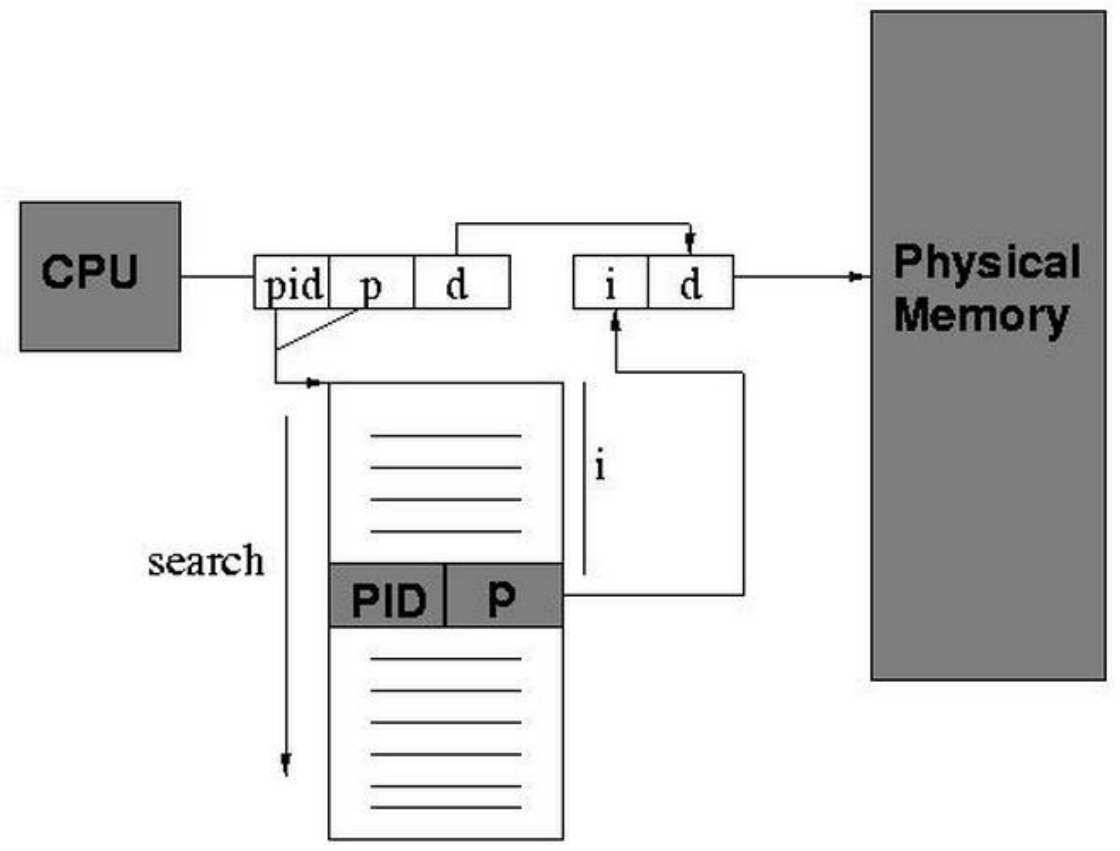
Multi-level PT

- **With two levels of page tables, how big is each table?**
  - We allocate 10 bits to the primary page, 10 bits to the secondary page, 12 bits to the page offset
  - Primary page table is then 2^10 * 4 Bytes per PTE = 4 KB
  - 1 secondary page table is also 4 KB
  - That's exactly the size of a page on most systems ...
- **Issues**
  - Page translation has very high overhead (may have page fault for the 2nd level PT)
  - Up to three memory accesses plus potential disk I/Os!!

Inveted page table

Concept

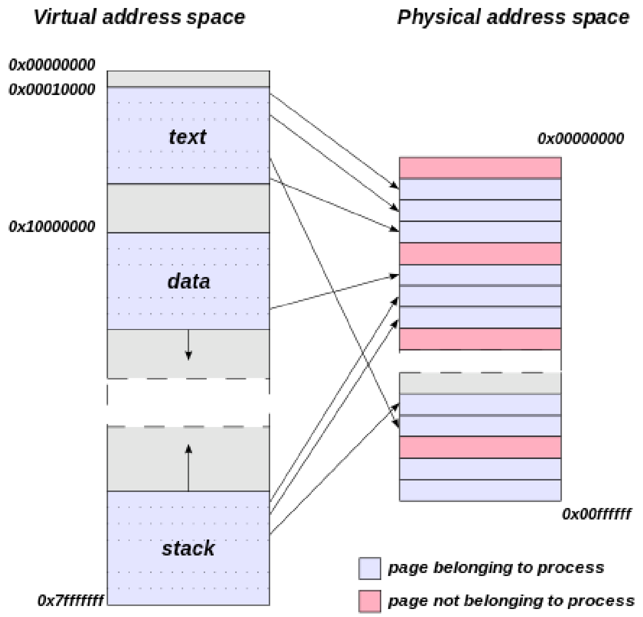
- Contains an entry for each **physical** page, not for each **logical** page.
- The size is proportional to physical memory, not the virtual address space



Virtual memory access

Access steps

- CPU issues a load for virtual address
- Split into page number, page offset
- Look-up in the page table (main memory) to translate page
- Concatenate translated page with offset → physical address
- A read is done from the main memory at physical address
- Data is delivered to the CPU

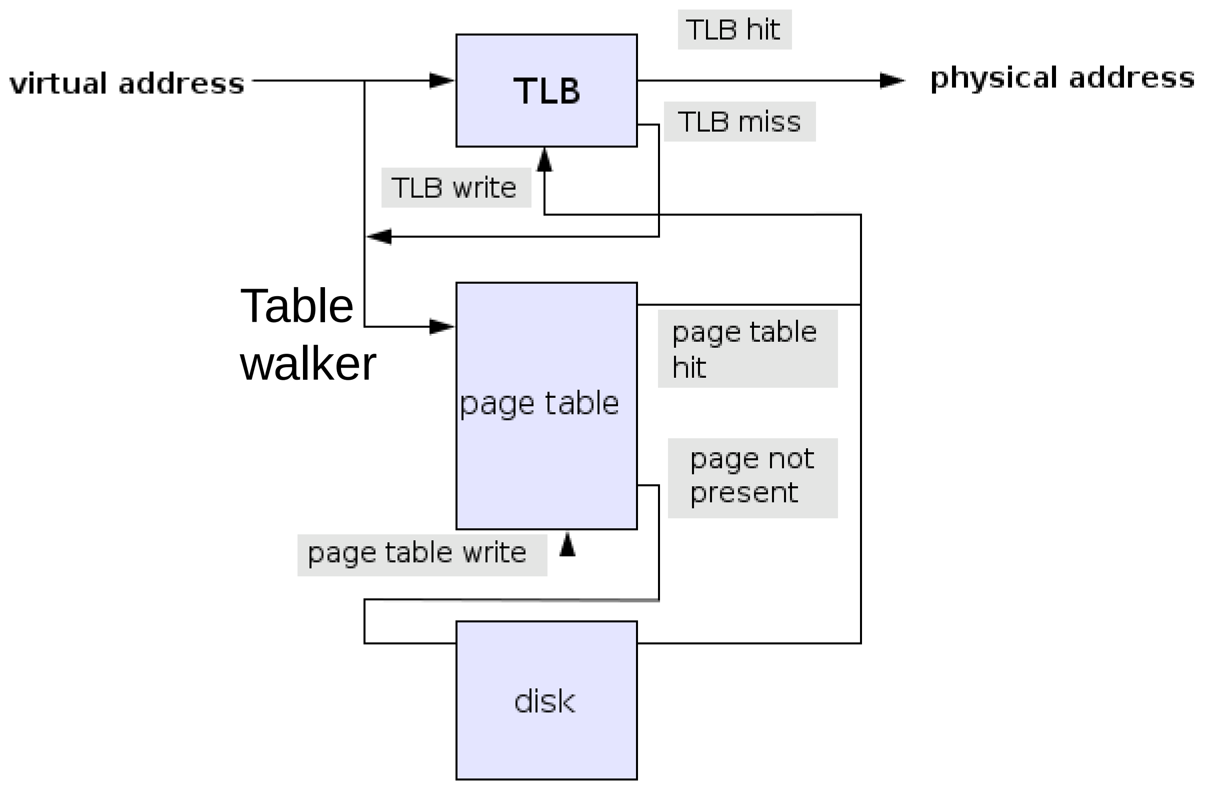


2 memory accesses!

