CMSC 611: Advanced Computer Architecture

Cache and Memory

Classification of Cache Misses

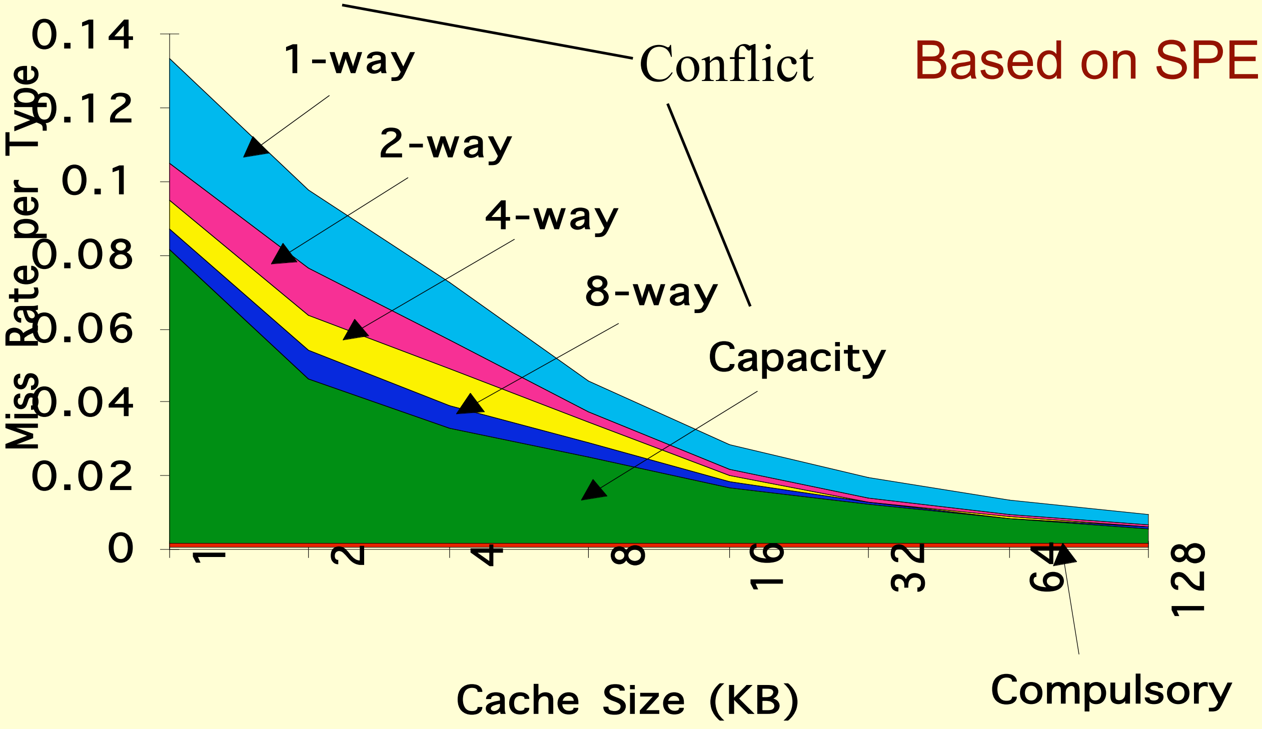
- Compulsory
  - The first access to a block is never in the cache. Also called cold start misses or first reference misses.
    (Misses in even an Infinite Cache)
- Capacity
  - If the cache cannot contain all the blocks needed during execution of a program, blocks must be discarded and later retrieved.
    (Misses in Fully Associative Size X Cache)
- Conflict
  - If block-placement strategy is set associative or direct mapped, blocks may be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses.
    (Misses in N-way Associative, Size X Cache)

Improving Cache Performance

- Capacity misses can be damaging to the performance (excessive main memory access)
- Increasing associativity, cache size and block width can reduces misses
- Changing cache size affects both capacity and conflict misses since it spreads out references to more blocks
- Some optimization techniques that reduces miss rate also increases hit access time

Miss Rate Distribution

- Compulsory misses are very small compared to other categories
- Capacity-based misses are diminishing with increased cache sizes
- Increasing associativity limits the potential of placement conflicts



Based on SPEC92

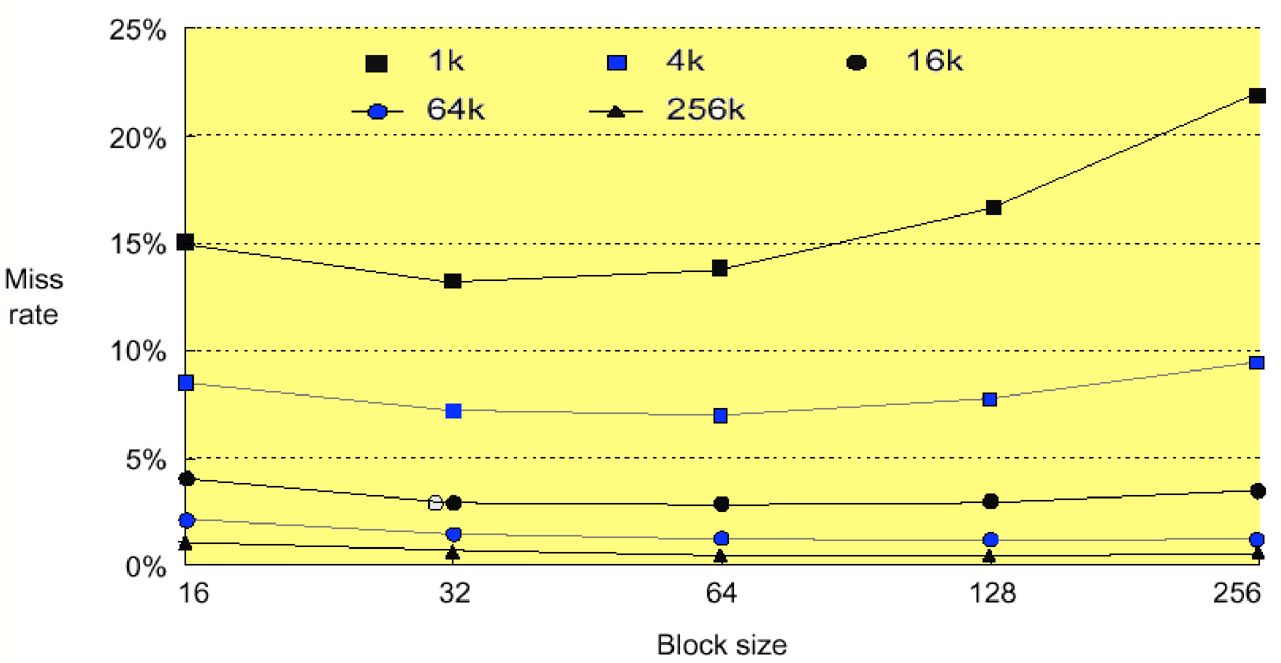
Techniques for Reducing Misses

$$\text{CPUtime} = \text{IC} \times \left(\text{CPI}_{\text{Execution}} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

1. Reducing Misses via Larger Block Size
2. Reducing Misses via Higher Associativity
3. Reducing Misses via Victim Cache
4. Reducing Misses via Pseudo-Associativity
5. Reducing Misses by H/W Prefetching Instr. and Data
6. Reducing Misses by S/W Prefetching Data
7. Reducing Misses by Compiler Optimizations

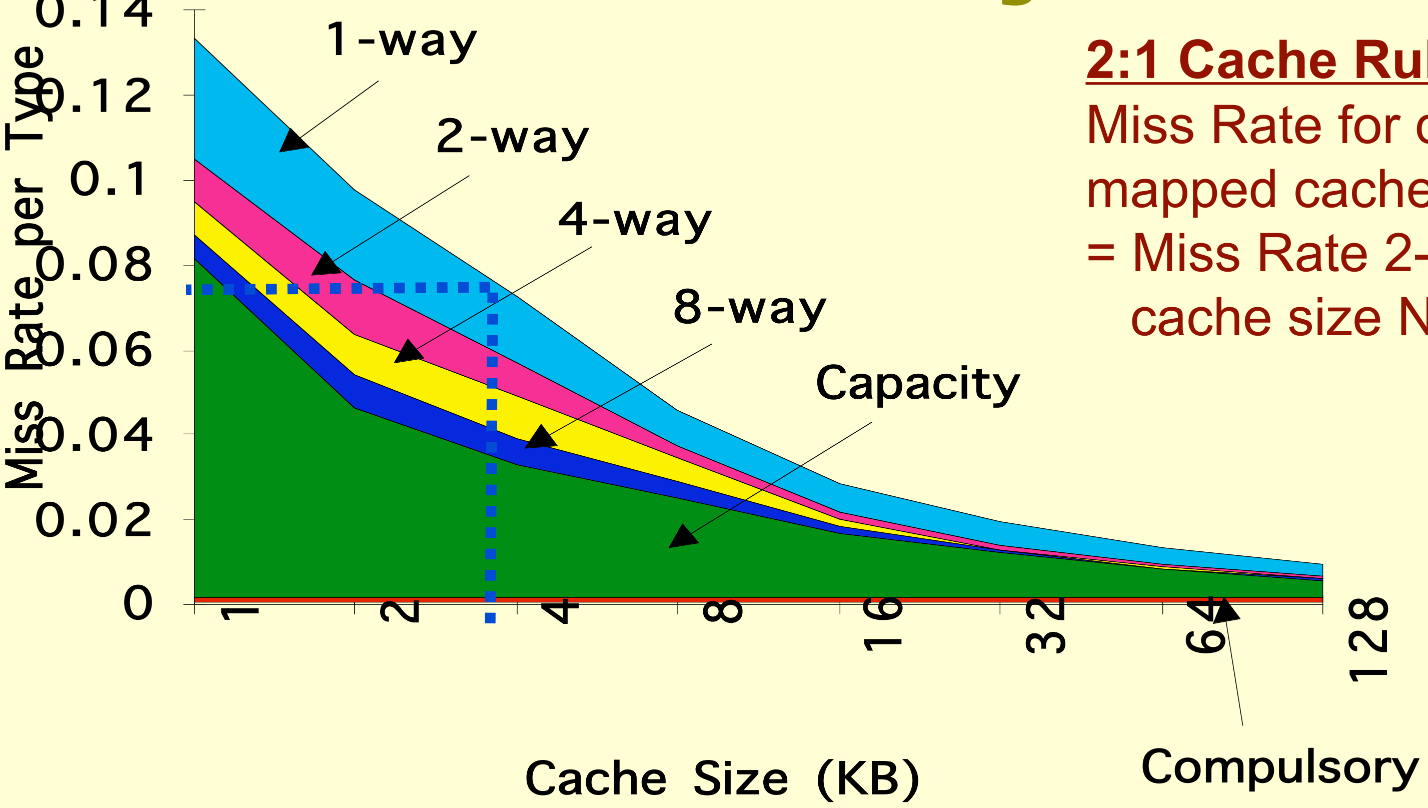
Slide: Dave Patterson

Reduce Misses via Larger Block Size



- Larger block sizes reduces compulsory misses (principle of spatial locality)
- Conflict misses increase for larger block sizes since cache has fewer blocks
- The miss penalty usually outweighs the decrease in the miss rate making large block sizes less favored

Reduce Misses via Higher Associativity



2:1 Cache Rule:
Miss Rate for direct mapped cache of size N = Miss Rate 2-way cache size N/2

- Greater associativity comes at the expense of larger hit access time
- Hardware complexity grows for high associativity and clock cycle increases

Example

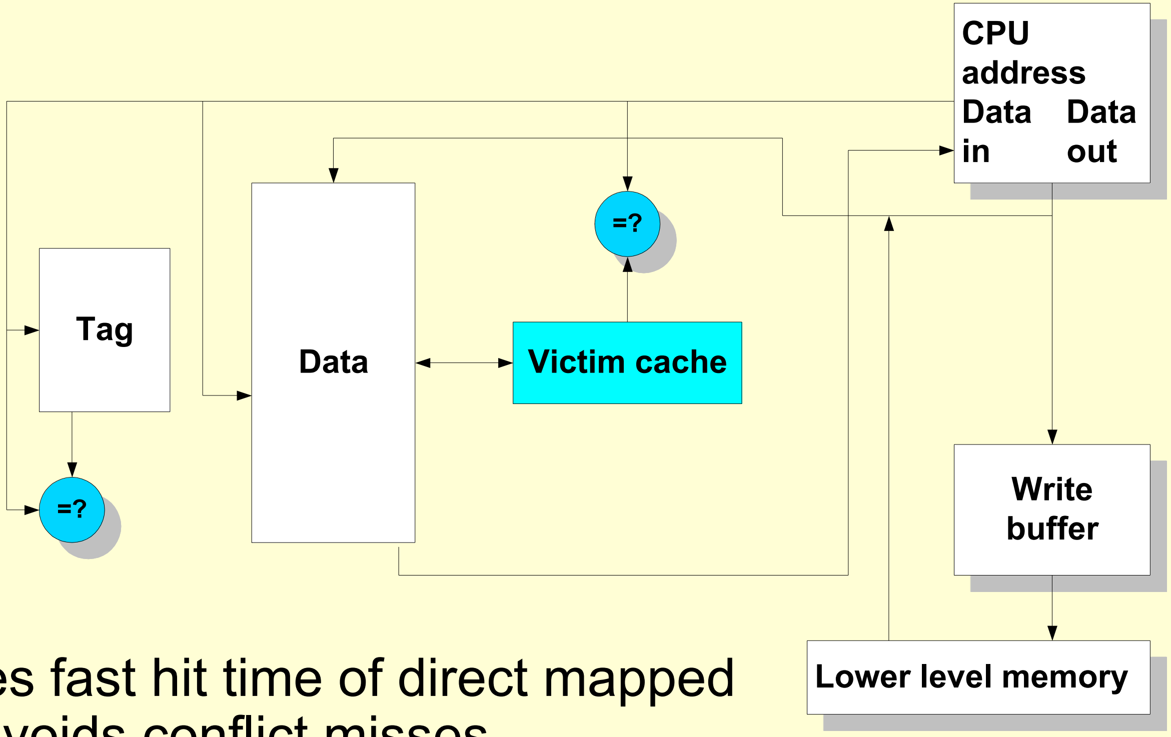
Assume hit time is 1 clock cycle and average miss penalty is 50 clock cycles for a direct mapped cache. The clock cycle increases by a factor of 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way associative cache. Compare the average memory access based on the previous figure miss rates

Cache Size (KB)	Associativity			
	1-way	2-way	4-way	8-way
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	**1.79**
64	1.70	1.60	1.57	**1.59**
128	1.50	1.45	1.42	**1.44**

A good size of direct mapped cache can be very efficient given its simplicity

High associativity becomes a negative aspect

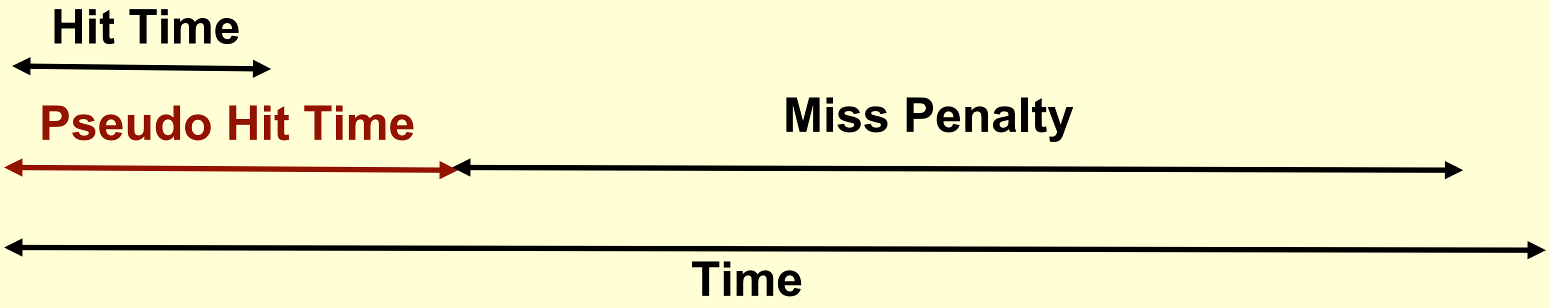
Victim Cache Approach



- Combines fast hit time of direct mapped yet still avoids conflict misses
  - Adds small fully asssociative cache between the direct mapped cache and memory to place data discarded from cache
  - Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
  - Technique is used in Alpha, HP machines and does not impair the clock rate

Pseudo-Associativity Mechanism

- Combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way set associative cache
- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)
- Simplest implementation inverts the most significant bit in the index field to find the other pseudo set
- Pseudo associative caches has two hit times (hit and pseudo hit)
- To limit the impact of hit time variability on performance, the contents of the blocks can be swapped



- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Better for caches not tied directly to processor (L2)
  - Used in MIPS R1000 L2 cache, similar in UltraSPARC

Slide: Dave Patterson

H/W Pre-fetching of Instructions & Data

- The hardware pre-fetch instructions and data while handing other cache misses, assuming that the pre-fetched items will be referenced shortly
- Pre-fetching relies on having extra memory bandwidth that can be used without penalty

Average memory access time$_{\text{pre-fetch}}$ = Hit time + Miss rate □ Pre-fetch hit rate

□ 1 + Miss rate □ (1 - Pre-fetch hit rate) □ Miss penalty

- Examples of Instruction Pre-fetching:
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in “stream buffer”
  - On miss check stream buffer
- Works with data blocks too:
  - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches

Slide: Dave Patterson

Software Pre-fetching Data

- Uses special instructions to pre-fetch data:
  - Load data into register (HP PA-RISC loads)
  - Cache Pre-fetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
- Special pre-fetching instructions cannot cause faults (undesired exceptions) since it is a form of speculative execution
- Makes sense if the processor can proceeds without blocking for a cache access (lock-free cache)
- Loops are typical target for pre-fetching after unrolling (miss penalty is small) or after applying software pipelining (miss penalty is large)
- Issuing Pre-fetch Instructions takes time
  - Is cost of pre-fetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

```
for (i = 0; i < 3; i = i+1)
  for (j = 0; j < 100; j = j+1)
    a[i][j] = b[j][0] * b[j+1][0];
```

```
for (j = 0; j < 100; j = j+1)
   pre-fetch (b[i+7][0]);
   a[0][j] = b[j][0] * b[j+1][0];
   for (i = 1; i < 3; i = i+1)
     pre-fetch (a[i][j+7]);
     a[i-1][j] = b[j][0] * b[j+1][0];
```

Compiler-based Cache Optimizations

- Complier-based cache optimization reduces the miss rate without any hardware change or complexity
- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- For Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to determine likely conflicts among groups of instructions
- For Data
  - Merging Arrays: improve spatial locality by single array of compound elements vs. two 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

Slide: Dave Patterson

Examples

***Merging Arrays*:**

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];
```

→

```
/* After: 1 array of stuctures */
struct merge {
        int val;
        int key;
};
struct merge merged_array[SIZE];
```

- Reduces misses by improving spatial locality through combined arrays that are accessed simultaneously

***Loop Interchange*:**

```
/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
         x[i][j] = 2 * x[i][j];
```

→

```
/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
         x[i][j] = 2 * x[i][j];
```

- Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Loop Fusion Example

- Some programs have separate sections of code that access the same arrays
  - (performing different computation on common data)
- Fusing multiple loops into a single loop allows the data in cache to be used repeatedly before being swapped out
- Loop fusion reduces missed through improved temporal locality (rather than spatial locality in array merging and loop interchange)

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
          a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
          d[i][j] = a[i][j] + c[i][j];
```

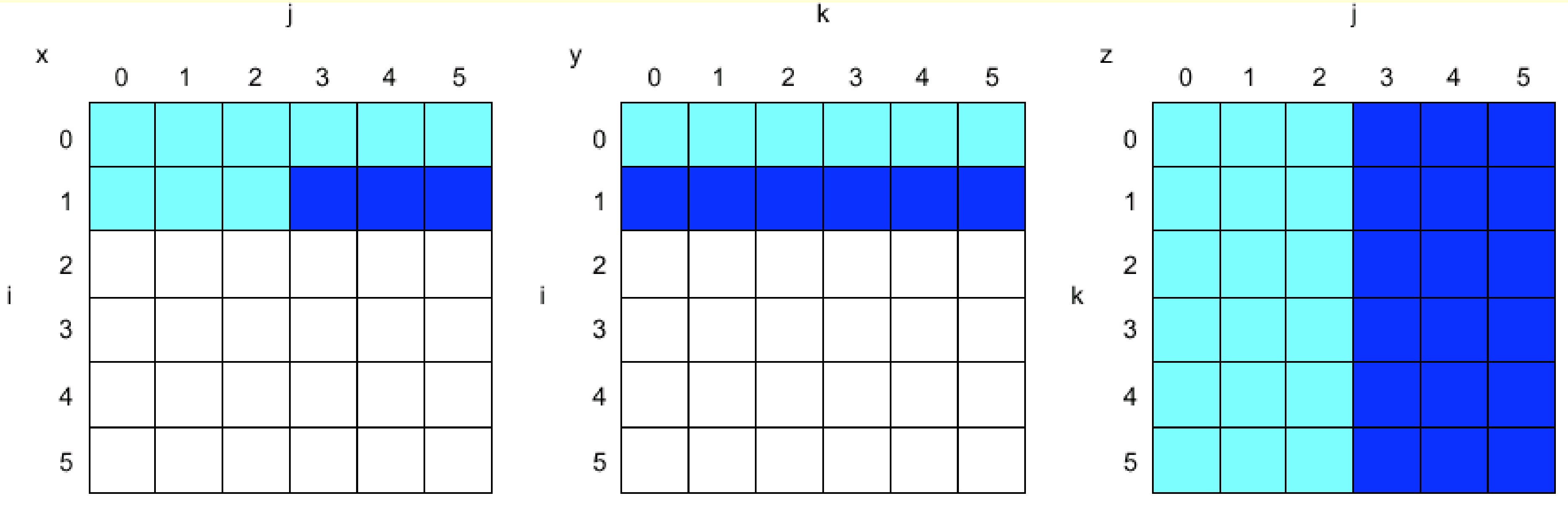
```
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1) {
          a[i][j] = 1/b[i][j] * c[i][j];
          d[i][j] = a[i][j] + c[i][j];
}
```

Accessing array “a” and “c” would have caused twice the number of misses without loop fusion

Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1) {
    r = 0;
    for (k = 0; k < N; k = k+1)
        r = r + y[i][k] * z[k][j];
    x[i][j] = r;
  }
```

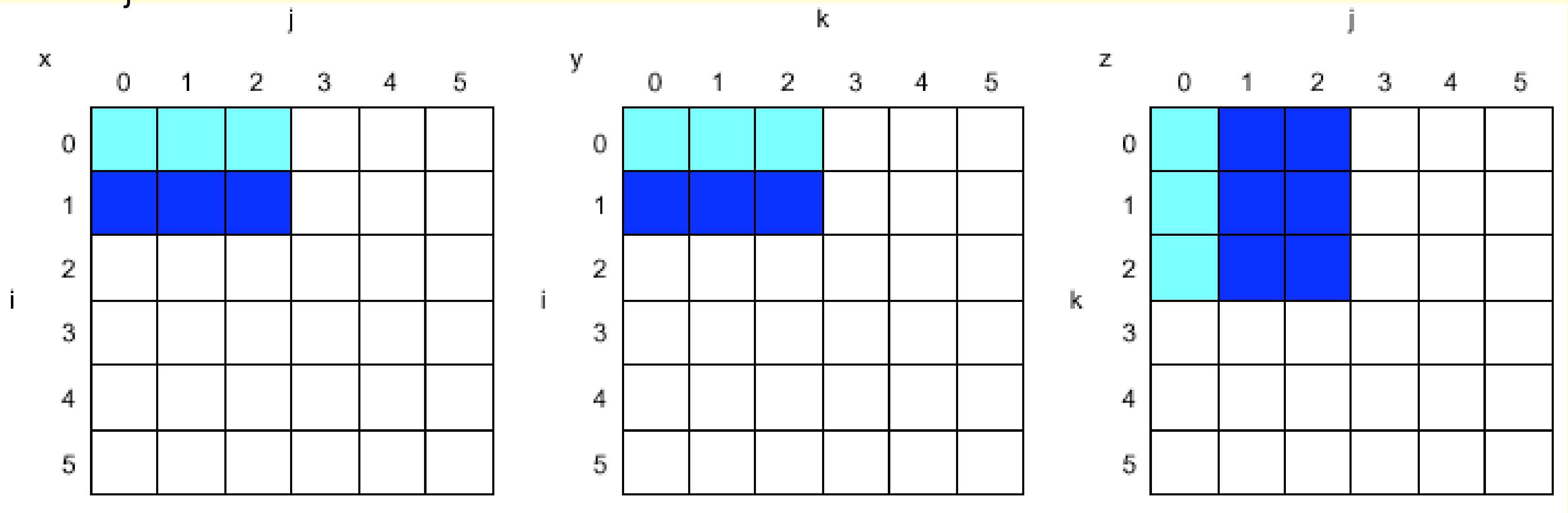
- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]
- Capacity Misses a function of N & Cache Size:
  - 3 × N × N × 4 bytes => no capacity misses;
- Idea: compute on B × B sub-matrix that fits



Blocking Example

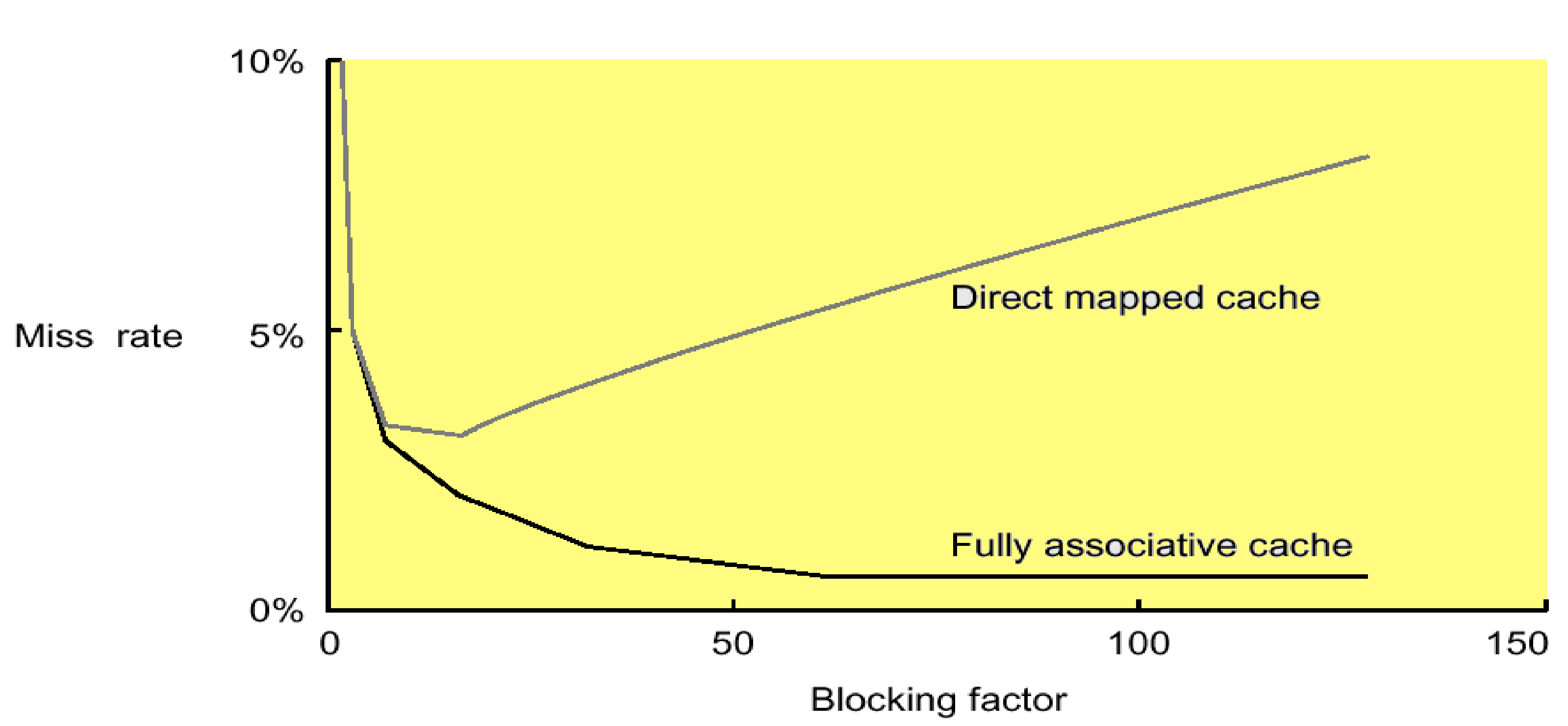
```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1) {
        r = 0;
        for (k = kk; k < min(kk+B-1,N); k = k+1) {
            r = r + y[i][k] * z[k][j];};
            x[i][j] = x[i][j] + r;
   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

- B called Blocking Factor
- Memory words accessed
  2N3 + N2 → 2N3/B +N2
- Conflict misses can go down too
- Blocking is also useful for register allocation



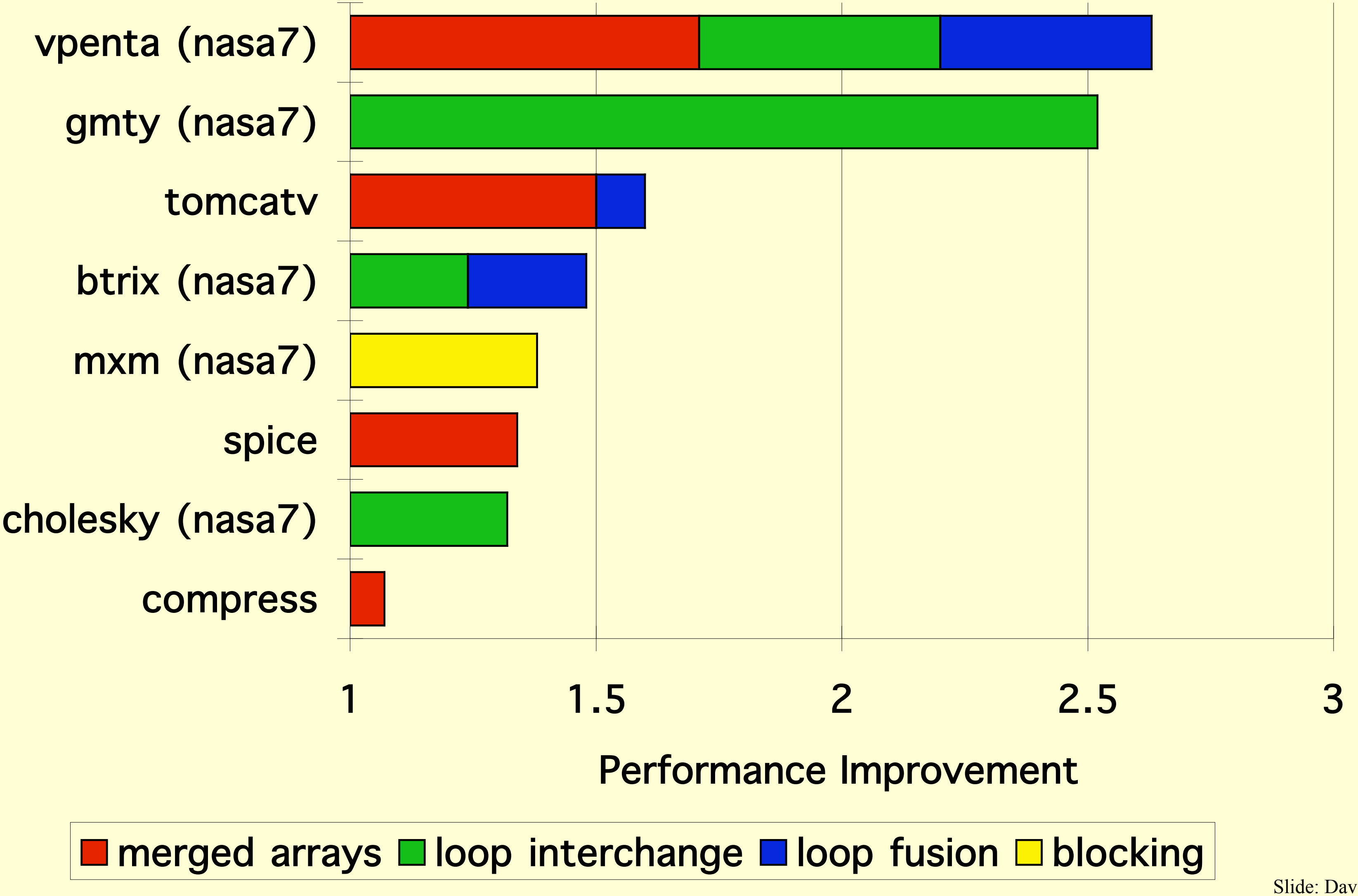
Blocking Factor

- Traditionally blocking is used to reduce capacity misses relying on high associativity to tackle conflict misses
- Choosing smaller blocking factor than the cache capacity can also reduce conflict misses (fewer words are active in cache)



Lam et al [1991] a blocking factor of 24 had a fifth the misses compared to a factor of 48 despite both fitting in cache

Efficiency of Compiler-Based Cache Opt.



Slide: Dave Patterson

Reducing Miss Penalty

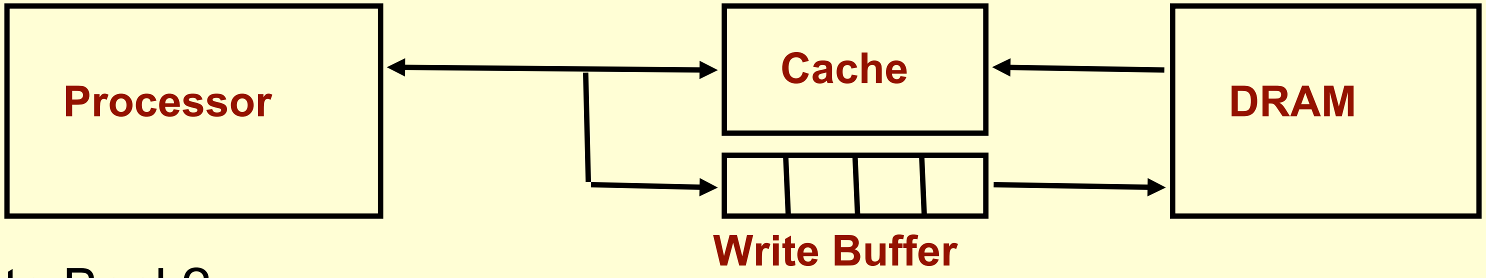
$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ \ accesses}{Instruction} \times \textbf{Miss rate} \times \boxed{Miss\ \ penalty} \right) \times Clock\ \ cycle\ \ time$$

- Reducing the miss penalty can be as effective as the reducing the miss rate
- With the gap between the processor and DRAM widening, the relative cost of the miss penalties increases over time
- Seven techniques
  1. Read priority over write on miss
  2. Sub-block placement
  3. Merging write buffer
  4. Victim cache
  5. Early Restart and Critical Word First on miss
  6. Non-blocking Caches (Hit under Miss, Miss under Miss)
  7. Second Level Cache
- Can be applied recursively to Multilevel Caches
  - Danger is that time to DRAM will grow with multiple levels in between
  - First attempts at L2 caches can make things worse, since increased worst case is worse

Slide: Dave Patterson

Read Priority over Write on Miss

- Write through with write buffers offer RAW conflicts with main memory reads on cache misses
- If simply wait for write buffer to empty, might increase read miss penalty (old MIPS 1000 by 50%)
- Check write buffer contents before read; if no conflicts, let the memory access continue



❑ Write Back?

➡ Read miss replacing dirty block

➡ Normal: Write dirty block to memory, and then do the read

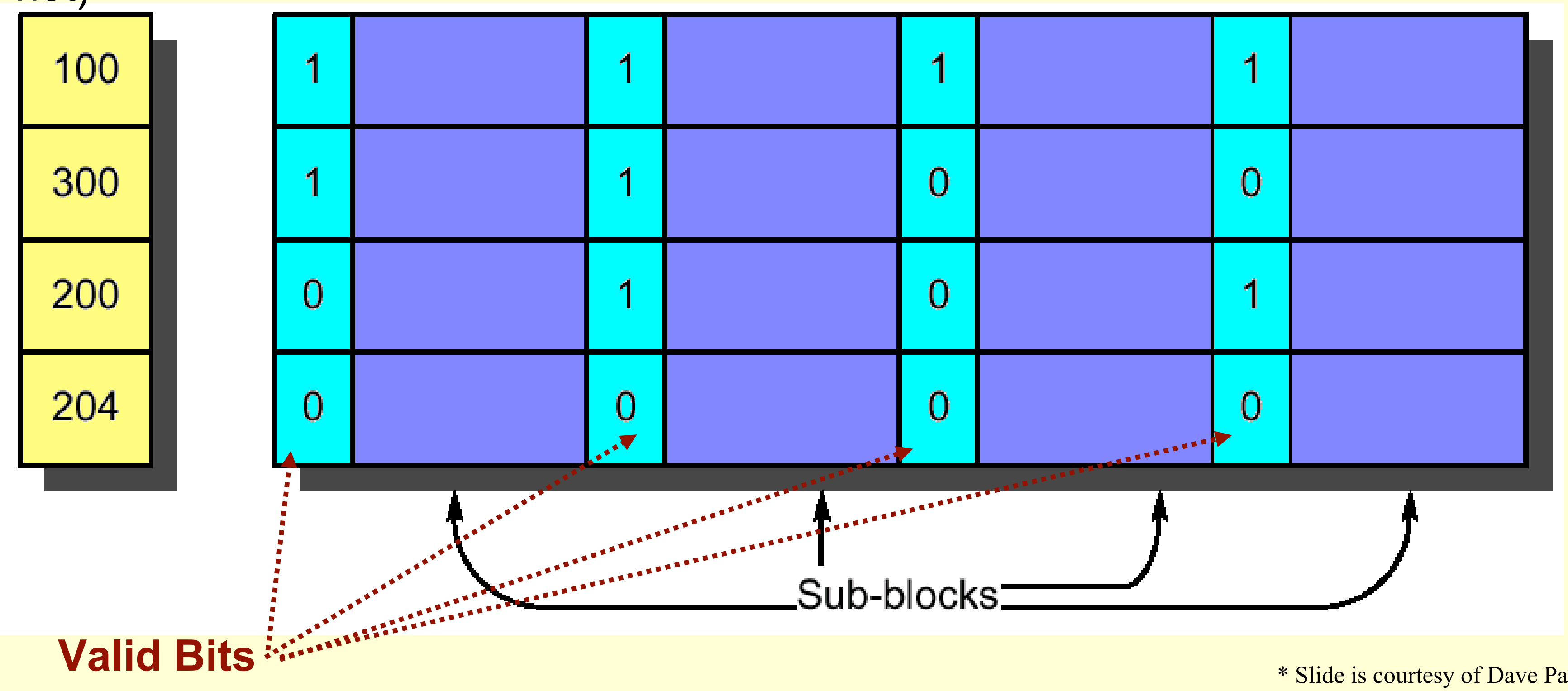
➡ Instead copy the dirty block to a write buffer, then do the read, and then do the write

➡ CPU stall less since restarts as soon as do read

* Slide is courtesy of Dave Patterson

Sub-block Placement

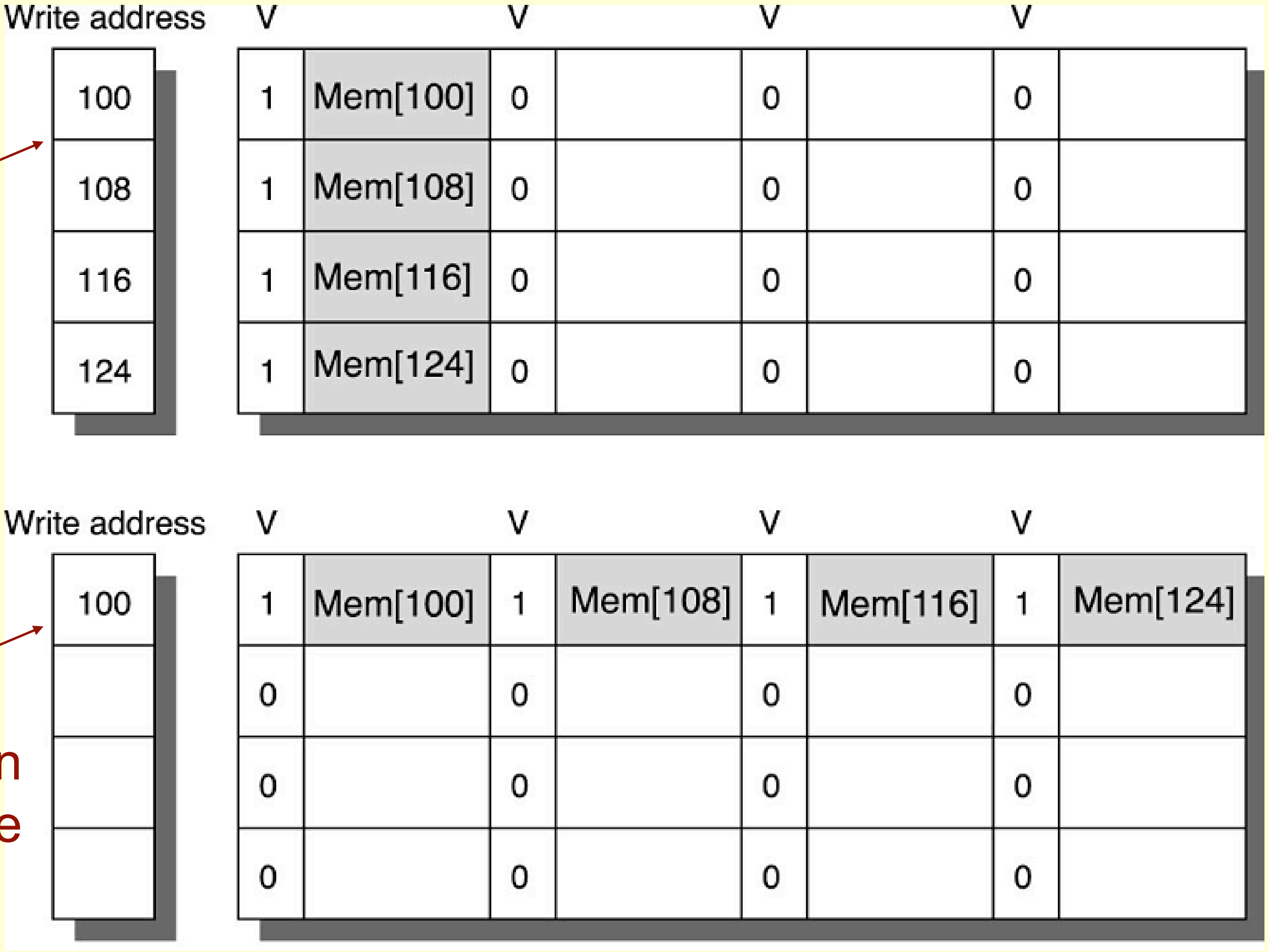
- Originally invented to reduce tag storage while avoiding the increased miss penalty caused by large block sizes
- Enlarge the block size while dividing each block into smaller units (sub-blocks) and thus does not have to load full block on a miss
- Include valid bits per sub-block to indicate the status of the sub-block (in cache or not)



100	1	1	1	1
300	1	1	0	0
200	0	1	0	1
204	0	0	0	0

* Slide is courtesy of Dave Patterson

Merging Write Buffer



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	

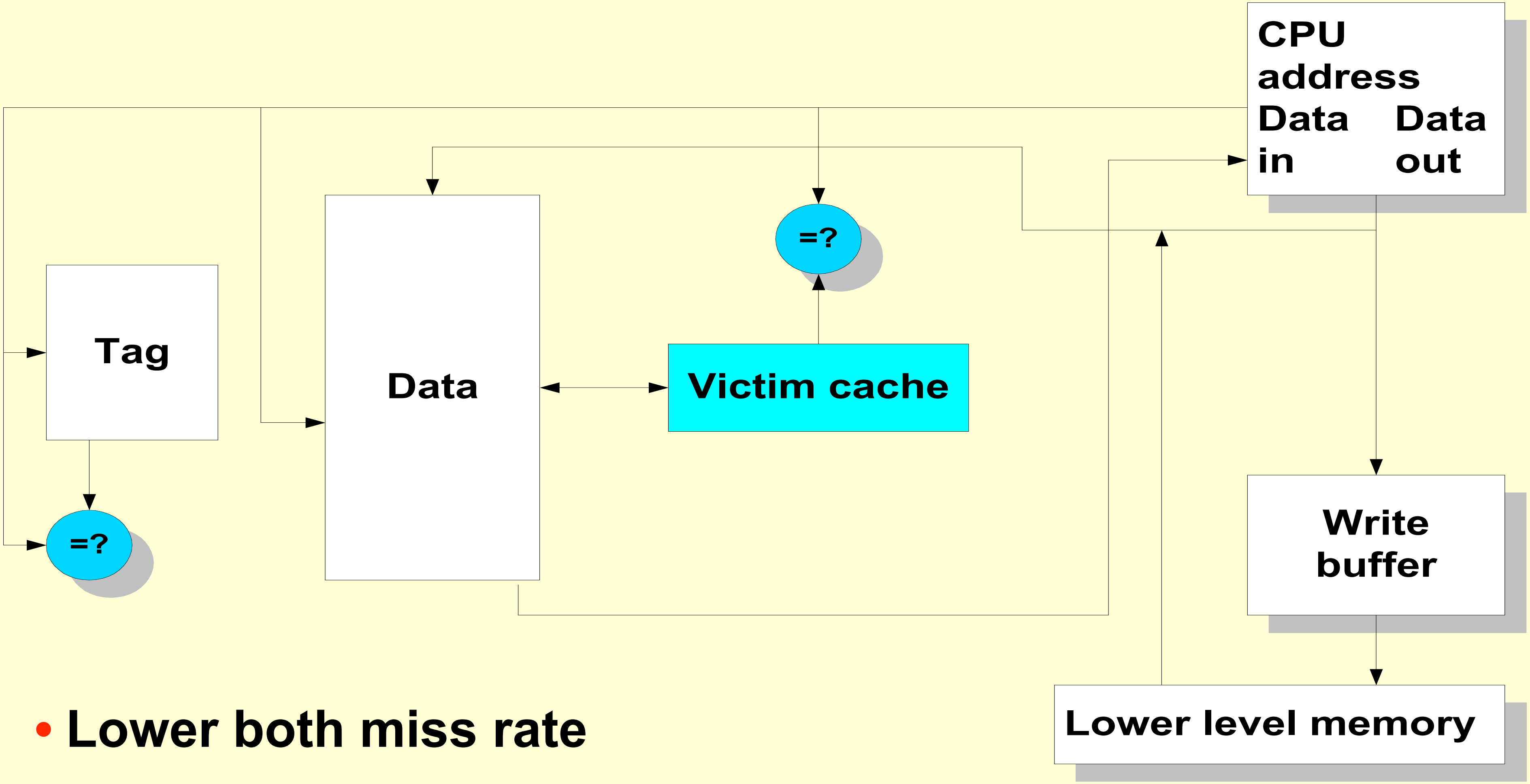
Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Buffer is full

Consolidation free up space

Extend the concept of sub-block by optimizing the write buffer handling

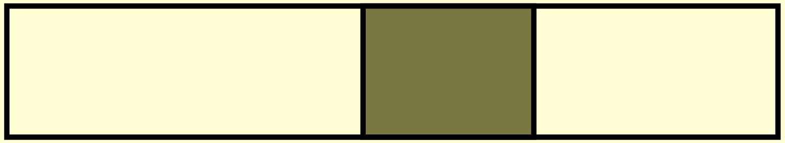
Victim Cache Approach



- Lower both miss rate
- Reduce average miss penalty
- Slightly extend the worst case miss penalty

Early Restart and Critical Word First

- Don’t wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first



block

- This technique complicates the design of the cache controller
- Early dispatching of required word is generally useful only in large blocks
- Given spatial locality program tend to want next sequential word, so not clear if benefit by early restart