How do we make the page table look-up faster?

Page identification (TLB)

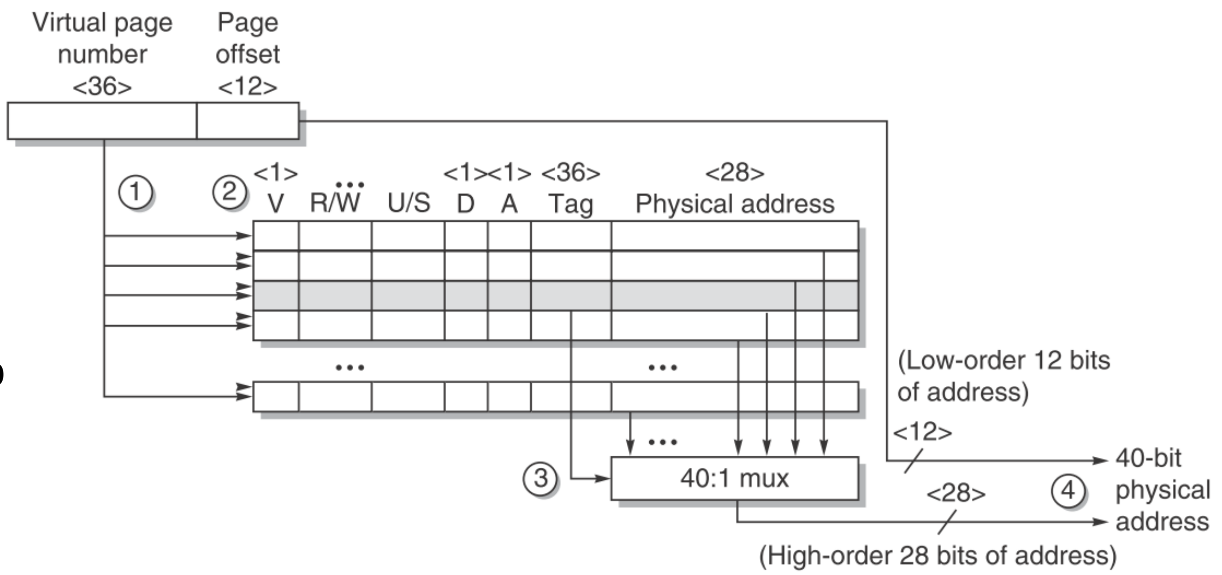
- **How do we avoid two (or more) memory references for each original memory reference?**
  - Cache address translations – Translation Look-aside Buffer (TLB)



Page identification (TLB)

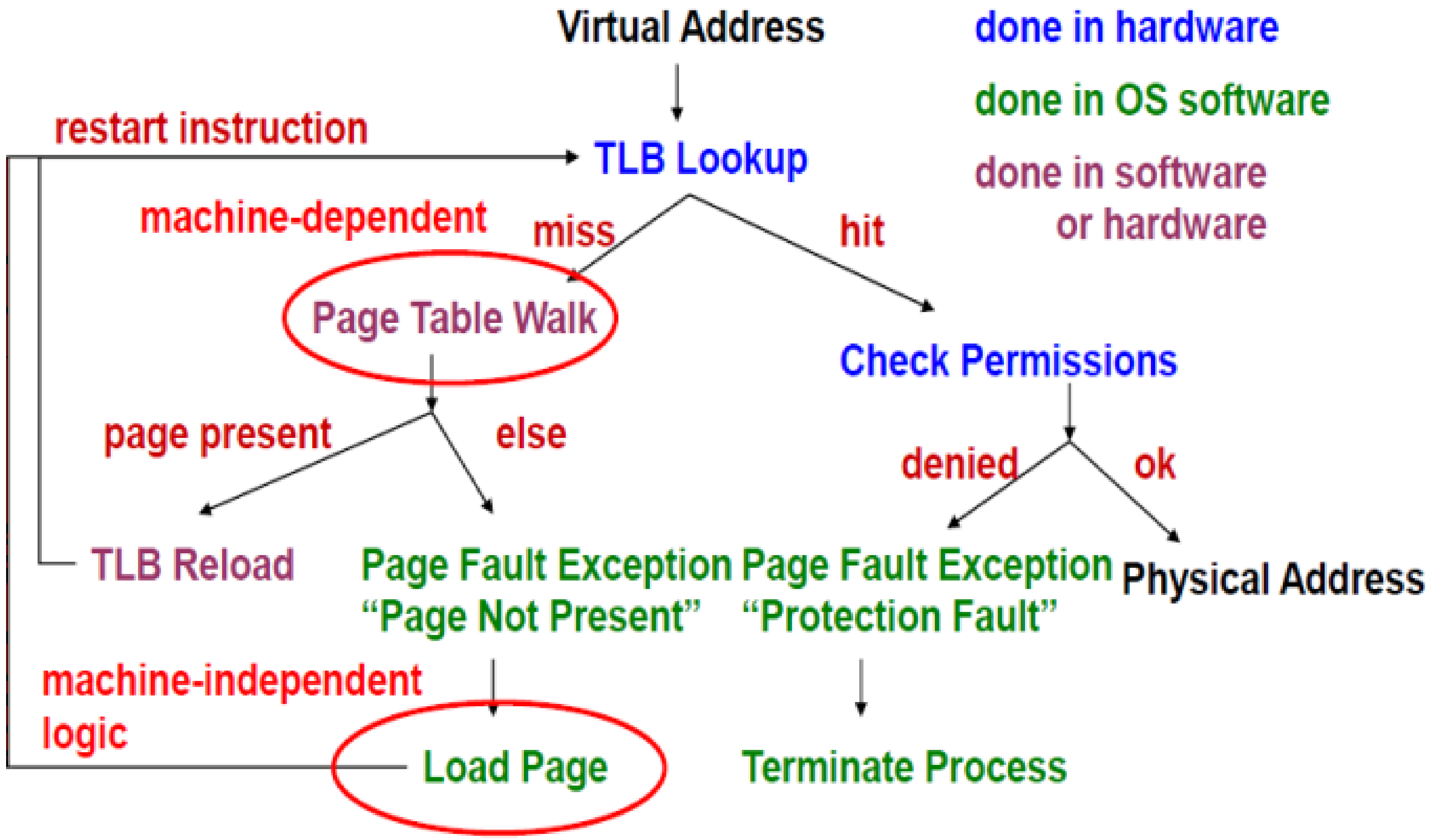
- **Translation look aside buffer (translation buffer)**
  - Tag: virtual address
  - Data portion: physical address, control bits
  - This example: Fully associative placement

1. VPN is extracted
2. Protections checked
3. One of 40 entries muxed (or miss registered)
4. Physical page address combined with offset to generate real address