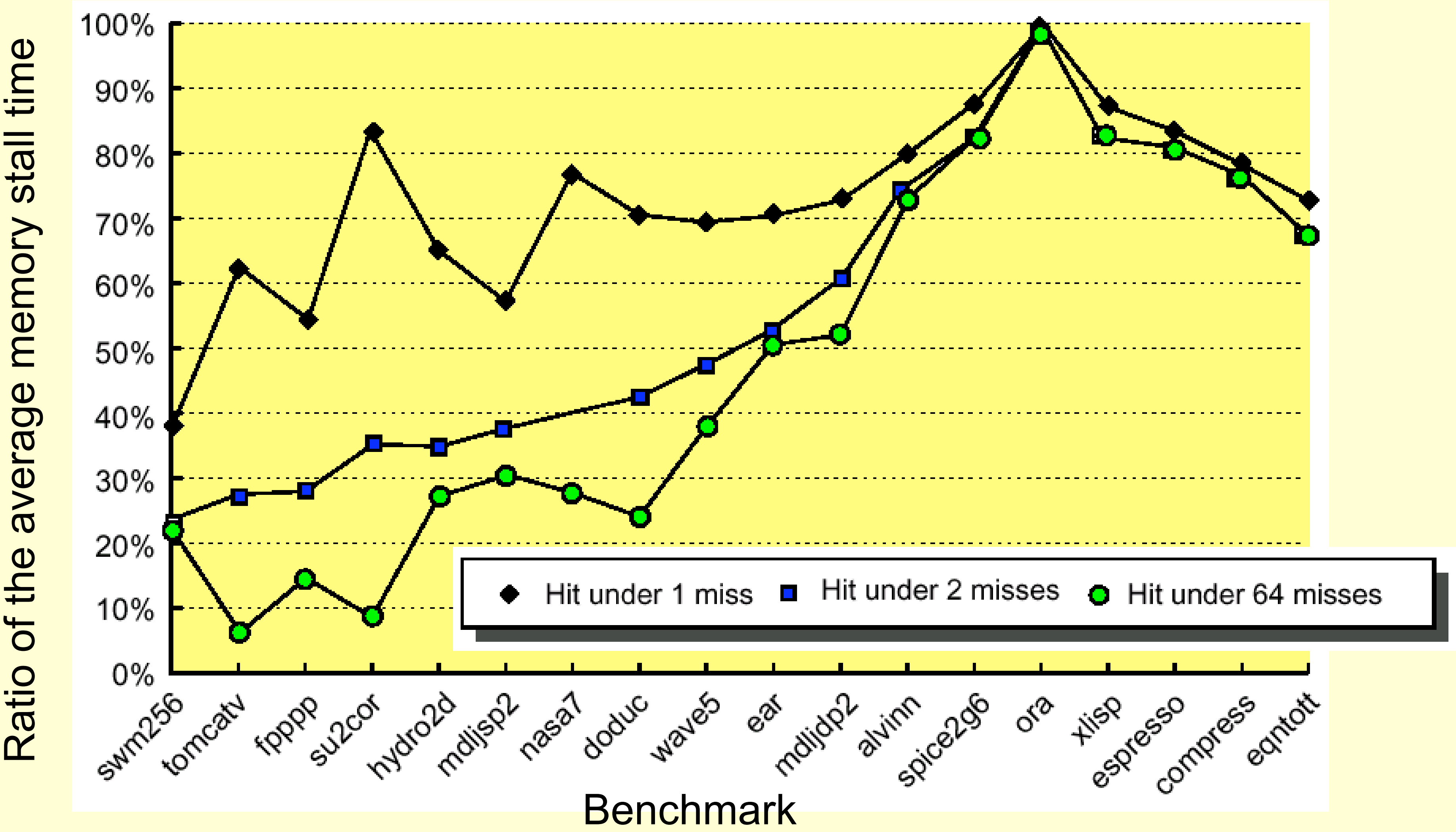
* Slide is courtesy of Dave Patterson

Non-blocking Caches

- Early restart still waits for the requested word to arrive before the CPU can continue execution
- For machines that allows out-of-order execution using a scoreboard or a Tomasulo-style control the CPU should not stall on cache misses
- “Non-blocking cache” or “lock-free cache” allows data cache to continue to supply cache hits during a miss
- “hit under miss” reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- “hit under multiple miss” or “miss under miss” may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - Requires multiple memory banks (otherwise cannot support)
  - Pentium Pro allows 4 outstanding memory misses

* Slide is courtesy of Dave Patterson

Performance of Non-blocking Caches



Second Level Cache

- The previous techniques reduce the impact of the miss penalty on the CPU while inserting a second level cache handles the cache-memory interface
- The idea of a L2 cache fits with the concept of memory hierarchy
- Measuring cache performance

Average memory access time = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Miss Penalty$_{L1}$

Miss Penalty$_{L1}$ = Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$

Average memory access time with L2 = Hit Time$_{L1}$ +

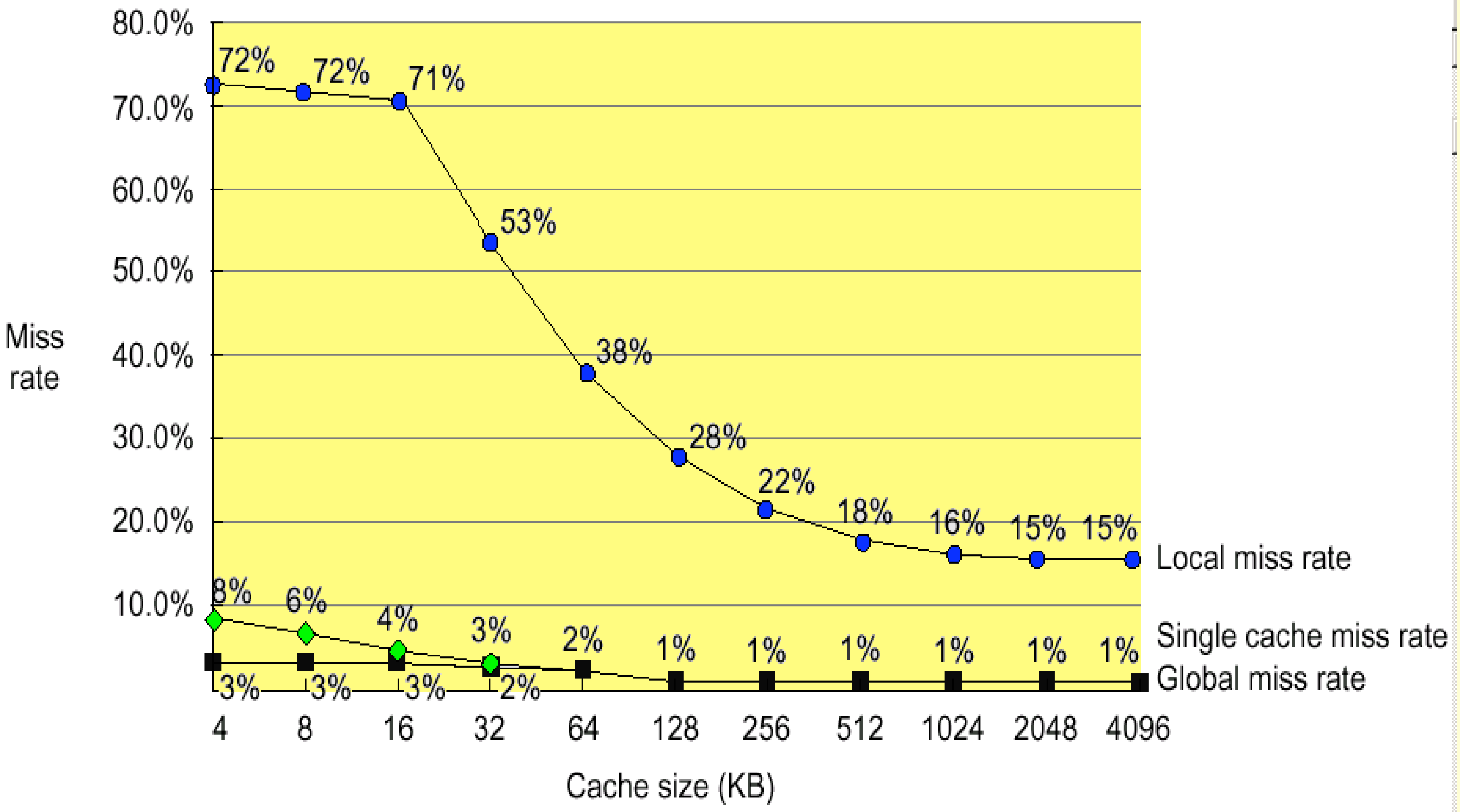
Miss Rate$_{L1}$ x (Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$)

Local miss rate— misses in this cache divided by the total number of memory accesses *to this cache* (Miss rate$_{L2}$)

Global miss rate—misses in this cache divided by the total number of memory accesses *generated by the CPU* (Miss Rate$_{L1}$ x Miss Rate$_{L2}$)

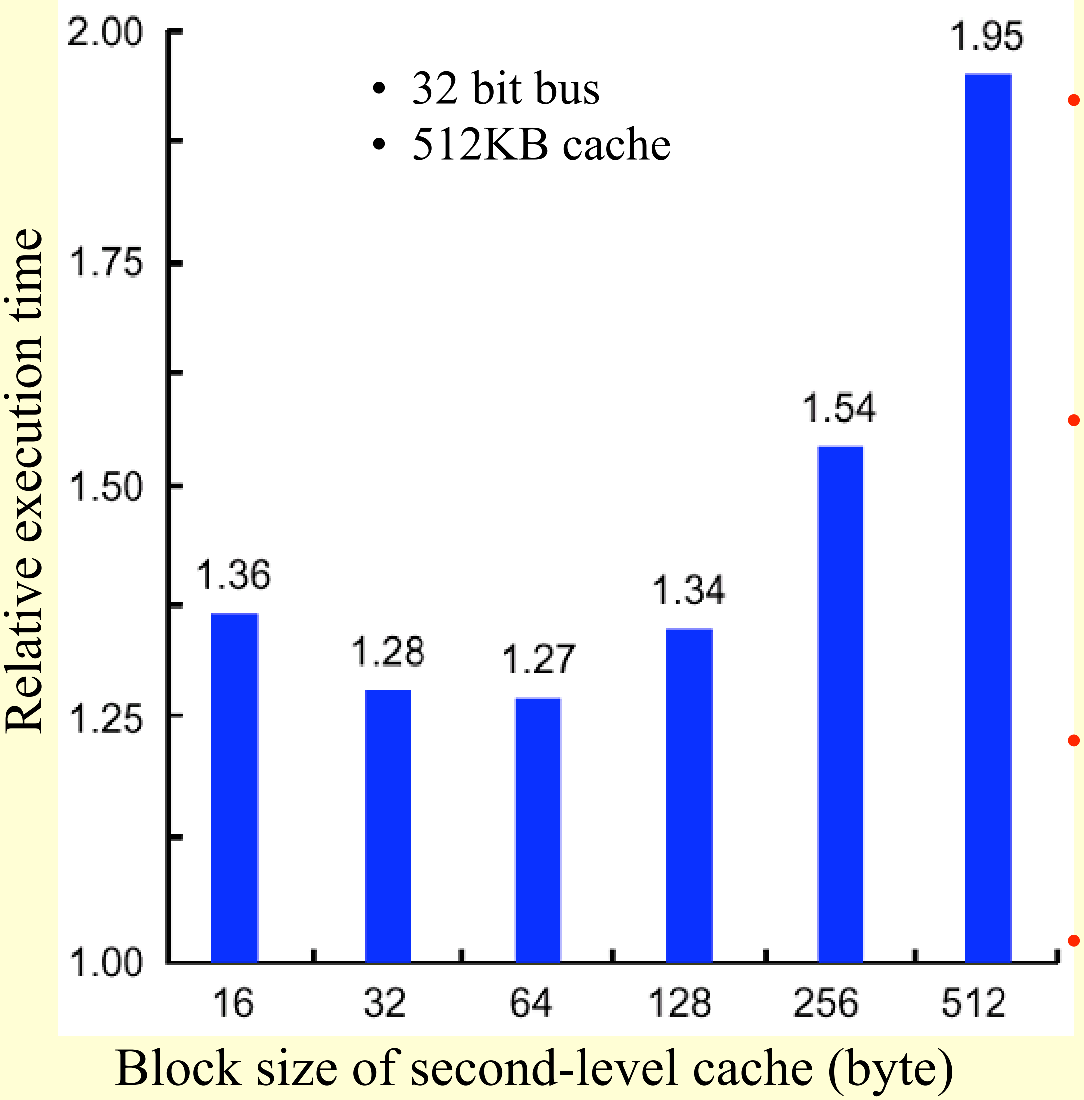
Global Miss Rate is what matters since the local miss rate is a function only of the secondary cache

Local & Global Misses



(Global miss rate close to single level cache rate provided L2 >> L1)

L2 Cache Parameters



- Since the primary cache directly affects the processor design and clock cycle, it should be kept simple and small
- The bulk of the optimization techniques can go easily to L2 cache, including large cache and block sizes, high level of associativity, etc.
- Techniques for reducing the miss rate are more practical for the L2 cache
- Considering the L2 cache can improve the L1 cache design, e.g. use write-through if L2 cache applies write-back

Reducing Hit Time

Average Access Time = Hit Time x (1 - Miss Rate) + Miss Penalty x Miss Rate

- Since hit rate is typically very high compared to miss rate, any reduction in hit time is magnified to significant gain in cache performance
- Hit time is critical because it affects the clock rate of the processor (many processors include on chip cache)
- Three techniques to reduce hit time
  1. Simple and small caches
  2. Avoid address translation during cache indexing
  3. Pipelining writes for fast write hits

Simple and small caches

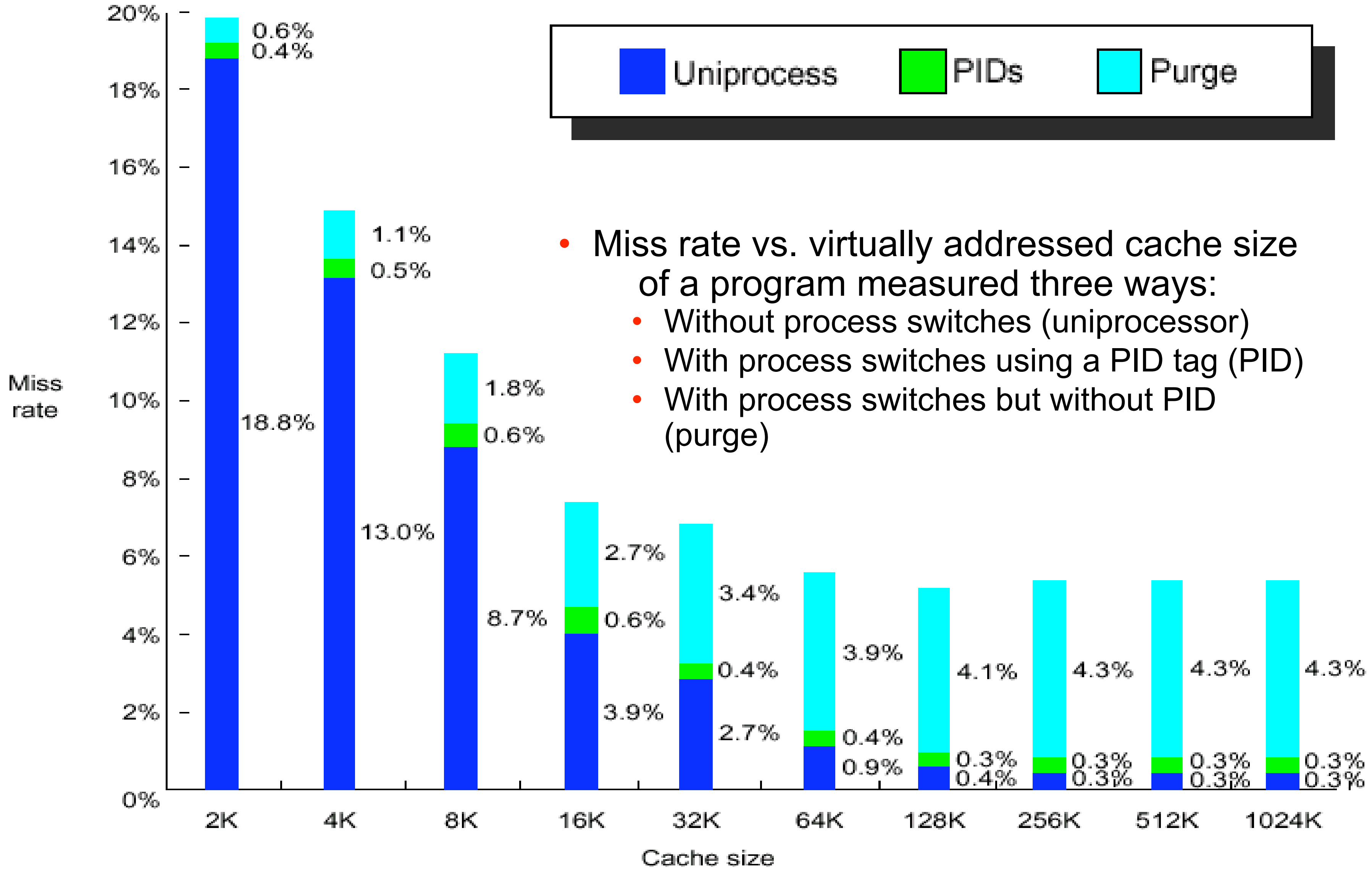
- Design simplicity limits the complexity of the control logic and allows to shorter clock cycles (e.g. direct mapped organization)
- On-chip integration decreases signal propagation delay, thus reducing hit time (small on-chip first level cache and large off-chip L2 cache)
  - Alpha 21164 has 8KB Instruction and 8KB data cache and 96KB second level cache to reduce clock rate

Avoiding Address Translation

- Send virtual address to cache? Called *Virtually Addressed Cache* or just *Virtual Cache* vs. *Physical Cache*
  - Every time process is switched logically must flush the cache; otherwise get false hits
    - Cost is time to flush + "compulsory" misses from empty cache
  - Dealing with *aliases* (sometimes called *synonyms*);
    Two different virtual addresses map to same physical address causing unnecessary read miss or even RAW problems in case user and system level processes
  - I/O must interact with cache, so forced to use virtual addresses
- Solution to aliases
  - HW guarantees that every cache block has unique physical address (simply check all cache entries)
  - SW guarantee: lower *n* bits must have same address so that it overlap with index; as long as covers index field & direct mapped, they must be unique; called *page coloring*
- Solution to cache flush
  - Add *process identifier tag* that identifies process as well as address within process: cannot get a hit if wrong process

* Slide is courtesy of Dave Patterson

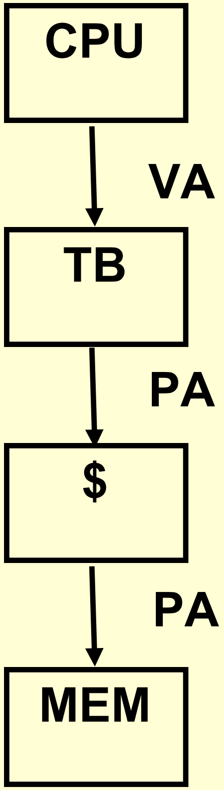
Impact of Using Process ID



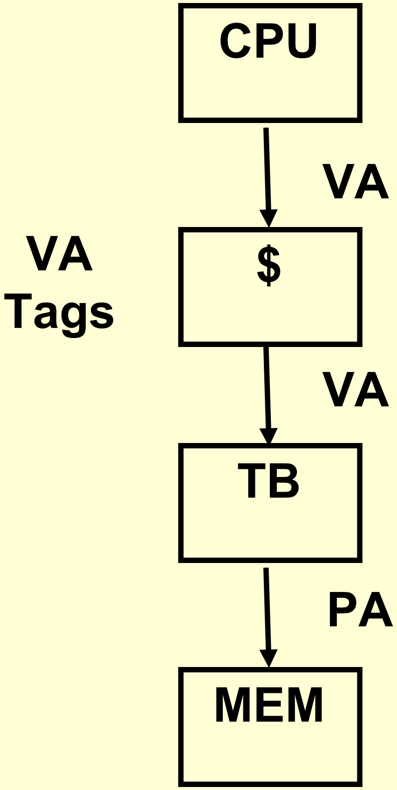
- Miss rate vs. virtually addressed cache size of a program measured three ways:
  - Without process switches (uniprocessor)
  - With process switches using a PID tag (PID)
  - With process switches but without PID (purge)

Virtually Addressed Caches

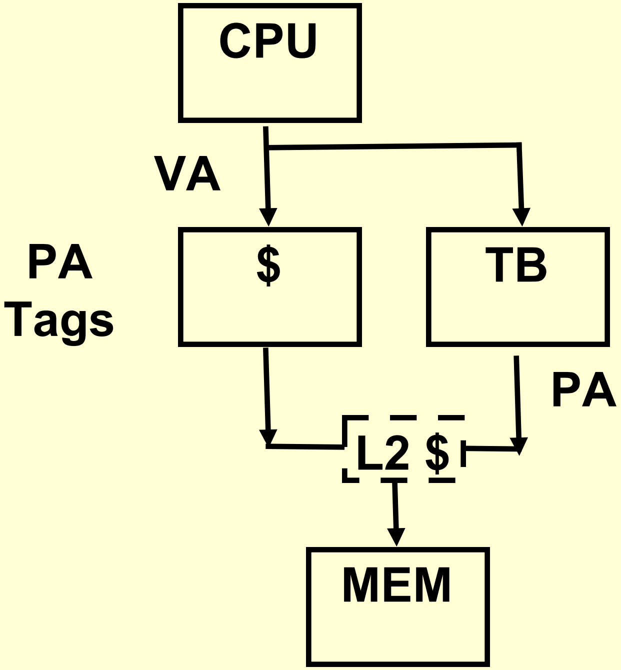
VA: Virtual address **TB**: Translation buffer **PA**: Page address



Conventional Organization



Virtually Addressed Cache
Translate only on miss
Synonym Problem

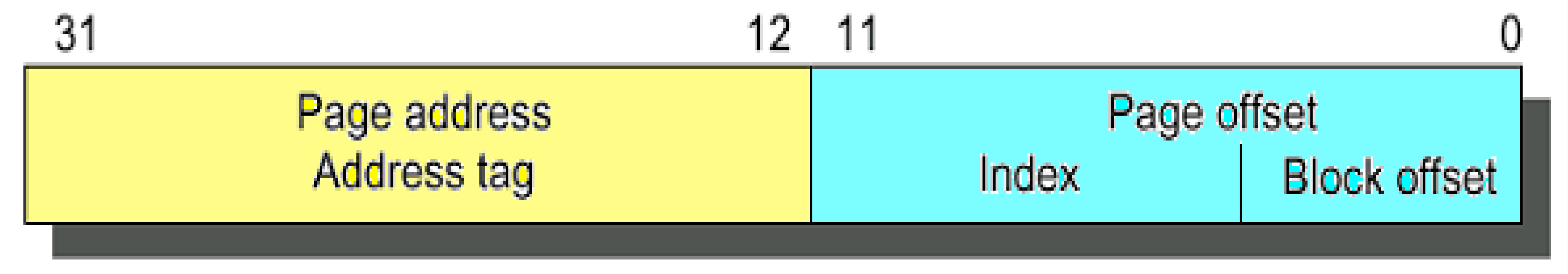


Overlap $ access with VA translation: requires $ index to remain invariant across translation

* Slide is courtesy of Dave Patterson

Indexing via Physical Addresses

- If index is physical part of address, can start tag access in parallel with translation so that can compare to physical tag
- To get the best of the physical and virtual caches is to use the page offset, which is not affected by the address translation to index the cache
- The drawback is that direct-mapped caches cannot be bigger than the page size (typically 4-KB)

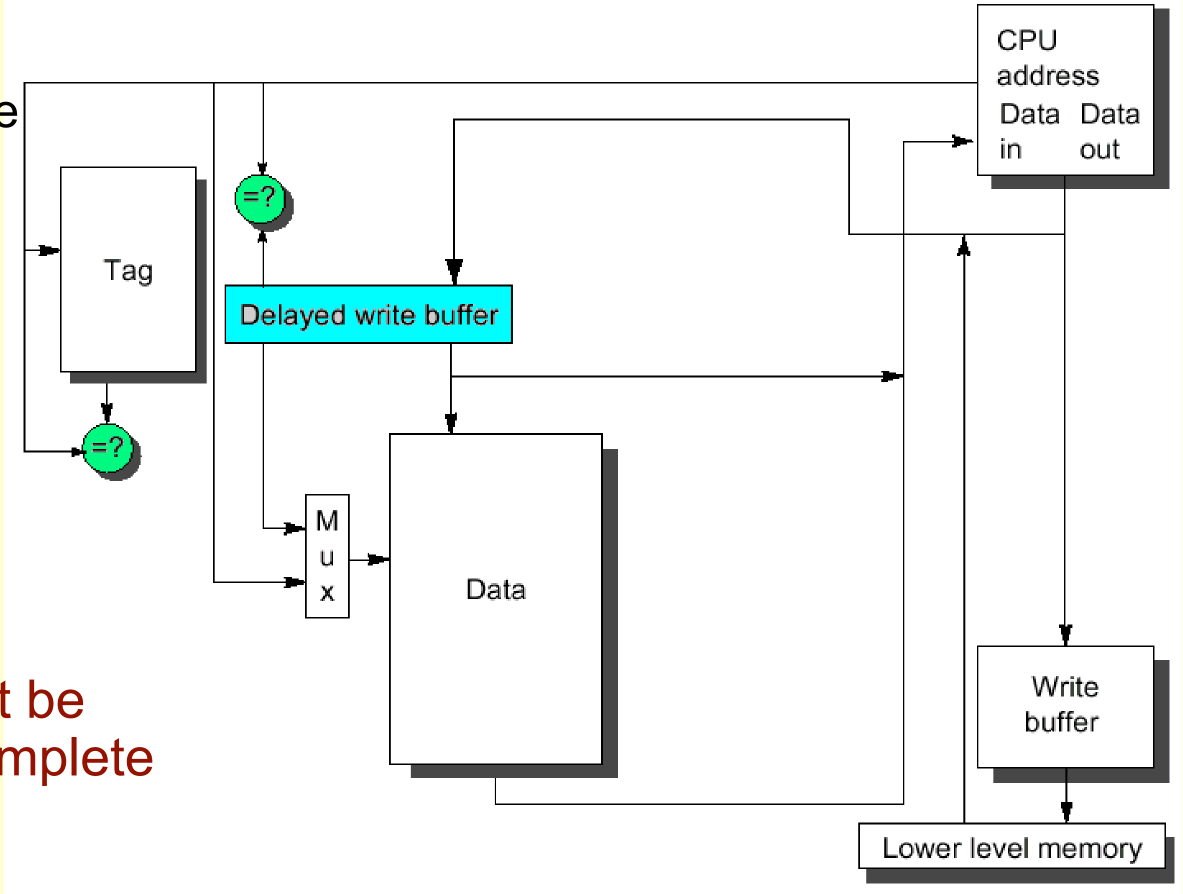


- To support bigger caches and uses same technique, one can:
  - Use higher associativity since the tag size gets smaller (moves barrier towards the most part of the address)
  - The operating system is to implement page coloring since it will fix a few least significant bits in the address (move part of the index to the tag)

Pipelined Cache Writes

- In cache read, tag check and block reading are performed in parallel while writing requires validating the tag first
  - Tag Check can be performed in parallel with a previous cache update
  - pipelined cache write

Pipeline Tag Check and Update Cache as separate stages; current write tag check & previous write cache update



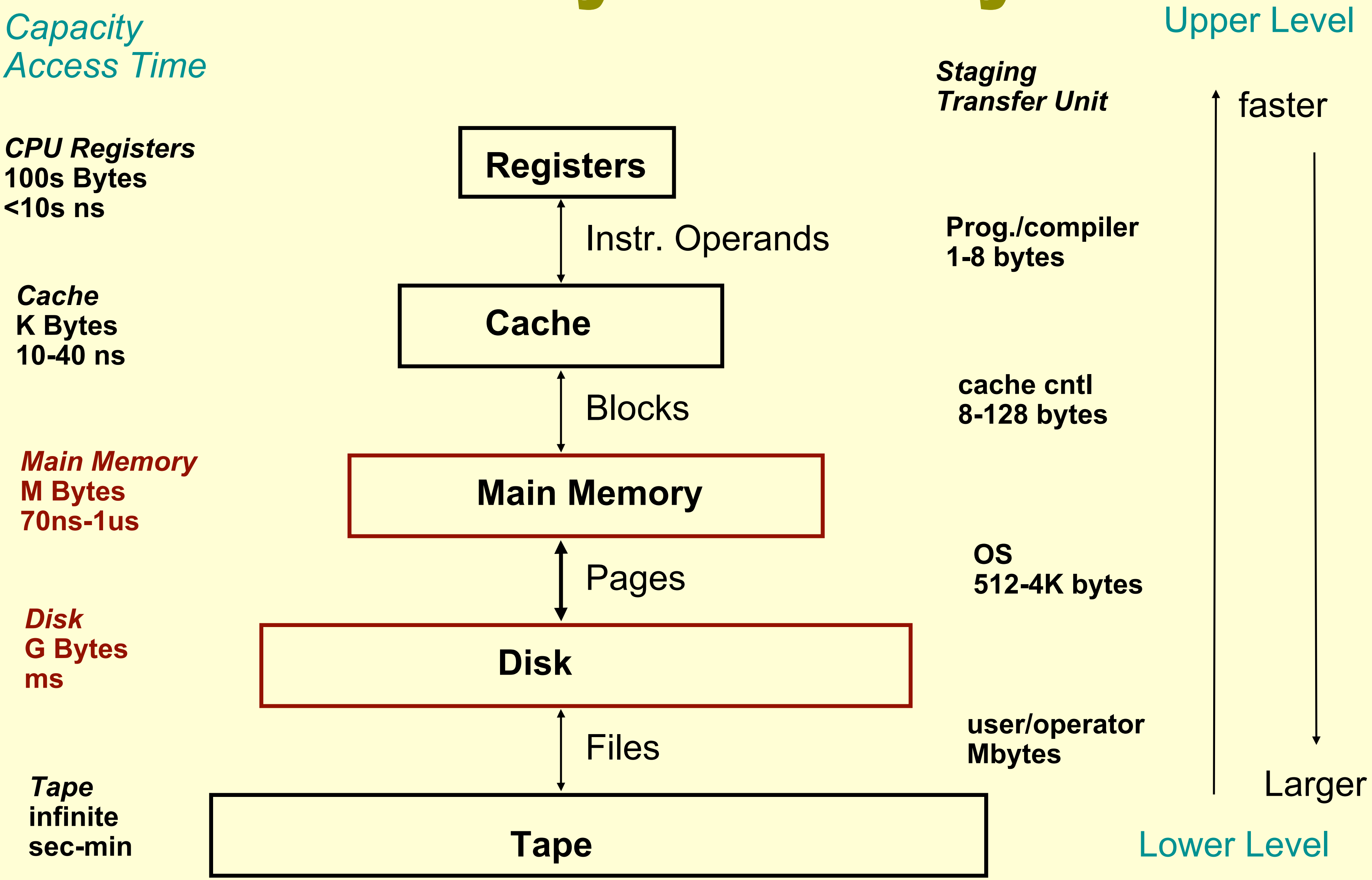
"Delayed Write Buffer"; must be checked on reads; either complete write or read from buffer

Cache Optimization Summary

	Technique	***MR***	***MP***	***HT***	***Complexity***
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Pre-fetching of Instr/Data	+			2
	Compiler Controlled Pre-fetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Sub-block Placement		+	+	1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2
hit time	Small & Simple Caches	–		+	0
	Avoiding Address Translation			+	2
	Pipelining Writes			+	1

* Slide is courtesy of Dave Patterson

Memory Hierarchy



Capacity
Access Time

Staging
Transfer Unit

Upper Level

faster

Larger

Lower Level

CPU Registers
100s Bytes
<10s ns

Registers

Instr. Operands

Prog./compiler
1-8 bytes

Cache
K Bytes
10-40 ns

Cache

Blocks

cache cntl
8-128 bytes

Main Memory
M Bytes
70ns-1us

Main Memory

Pages

OS
512-4K bytes

Disk
G Bytes
ms

Disk

Files

user/operator
Mbytes

Tape
infinite
sec-min

Tape

* Slide is courtesy of Dave Patterson

Main Memory Background

- Performance of Main Memory:
  - Latency: affects cache miss penalty
    - *Access Time*: time between request and word arrives
    - *Cycle Time*: time between requests
  - Bandwidth: primary concern for I/O & large Block Miss Penalty (L2)
- Main Memory is *DRAM*: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically (8 ms, 1% time)
  - Addresses divided into 2 halves (Memory as a 2D matrix):
    - *RAS* or *Row Access Strobe*
    - *CAS* or *Column Access Strobe*
- Cache uses *SRAM*: Static Random Access Memory
  - No refresh (6 transistors/bit vs. 1 transistor /bit, area is 10X)
  - Address not divided: Full address
- *Size*: DRAM/SRAM - *4-8*,