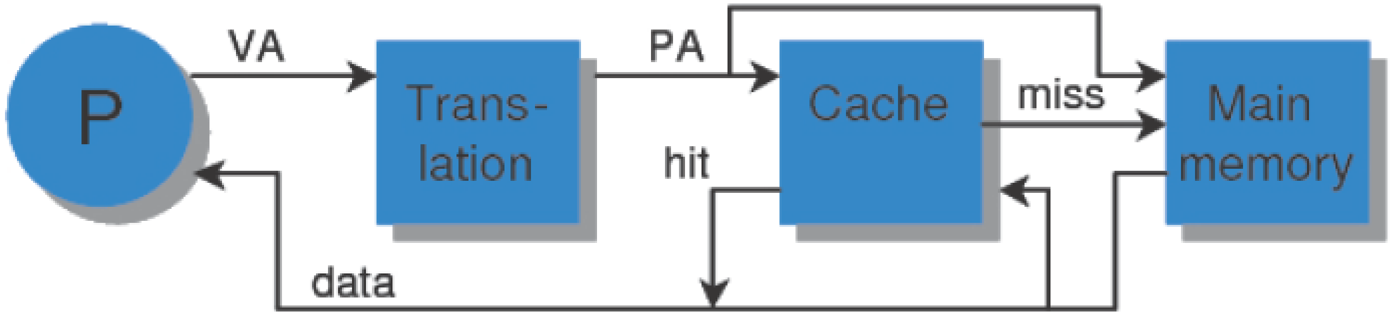
Opteron data TLB organization

Address translation and TLB

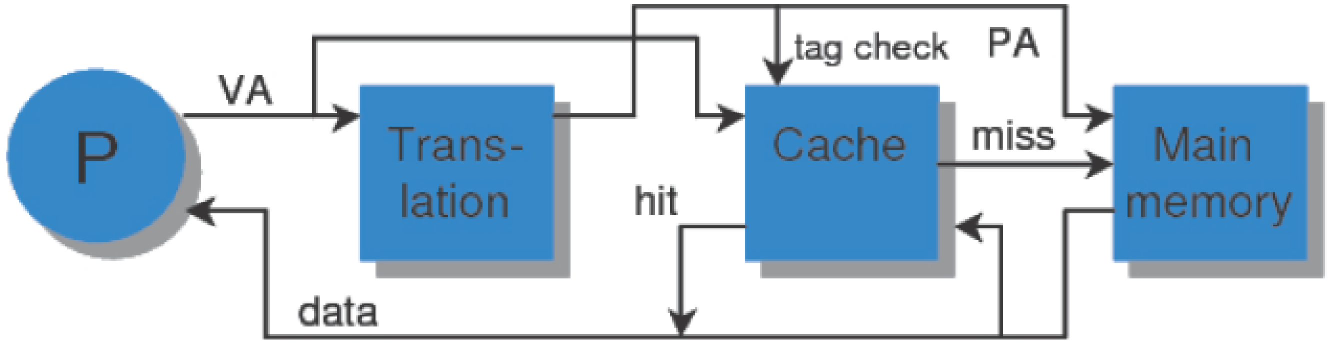


Reduce hit time 2: Address translation

- Processor uses virtual addresses (VA) while caches and main memory use physical addresses (PA)

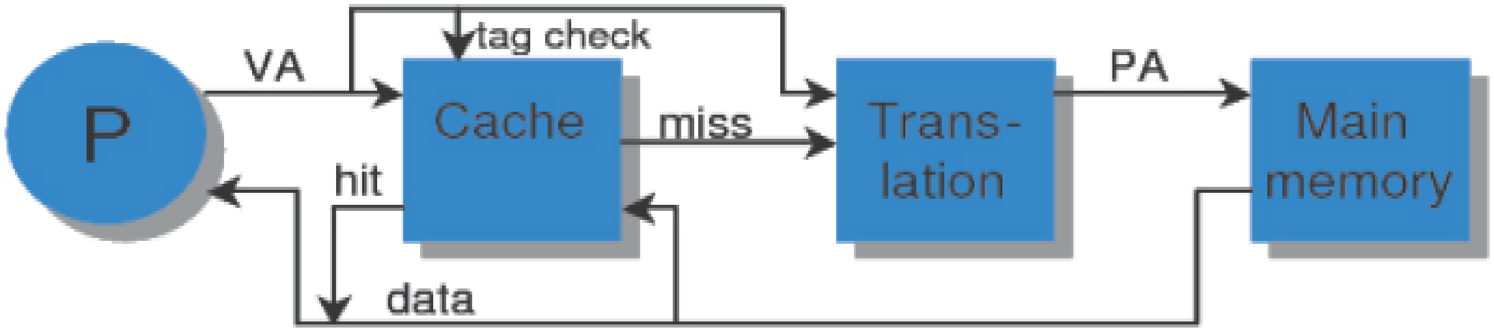


- Use the virtual address to index the cache in parallel



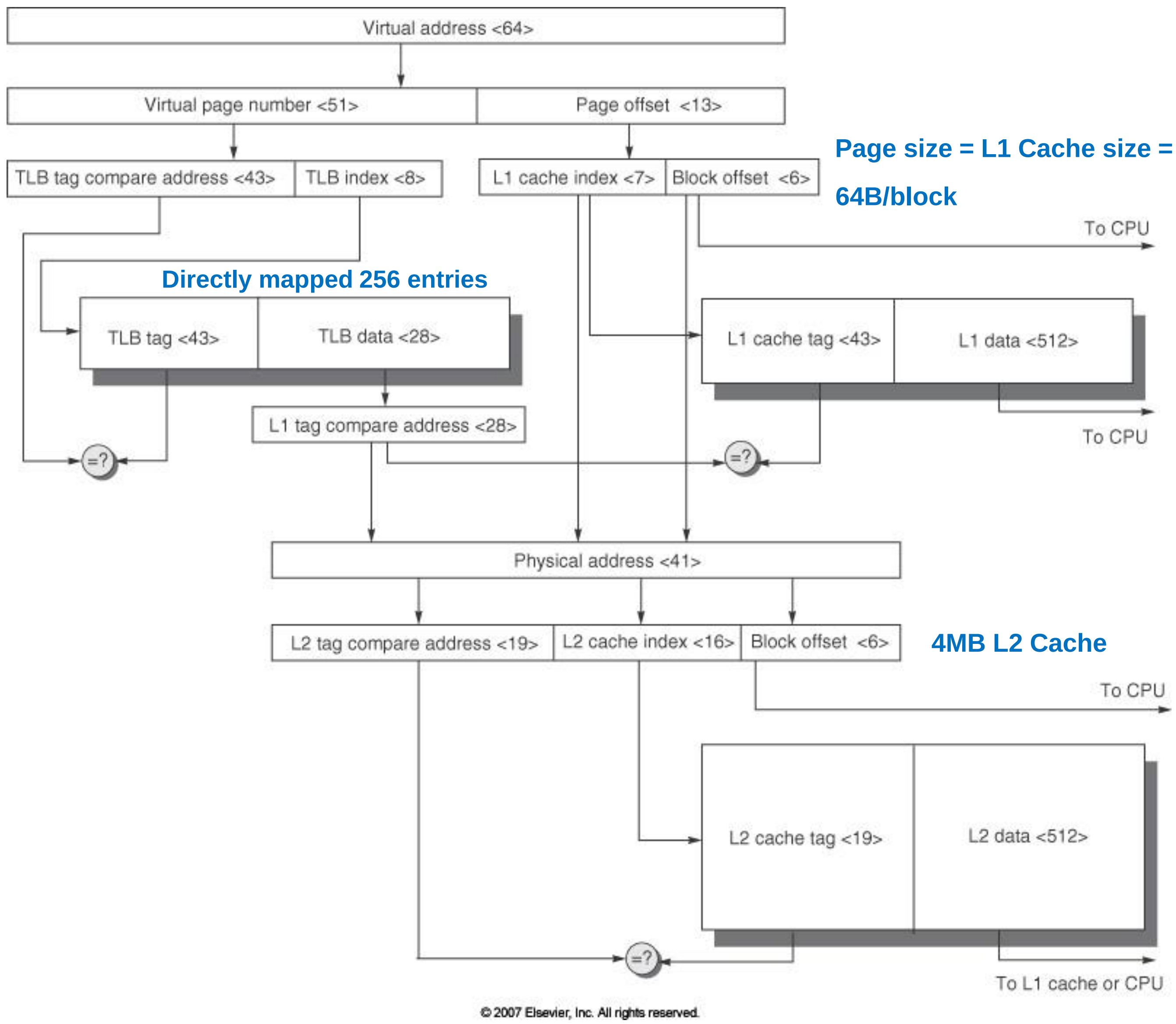
Reduce hit time 2: Address translation

- Use virtual addresses to both index cache and tag check



- Processes have different virtual address spaces (change process requires cache flush)
- Two virtual addresses may map to the same physical address – synonyms or aliases

Address translation cache and VM



Page size = L1 Cache size = 1KB

64B/block

4MB L2 Cache

Page replacement

- **Most important: *minimize number of page faults***

- **Replacement in cache handled by HW**
- **Replacement in VM handled by SW**

Page replacement strategies:

- **FIFO – First-In-First-Out**
- **LRU – Least Recently Used**
  - Approximation
  - Each page has a reference bit that is set on a reference
  - The OS periodically resets the reference bits
  - When a page needs to be replaced, a page with a reference bit that is not set is chosen

Write strategy

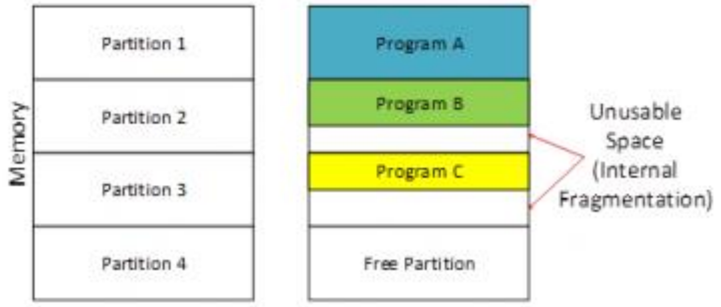
Write back or Write through?

- **Write back! + dirty bit**
- **Write through is impossible to use:**
  - Too long access time to disk
  - The write buffer would need to be very large
  - The I/O system would need an extremely high bandwidth

Page size

Larger page size?

- **Advantages**
  - Size of page table = $k * \frac{2^{addrbits}}{2^{pagebits}} \sim \frac{1}{page\ size}$
  - More memory can be mapped → reducing TLB misses (# of entries in TLB is limited)
  - More efficient to transfer large pages
- **Disadvantages**
  - More wasted storage, internal fragmentation
  - High bandwidth requirement
  - Long process start-up times (if the process size is much smaller than the page size)

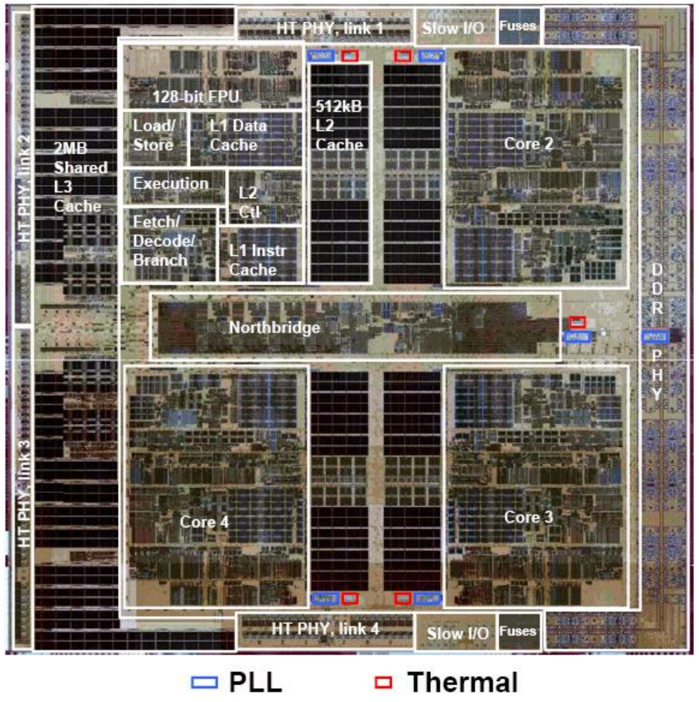


Cache vs VM

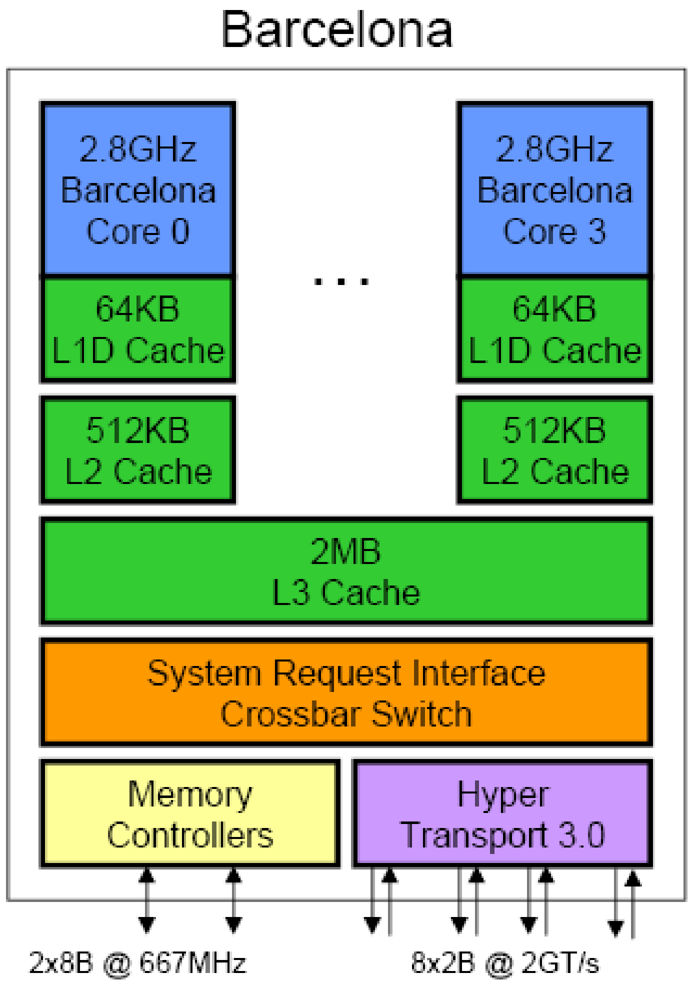
	Cache-MM	**MM-disk**
Access time ratio ("speed gap")	1:5 - 1:15	1:10000 - 1:1000000
Hit time	1-2 cycles	40-100 cycles
Hit ratio	0.90-0.99	0.99999-0.9999999
Miss (page fault) ratio	0.01-0.10	0.00000001-0.000001
Miss penalty	10-100 cycles	1M-6M cycles
CPU during block transfer	blocking/non-blocking	task switching
Block (page) size	16-128 bytes	4Kbytes - 64Kbytes
Implemented in	hardware	hardware + software
Mapping	Direct or set-associative	Page table ("fully associative")
Replacement algorithm	Not crucial	Very important (LRU)
Write policy	Many choices	Write back
Direct access to slow memory	Yes	No