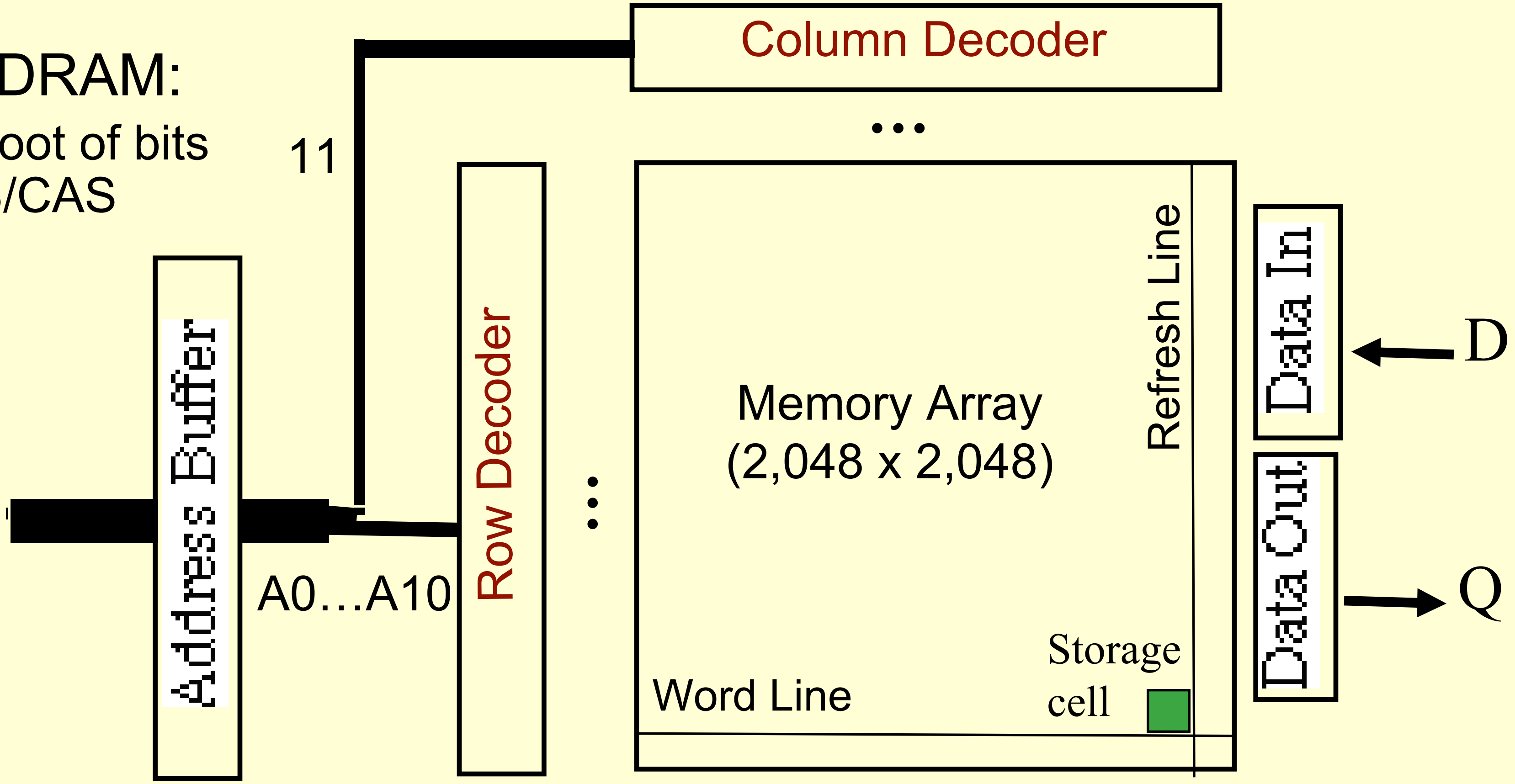
  Cost/Cycle time: SRAM/DRAM - *8-16*

* Slide is courtesy of Dave Patterson

DRAM Logical Organization

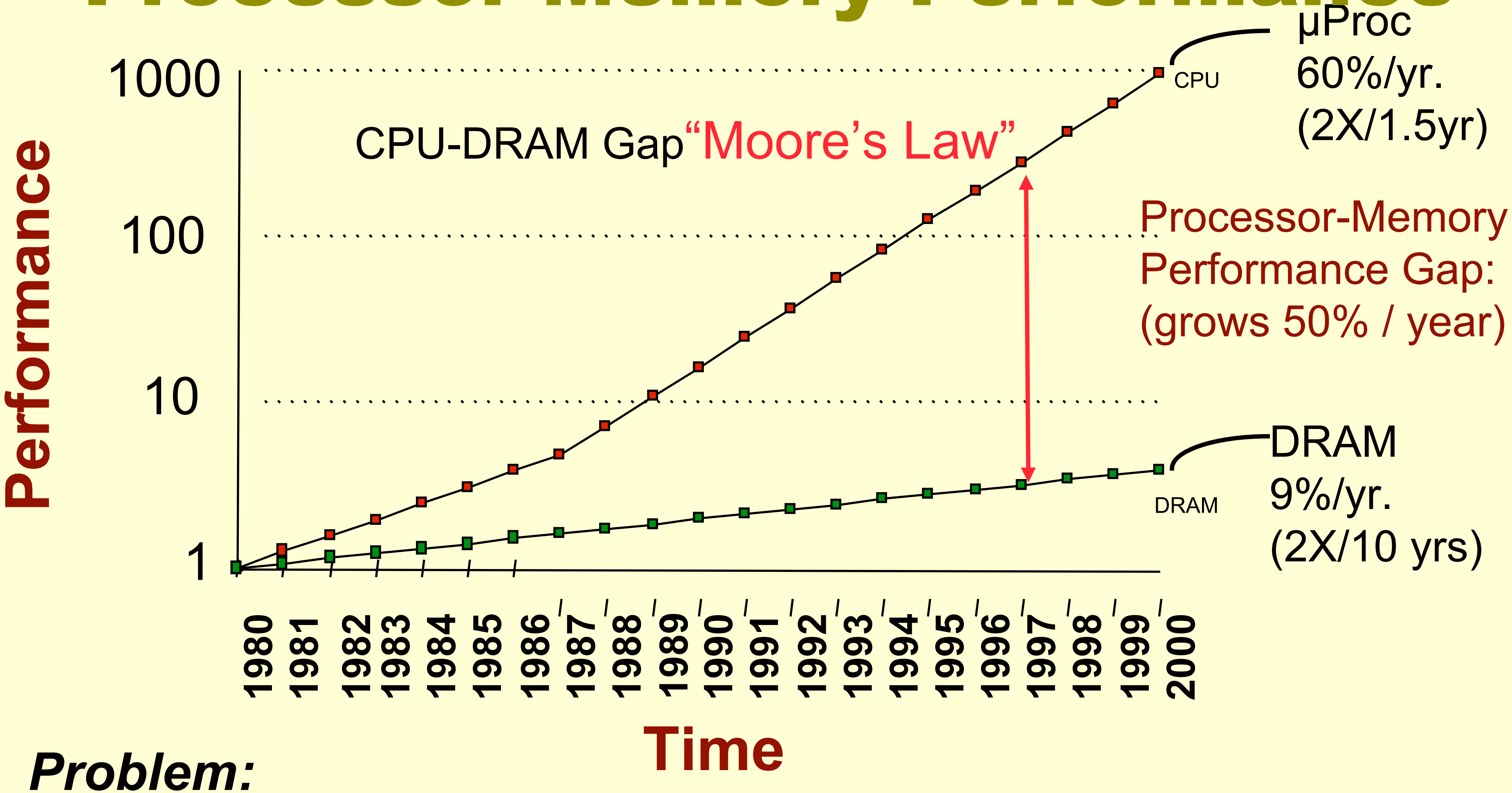
4 Mbit DRAM:

square root of bits per RAS/CAS



- Refreshing prevent access to the DRAM (typically 1-5% of the time)
- Reading one byte refreshes the entire row
- Read is destructive and thus data need to be re-written after reading
  - Cycle time is significantly larger than access time

Processor-Memory Performance



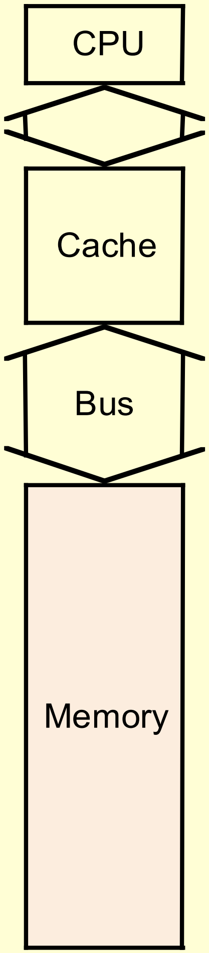
Problem:

Improvements in access time are not enough to catch up

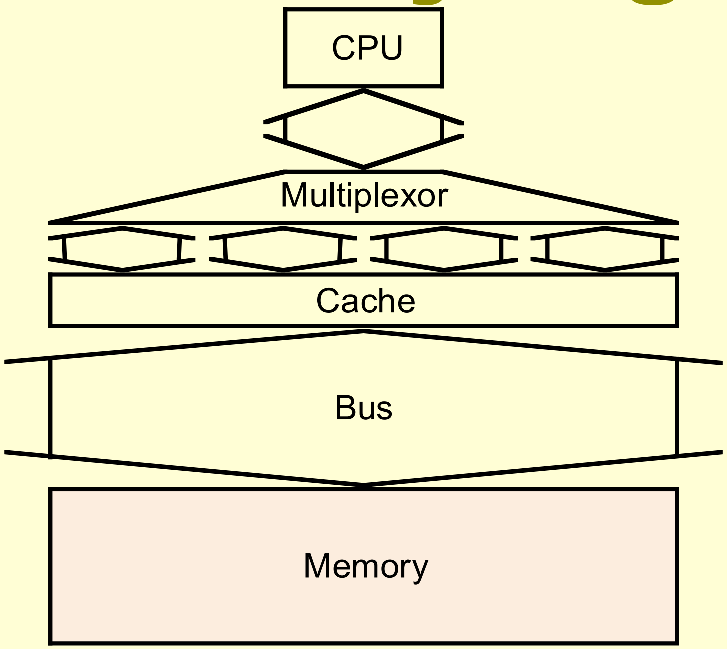
Solution:

Increase the bandwidth of main memory (improve throughput)

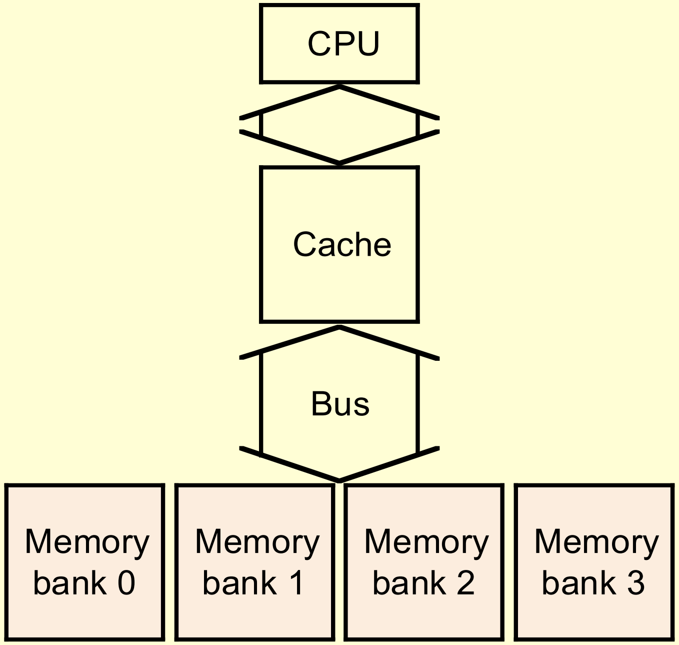
Memory Organization



a. One-word-wide memory organization



b. Wide memory organization



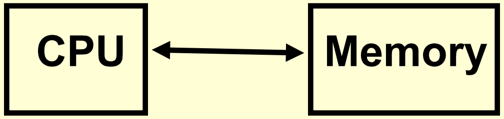
c. Interleaved memory organization

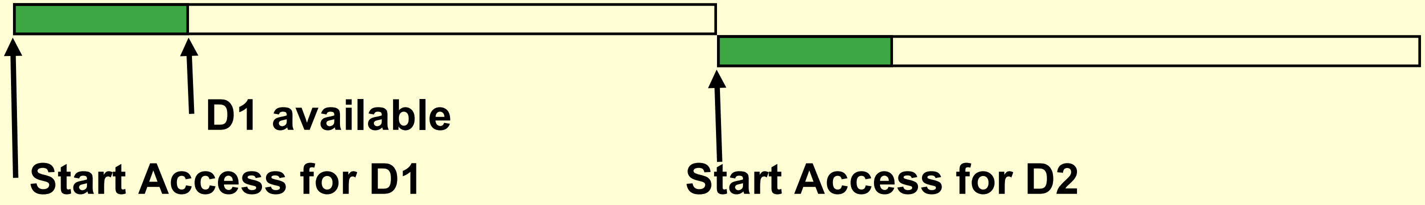
- *Simple*: CPU, Cache, Bus, Memory same width (32 bits)
- *Wide*: CPU/Mux 1 word; Mux/Cache, Bus, Memory N words
- *Interleaved*: CPU, Cache, Bus 1 word: Memory N Modules (4 Modules); example is *word interleaved*

Memory organization would have significant effect on bandwidth

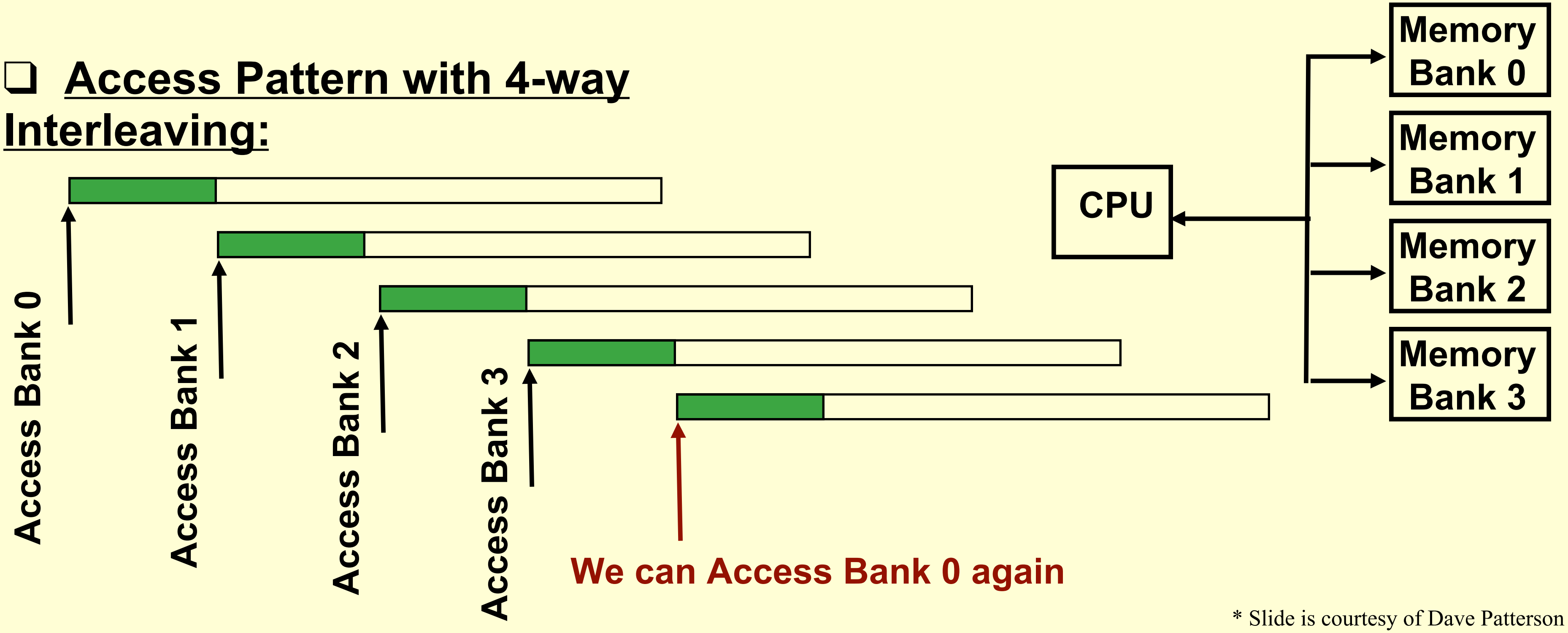
Memory Interleaving

- **Access Pattern without Interleaving:**





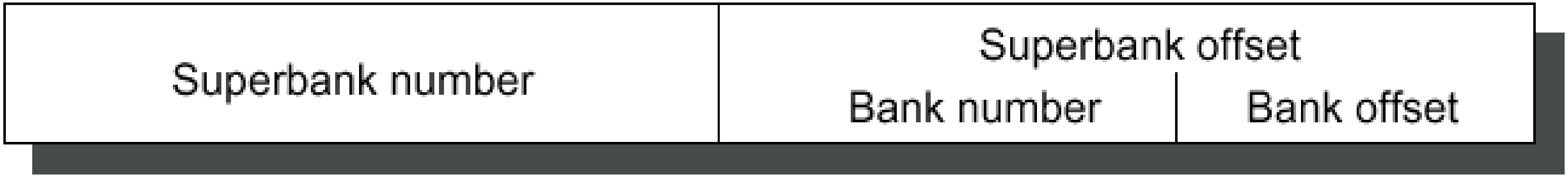
- **Access Pattern with 4-way Interleaving:**



* Slide is courtesy of Dave Patterson

Independent Memory Banks

- Original motivation for memory banks is higher bandwidth by interleaving sequential access using one memory controller and one data bus
- Memory banks that allows multiple independent accesses are useful for:
  - Multiprocessor system: allowing concurrent execution
  - I/O: limiting memory access contention and expedite data transfer
  - CPU with Hit under *n* Misses, Non-blocking Cache
- Supporting multiple independent accesses requires separate controller, address bus and possibly data buses for each bank