Outline

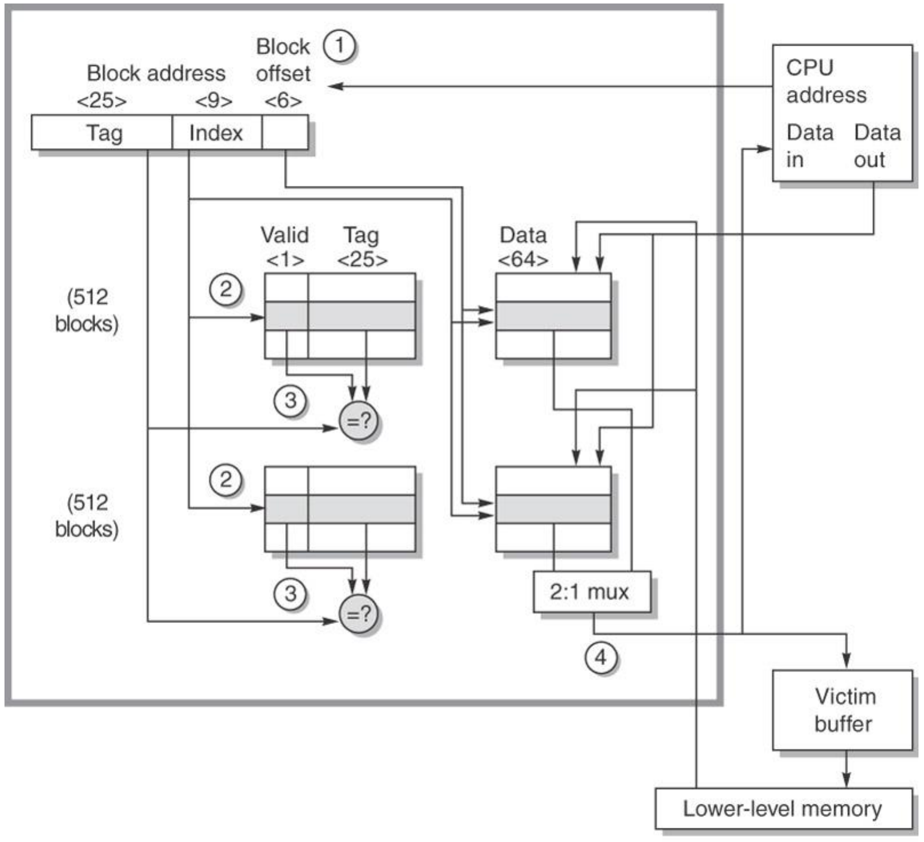
- Reiteration
- Virtual memory
- **Case study AMD Opteron**
- Summary



Memory overview



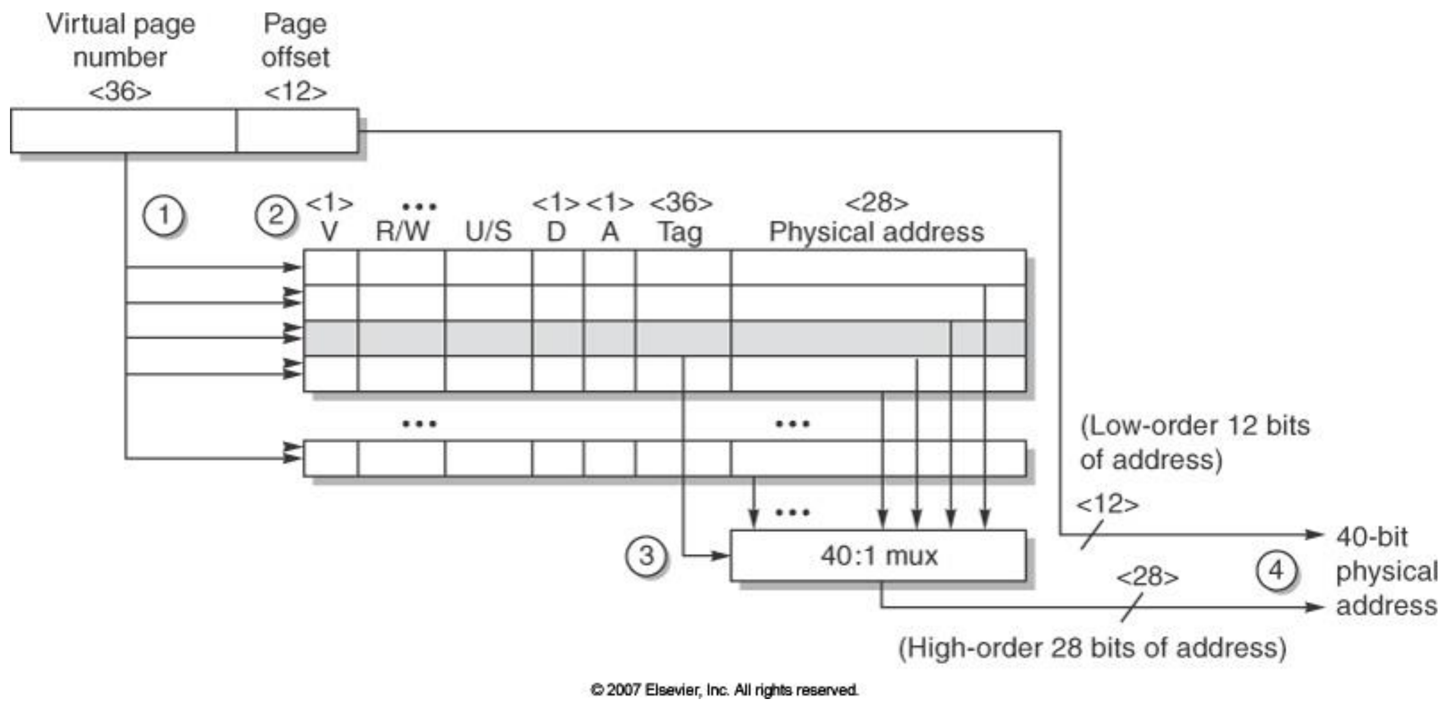
Basic L1 data cache



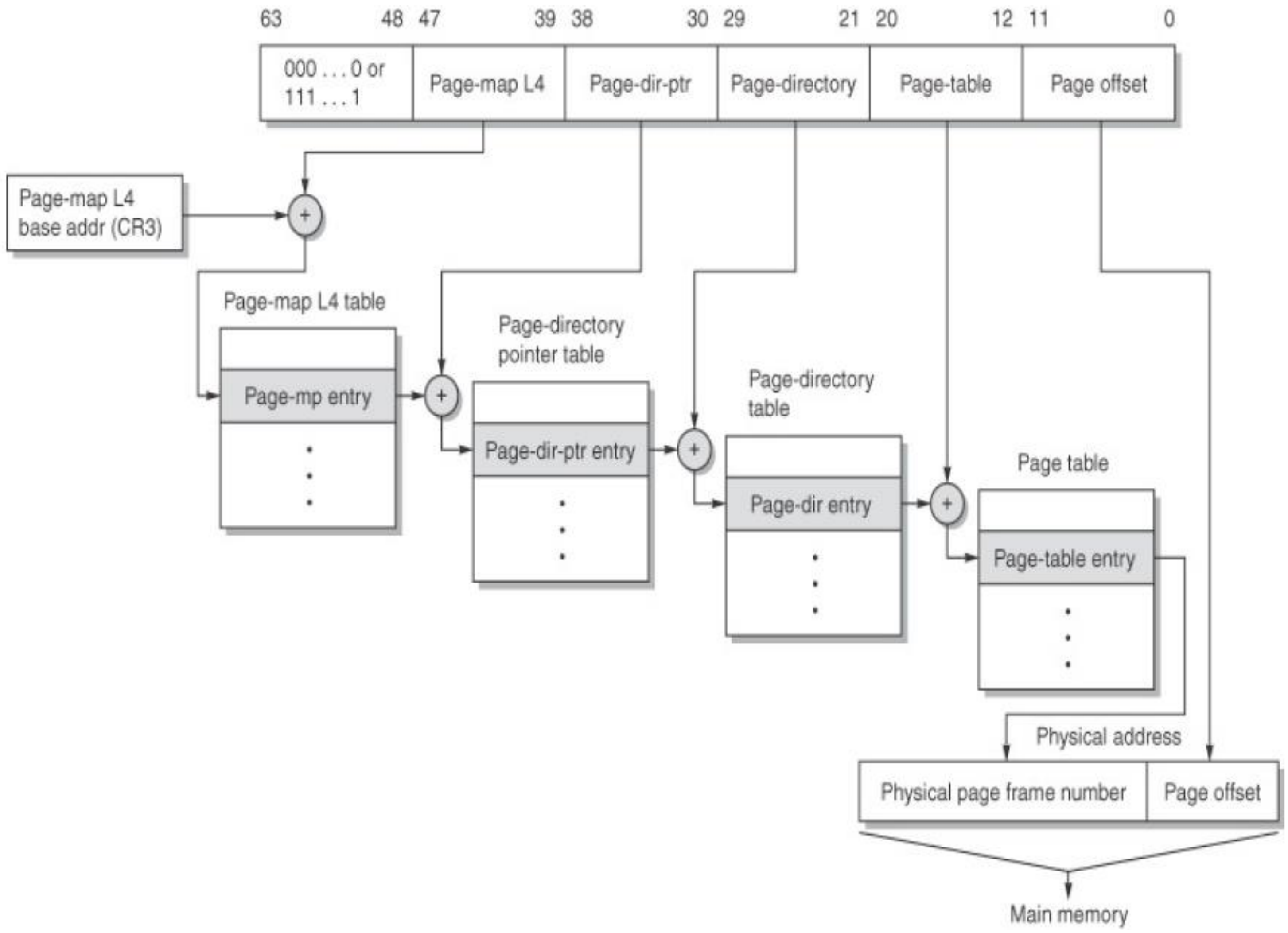
- 64 Kbyte, 64 byte block size ⇒ 1024 blocks
- 2-way set associative ⇒ 512 sets
- write-back, write allocate
- 8 block write buffer (victim)
- LRU - 1 bit

Data TLB

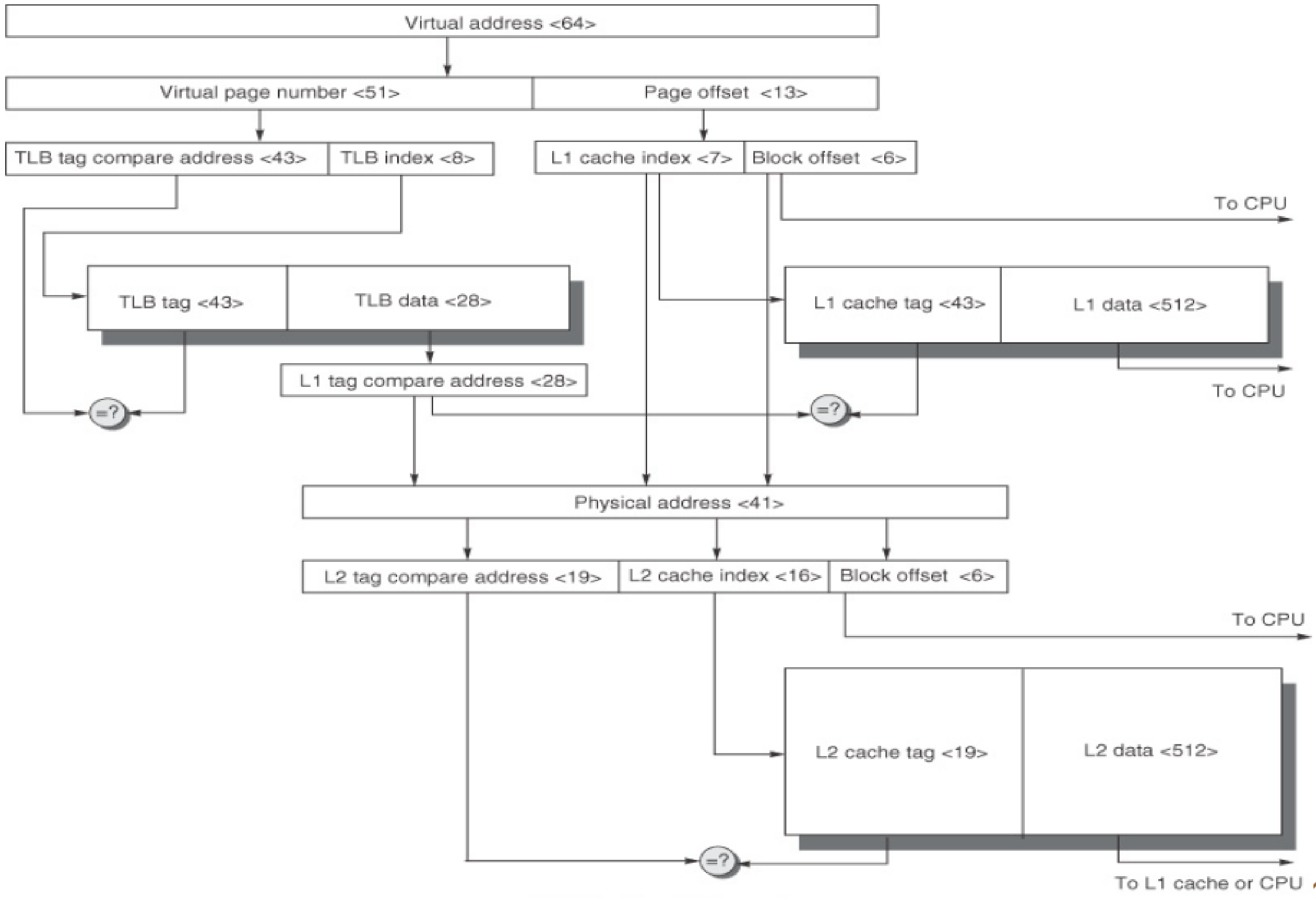
- 40 page table entries
- Fully associative
- Valid bit, kernel & user read/write permissions, protection