Superbank number	Superbank offset	
	Bank number	Bank offset

- *Superbank*: all memory active on one block transfer
- *Bank*: portion within a superbank that is word interleaved (or *Subbank*)

Superbanks act as separate memories mapped to the same address space

* Slide is courtesy of Dave Patterson

Avoiding Bank Conflicts

- The effectiveness of interleaving depends on the frequency that independent requests will go to different banks
- Sequential requests and accesses that differ by an odd number would work well with interleaving

Example: Assuming 128 banks

```
int x[256][512];
 for (j = 0; j < 512; j = j+1)
   for (i = 0; i < 256; i = i+1)
     x[i][j] = 2 * x[i][j];
```

- Bank number = address MOD number of banks
- Address within bank = address / number of words in bank
- Since 512 is multiple of 128, all elements of a column will be in the same bank and code will stall on data cache misses

Solutions

- SW: loop interchange or declaring array not power of 2 (“array padding”)
- HW: Prime number of banks and modulo interleaving
  - Complexity of modulo & divide per memory access with prime no. banks?
  - Simple address calculation using the *Chinese Remainder Theorem*

* Slide is courtesy of Dave Patterson

Chinese Remainder Theorem

- As long as two sets of integers ai and bi follow these rules:
  - $b_i = x \bmod a_i$, $0 \le b_i < a_i$
  - $0 \le x < a_0 \times a_1 \times a_2 \times \ldots$
  - a_i and a_j are co-prime with $i \neq j$
- then the integer x has only one solution for each pair of integers a_i and b_i
- ds in bank

Fast Bank Number

- Modulo interleaving
  - Bank num = b_0, Num banks = a_0
    - $0 \le$ bank num < num banks ($0 \le b_0 < a_0$)
  - Address within bank = b_1, Num words in bank = a_1
    - $0 \le$ Address in bank < bank size ($0 \le b_1 < a_1$)
    - Addr < num banks × bank size ($0 \le x < a_0 \times a_1$)
  - Num banks (a_0) and bank size (a_1) are co-prime
    - e.g. a_0 prime and a_1 a power of 2

Example

- Bank number = address MOD number of banks
- Address within bank = address MOD number words in bank
- Bank number = b_0, number of banks = a_0 (= 3 in example)
- Address within bank = b_1, number of words in bank = a_1 (= 8 in example)

	Seq. Interleaved			Modulo Interleaved		
Bank Number:	0	1	2	0	1	2
Address within Bank:						
0	0	1	2	0	16	8
1	3	4	5	9	1	17
2	6	7	8	18	10	2
3	9	10	11	3	19	11
4	12	13	14	12	4	20
5	15	16	17	21	13	5
6	18	19	20	6	22	14
7	21	22	23	15	7	23

Unambiguous mapping with simple bank addressing

DRAM-Specific Optimization

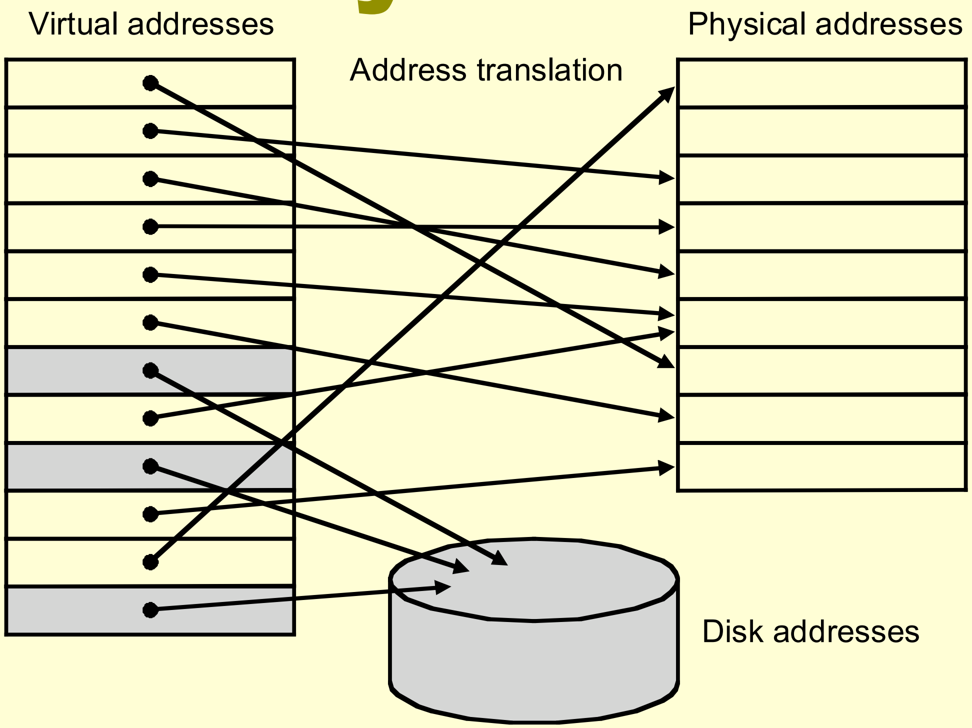
- DRAM Access Interleaving
  - DRAM must buffer a row of bits internally for the column access
    - Performance can be improved by allowing repeated access to buffer without another row access time (requires minimal additional cost)
  - Nibble mode: DRAM supplies 3 extra bits from sequential locations for every row access strobe (internal 4-way interleaving)
  - Page mode: The buffer acts like a SRAM allowing bit access from the buffer until a row change or a refresh
  - Static column: Similar to page mode but does not require change in CAS to access another bit from the buffer
  - DRAM optimization has been shown to give up to 4x speedup

DRAM-Specific Optimization

- Bus-based DRAM (RAMBUS)
  - Each chip act as a module with an internal bus replacing CAS and RAS
  - Allows for other access to take place while between the sending the address and returning the data
  - Each module performs its own refresh
  - Performance can reach 1 byte / 2 ns (500 MB/s per chip)
  - Expensive compared to the traditional DRAM

Virtual Memory

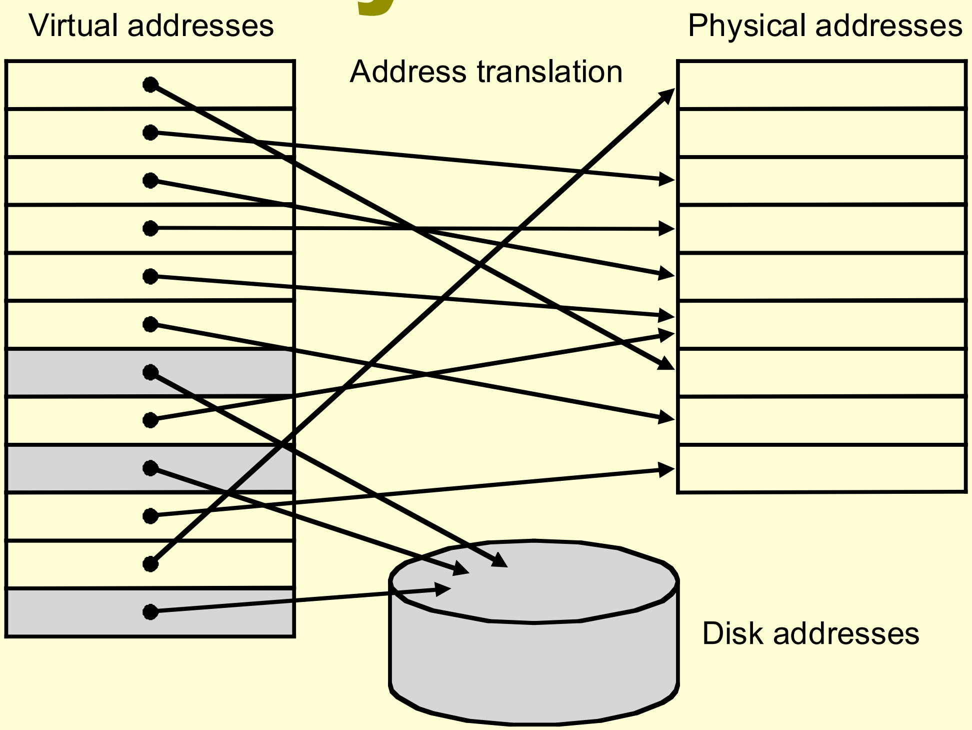
- Using virtual addressing, main memory plays the role of cache for disks
- The virtual space is much larger than the physical memory space
- Physical main memory contains only the active portion of the virtual space
- Address space can be divided into fixed size (pages) or variable size (segments) blocks



Cache		Virtual memory
Block	⇒	Page
Cache miss	⇒	page fault
Block addressing	⇒	Address translation

Virtual Memory

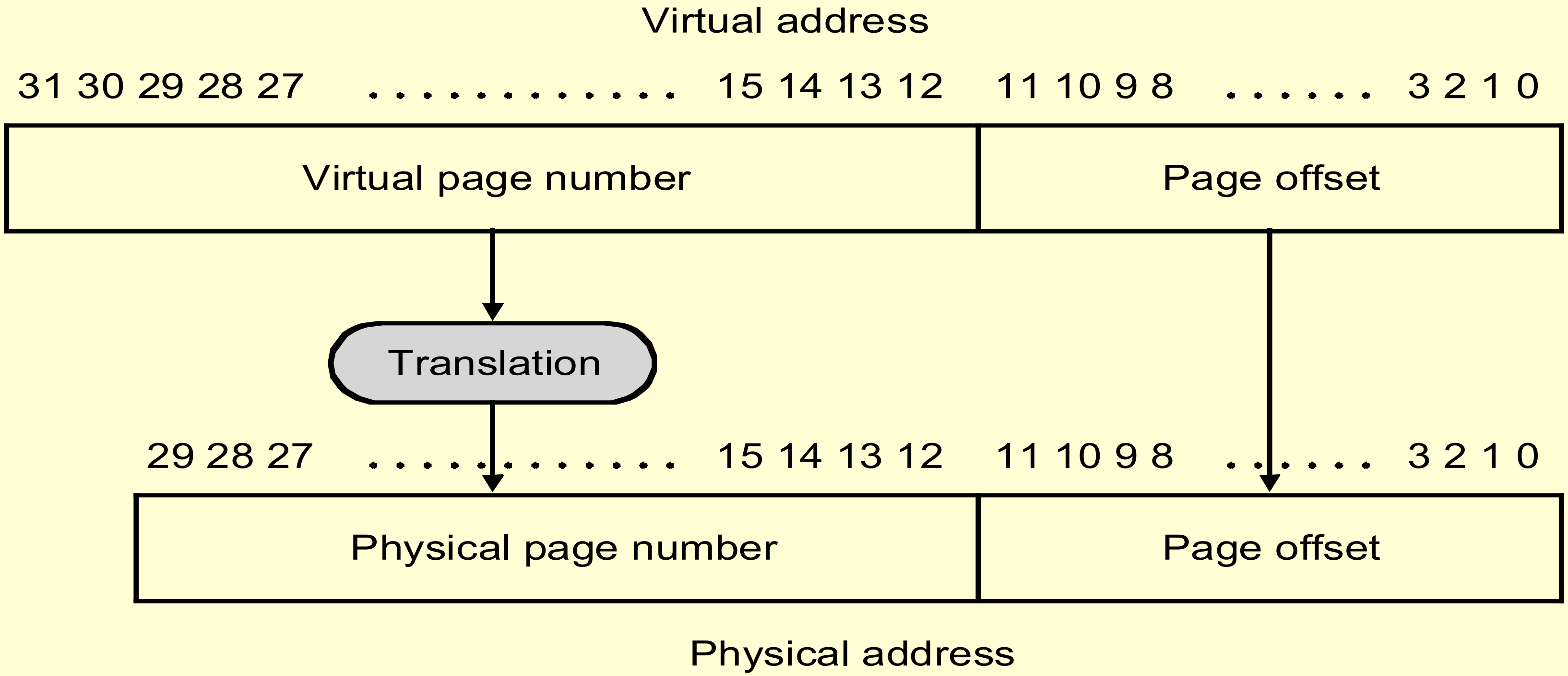
- Allows efficient and safe data sharing of memory among multiple programs
- Moves programming burdens of a small, limited amount of main memory
- Simplifies program loading and avoid the need for contiguous memory block allows programs to be loaded at any physical memory location



Cache		Virtual memory
Block	⇒	Page
Cache miss	⇒	page fault
Block addressing	⇒	Address translation

Virtual Addressing

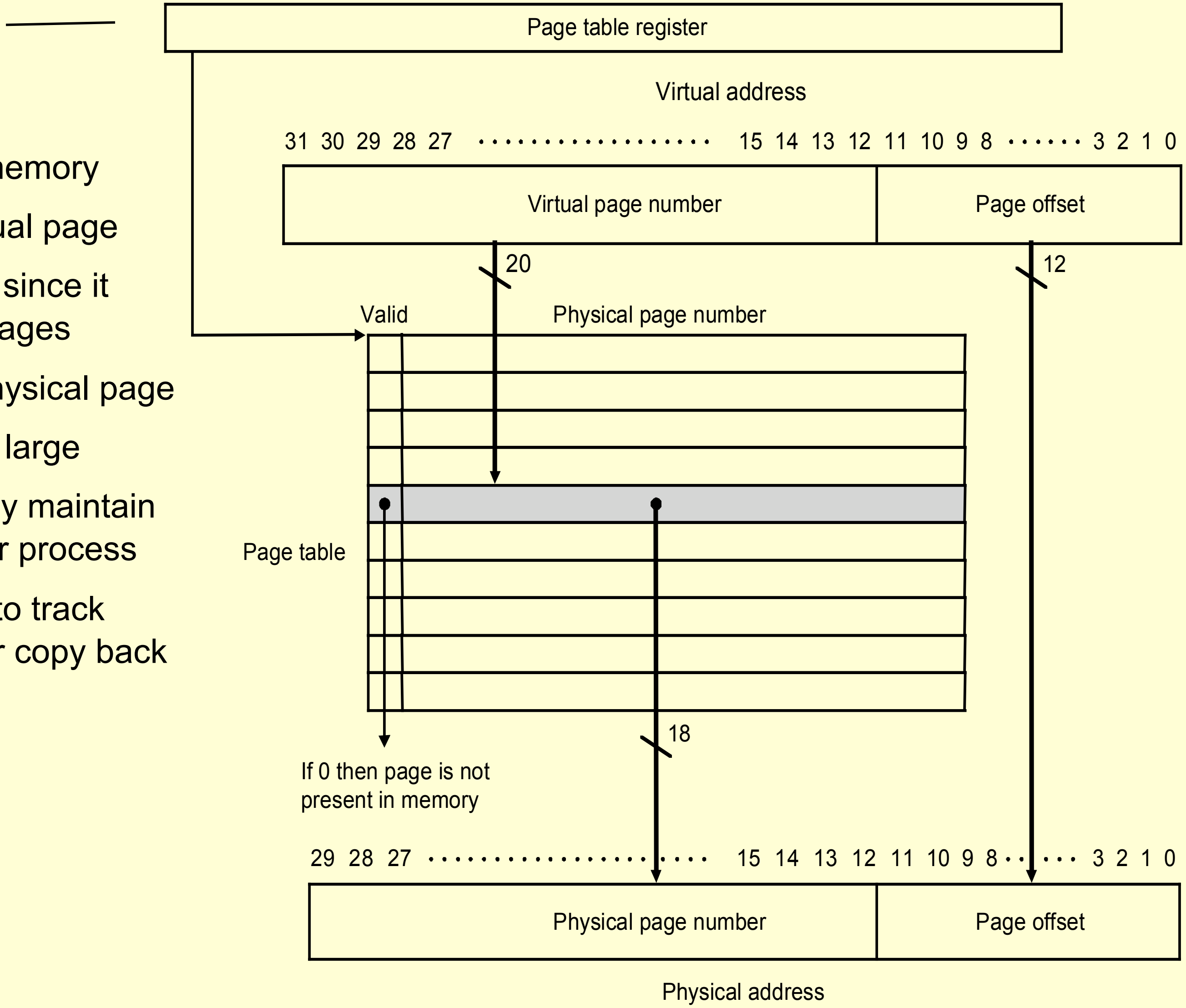
- Page faults are costly and take millions of cycles to process (disks are slow)
- Optimization Strategies:
  - Pages should be large enough to amortize the high access time
  - Fully associative placement of pages reduces page fault rate
  - Software-based handling of page faults allows using clever page placement
  - Write-through technique can make writing very time consuming (use copy back)