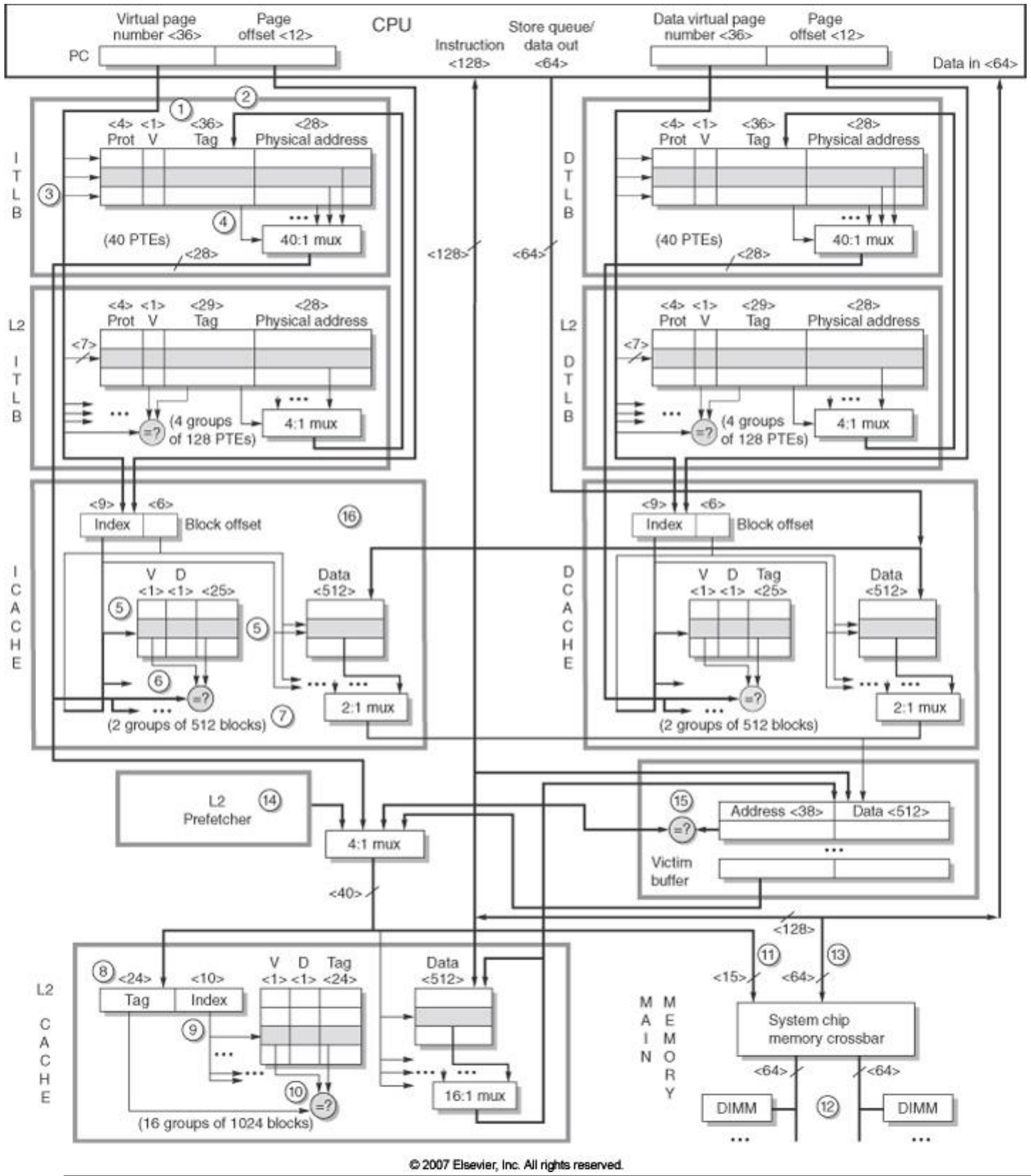
Page table structure



AMD Opteron cache + TLB



The memory hierarchy of AMD Opteron



- Separate Instr & Data TLB and Caches
- 2-level TLBs
  - L1 TLBs fully associative
  - L2 TLBs 4 way set associative
- Write buffer (and Victim cache)
- Way prediction
- Line prediction: prefetch
- hit under 10 misses
- 1 MB L2 cache, shared, 16 way set associative, write back

Outline

- Reiteration
- Virtual memory
- Case study AMD Opteron
- **Summary**

Summary memory hierarchy

Hide CPU - memory performance gap
Memory hierarchy with several levels
Principle of locality

Cache memories:

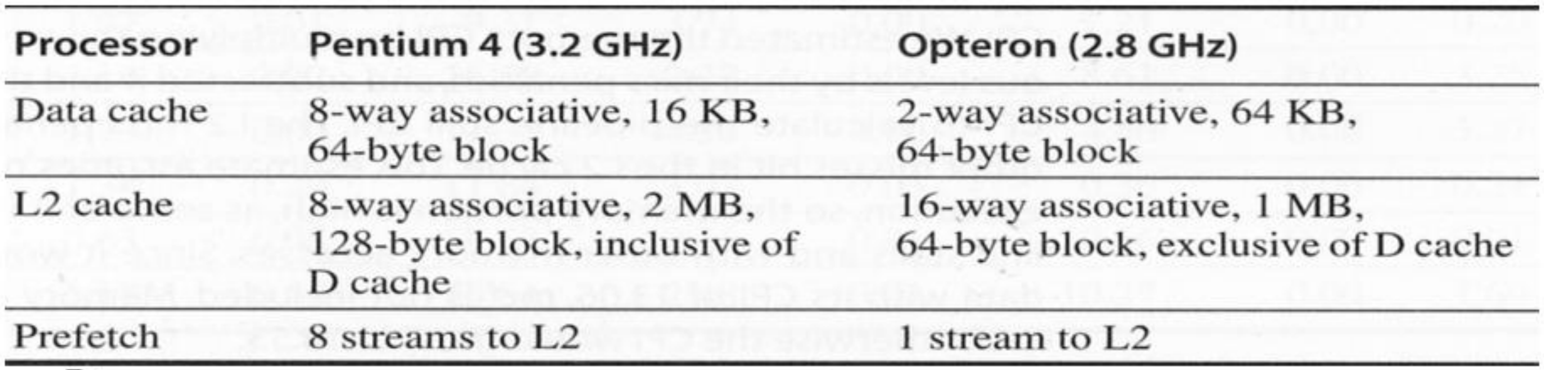
- Fast, small - Close to CPU
- Hardware
- TLB
- CPU performance equation
- Average memory access time
- Optimizations

Virtual memory:

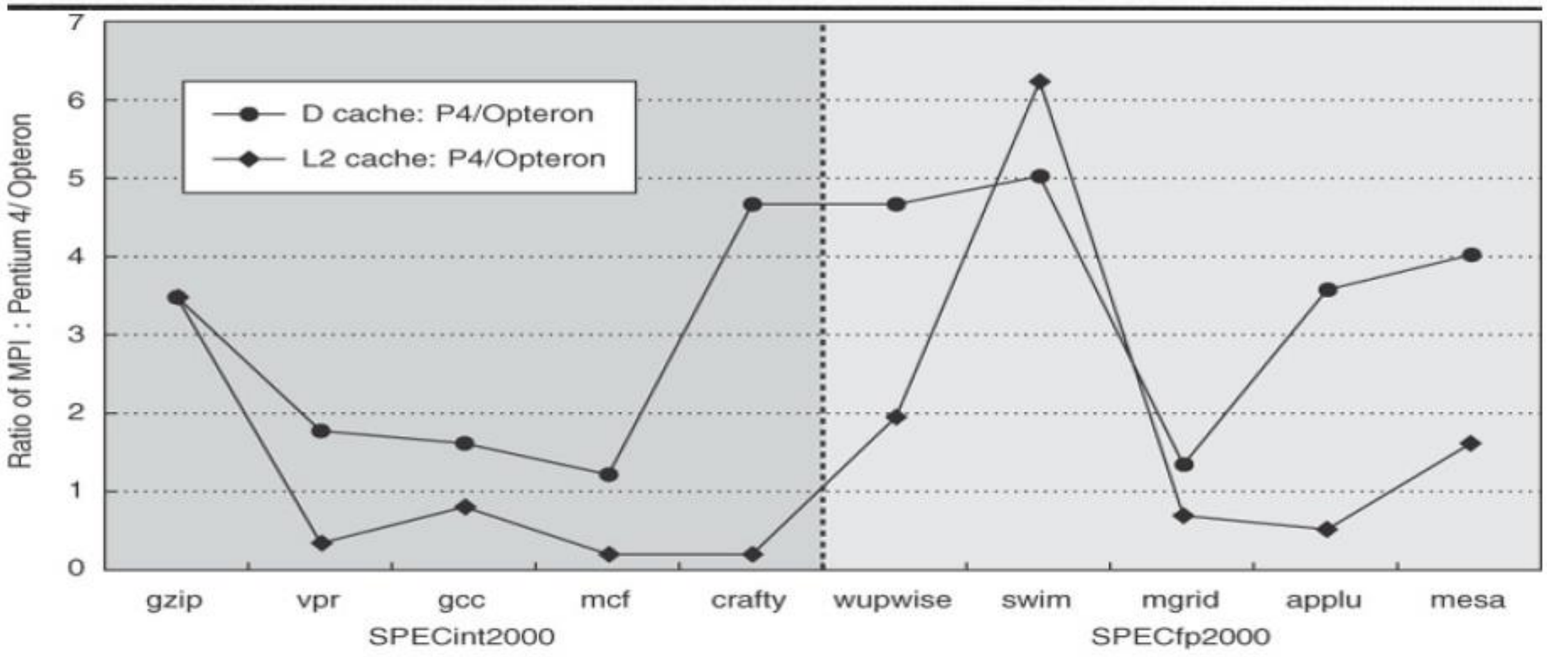
- Slow, big - Close to disk
- Software
- TLB
- Page-table
- Very high miss penalty $\Longrightarrow$ miss rate must be low
- Also facilitates: relocation; memory protection; and multiprogramming

Same 4 design questions - Different answers

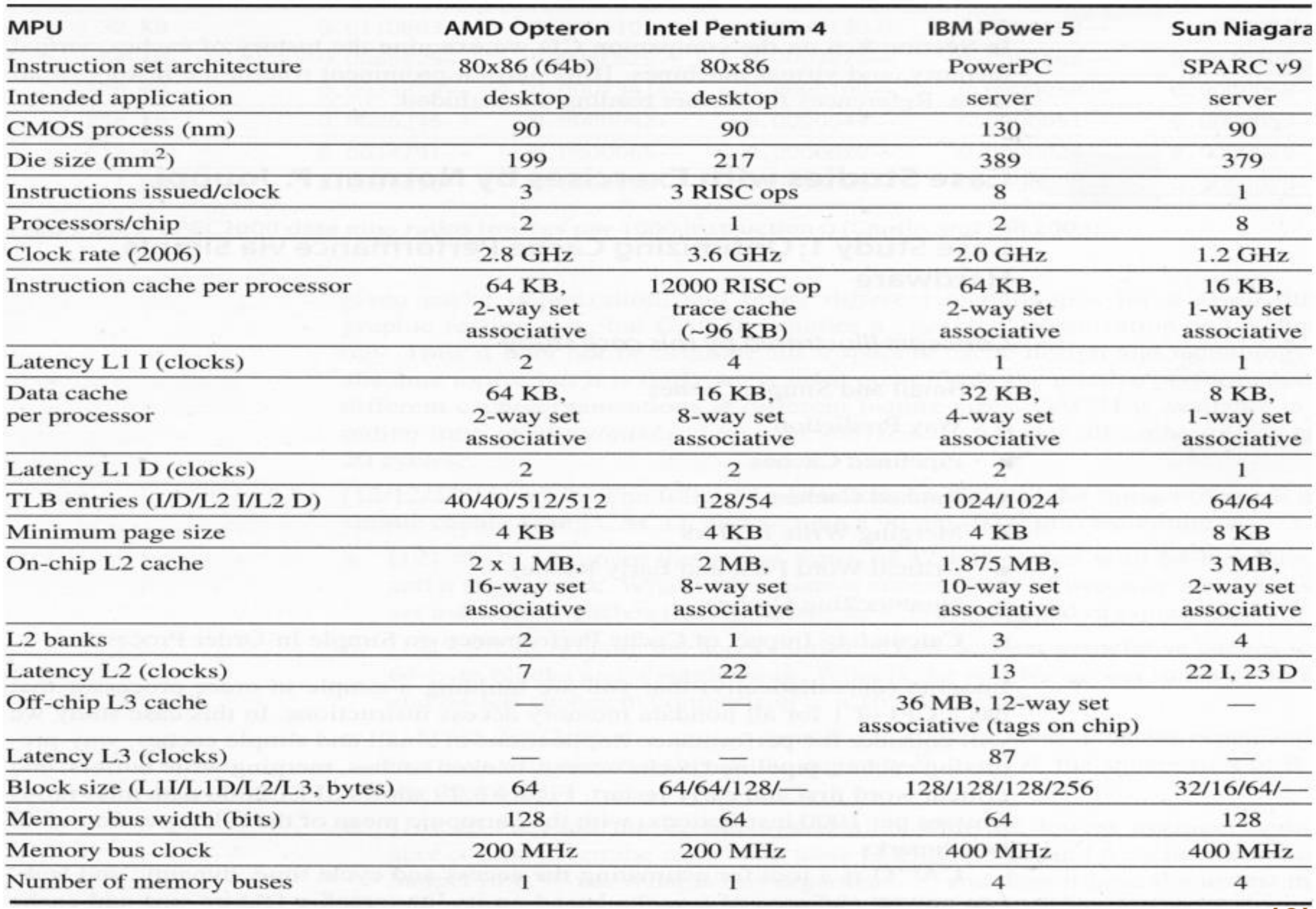
Program behavior vs cache organization



Processor	Pentium 4 (3.2 GHz)	Opteron (2.8 GHz)
Data cache	8-way associative, 16 KB, 64-byte block	2-way associative, 64 KB, 64-byte block
L2 cache	8-way associative, 2 MB, 128-byte block, inclusive of D cache	16-way associative, 1 MB, 64-byte block, exclusive of D cache
Prefetch	8 streams to L2	1 stream to L2



Example organizations



MPU	AMD Opteron	Intel Pentium 4	IBM Power 5	Sun Niagara
Instruction set architecture	80x86 (64b)	80x86	PowerPC	SPARC v9
Intended application	desktop	desktop	server	server
CMOS process (nm)	90	90	130	90
Die size (mm^2)	199	217	389	379
Instructions issued/clock	3	3 RISC ops	8	1
Processors/chip	2	1	2	8
Clock rate (2006)	2.8 GHz	3.6 GHz	2.0 GHz	1.2 GHz
Instruction cache per processor	64 KB, 2-way set associative	12000 RISC op trace cache (~96 KB)	64 KB, 2-way set associative	16 KB, 1-way set associative
Latency L1 I (clocks)	2	4	1	1
Data cache per processor	64 KB, 2-way set associative	16 KB, 8-way set associative	32 KB, 4-way set associative	8 KB, 1-way set associative
Latency L1 D (clocks)	2	2	2	1
TLB entries (I/D/L2 I/L2 D)	40/40/512/512	128/54	1024/1024	64/64
Minimum page size	4 KB	4 KB	4 KB	8 KB
On-chip L2 cache	2 x 1 MB, 16-way set associative	2 MB, 8-way set associative	1.875 MB, 10-way set associative	3 MB, 2-way set associative
L2 banks	2	1	3	4
Latency L2 (clocks)	7	22	13	22 I, 23 D
Off-chip L3 cache	—	—	36 MB, 12-way set associative (tags on chip)	—
Latency L3 (clocks)	—	—	87	—
Block size (L1I/L1D/L2/L3, bytes)	64	64/64/128/—	128/128/128/256	32/16/64/—
Memory bus width (bits)	128	64	64	128
Memory bus clock	200 MHz	200 MHz	400 MHz	400 MHz
Number of memory buses	1	1	4	4