Page Table

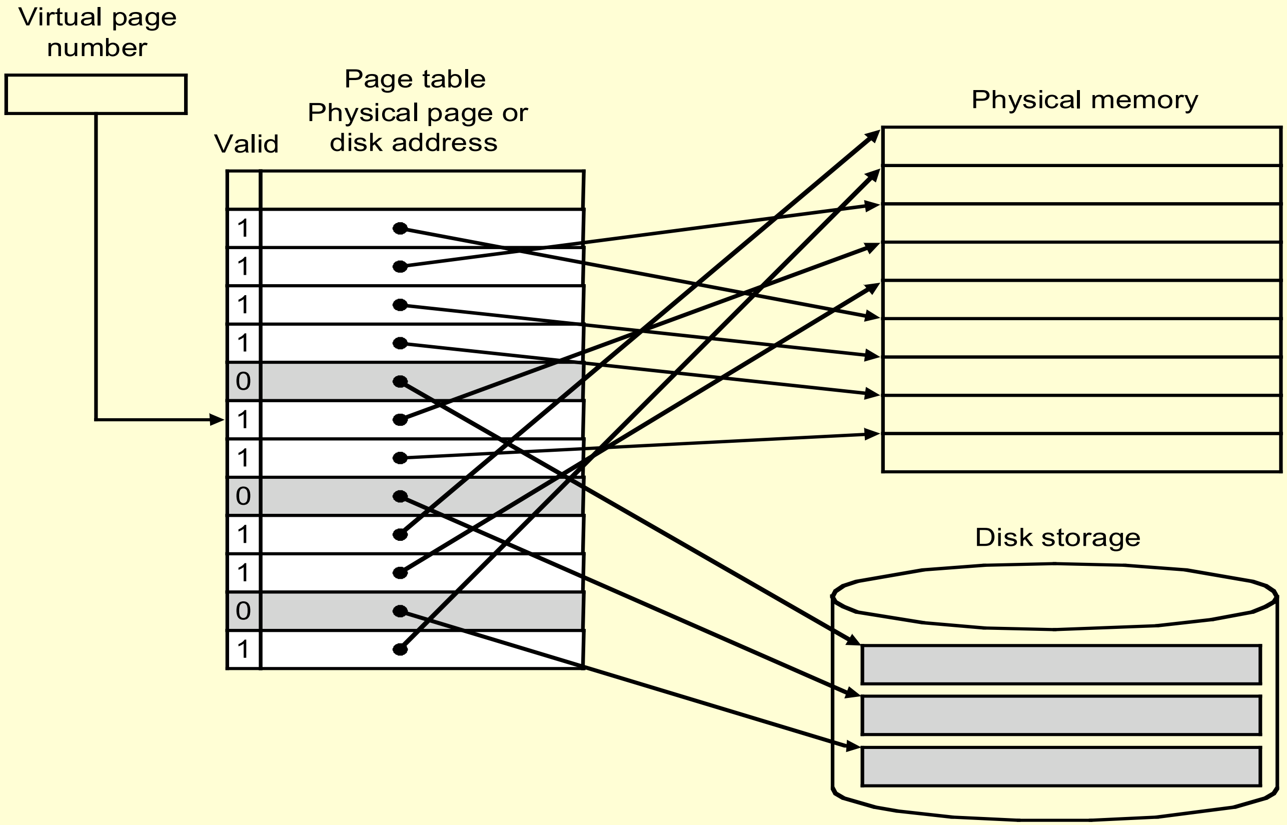
Hardware supported

- Page table
  - Resides in main memory
  - One entry per virtual page
  - No tag is required since it covers all virtual pages
  - Point directly to physical page
  - Table can be very large
  - Operating sys. may maintain one page table per process
  - A dirty bit is used to track modified pages for copy back



Page Faults

- A page fault happen when the valid bit of a virtual page is off
- A page fault generates an exception to be handled by the operating system to bring the page to main memory from a disk
- The operating system creates space for all pages on disk and keeps track of the location of pages in main memory and disk
- Page location on disk can be stored in page table or in an auxiliary structure



- LRU page replacement strategy is the most commonly used
- Simplest LRU implementation uses a reference bit per page and periodically reset reference bits

Optimizing Page Table Size

With a 32-bit virtual address, 4-KB pages, and 4 bytes per page table entry:

$$\text{Number of page table entries} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{Size of page table} = 2^{20} \text{ page table entries} \,□\, 2^{2} \frac{\text{bytes}}{\text{page table entry}} = 4\,\text{MB}$$

Optimization techniques:

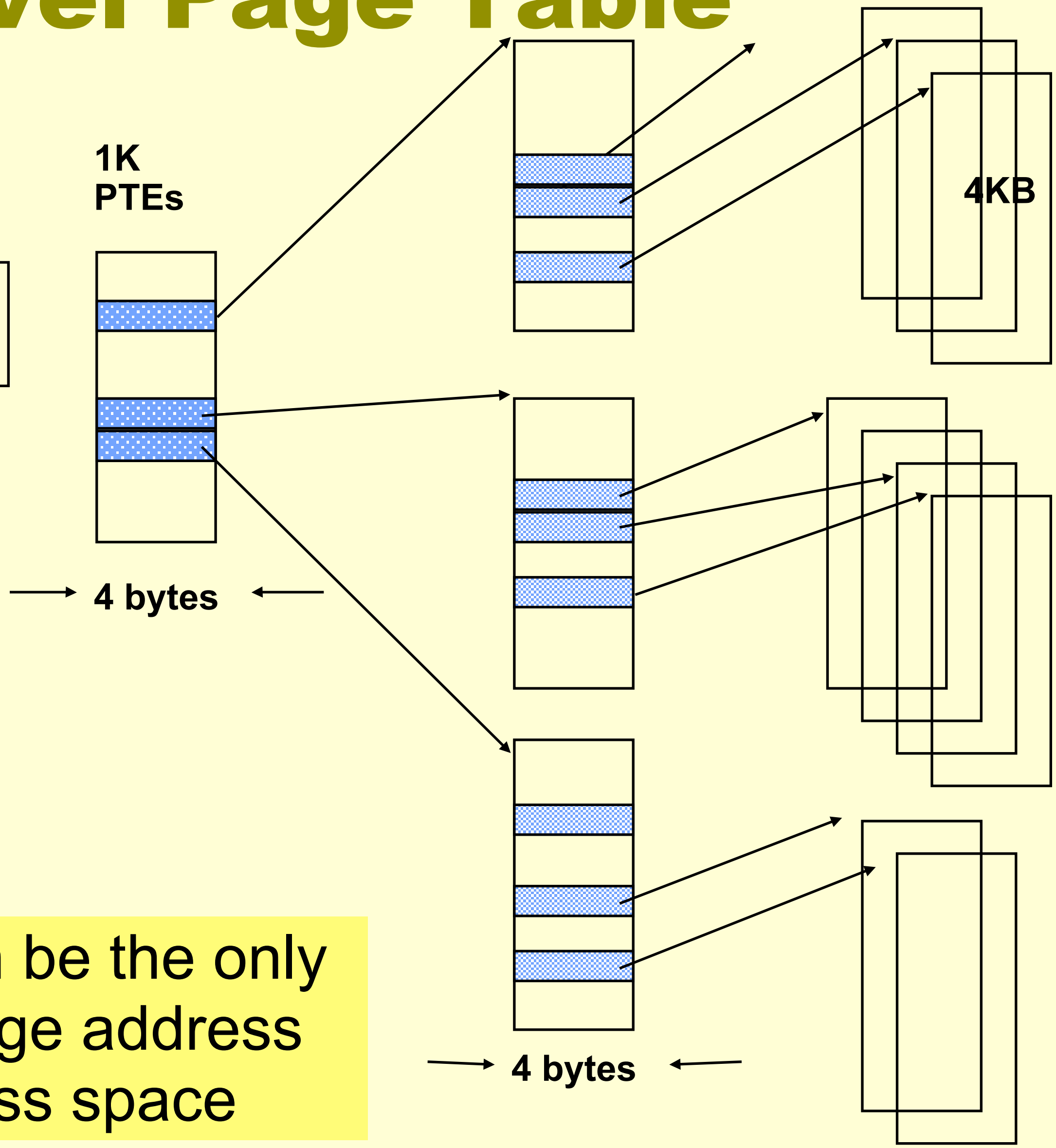
- Keep bound registers to limit the size of page table for given process in order to avoid empty slots
- Store only physical pages and apply hashing function of the virtual address (inverted page table)
- Use multi-level page table to limit size of the table residing in main memory
- Allow paging of the page table, i.e. apply virtual addressing recursively
- Cache the most used pages □ Translation Look-aside Buffer

Multi-Level Page Table

32-bit address:

10	10	12
P1 index	P2 index	page offest

- 2 GB virtual address space
- 4 MB of PTE2
  - paged, holes
- 4 KB of PTE1

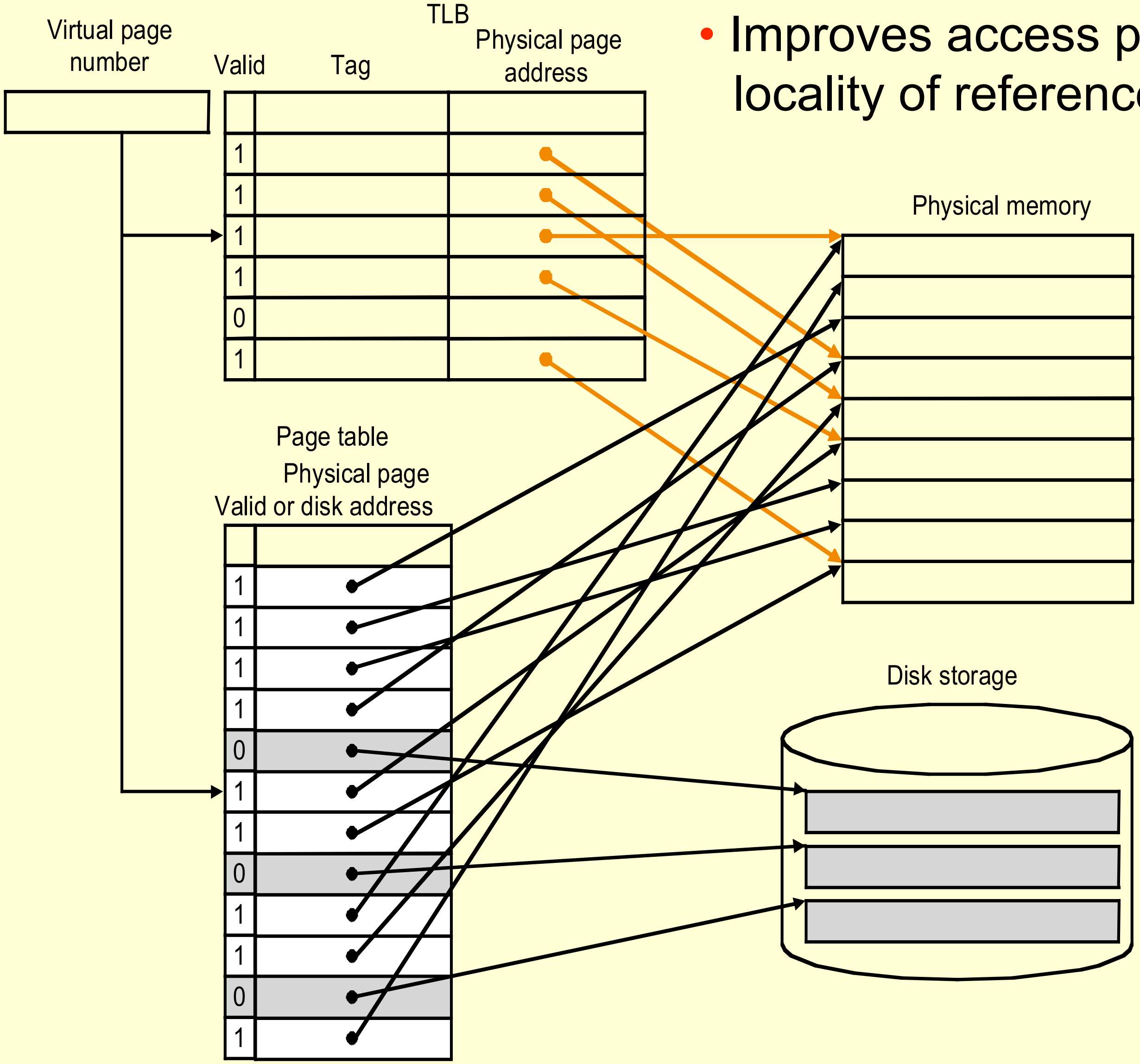


Inverted page table can be the only practical solution for huge address space, e.g 64-bit address space

* Slide is courtesy of Dave Patterson

Translation Look-aside Buffer

- Special cache that keeps track of recently used translation
- Improves access performance relying on locality of reference principle



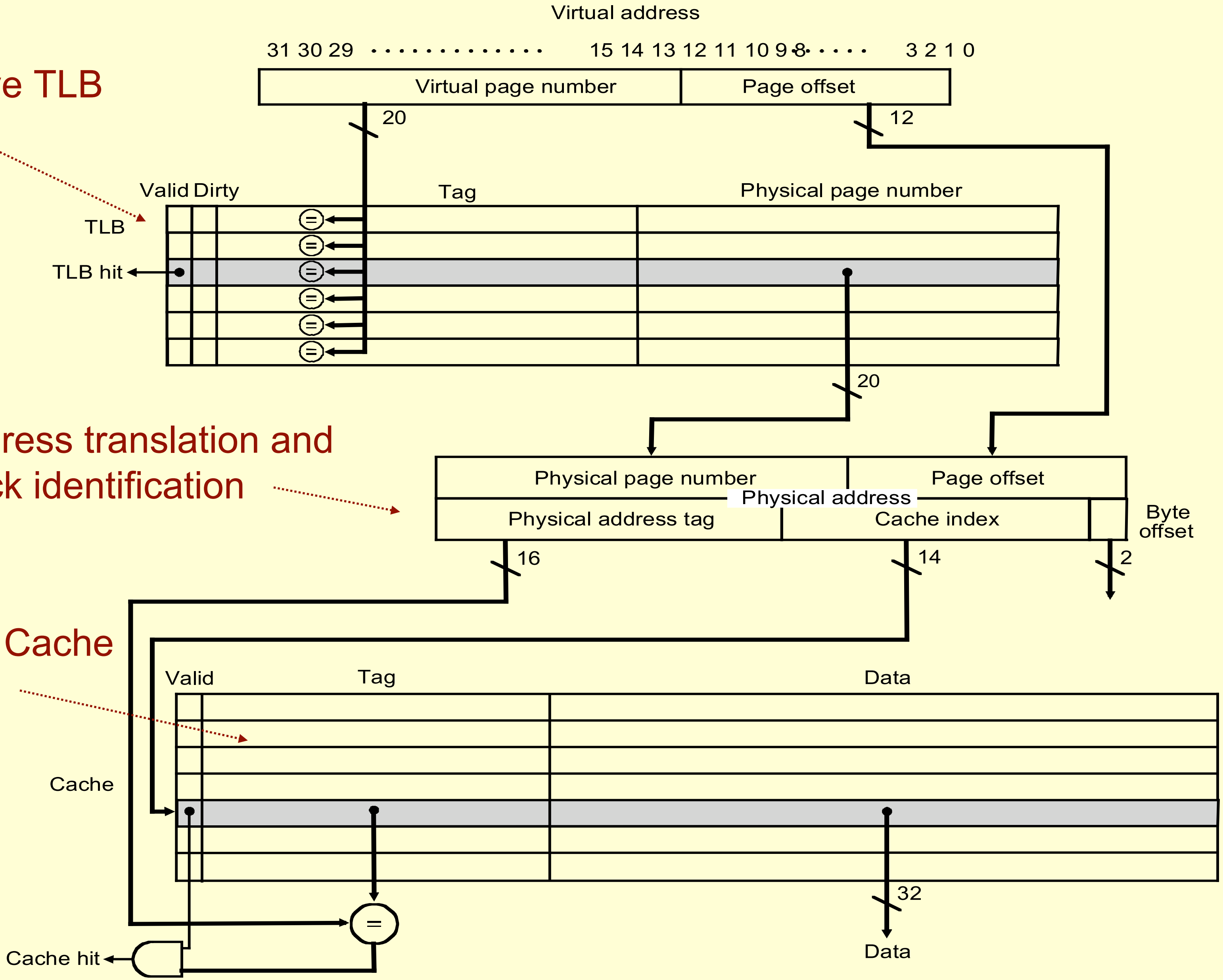
- TLB misses are exceptions that are typically handled by the operating system
- Simple replacement strategy is applied to TLB misses since it happens frequently

TLB and Cache in MIPS

Fully associative TLB

Address translation and block identification

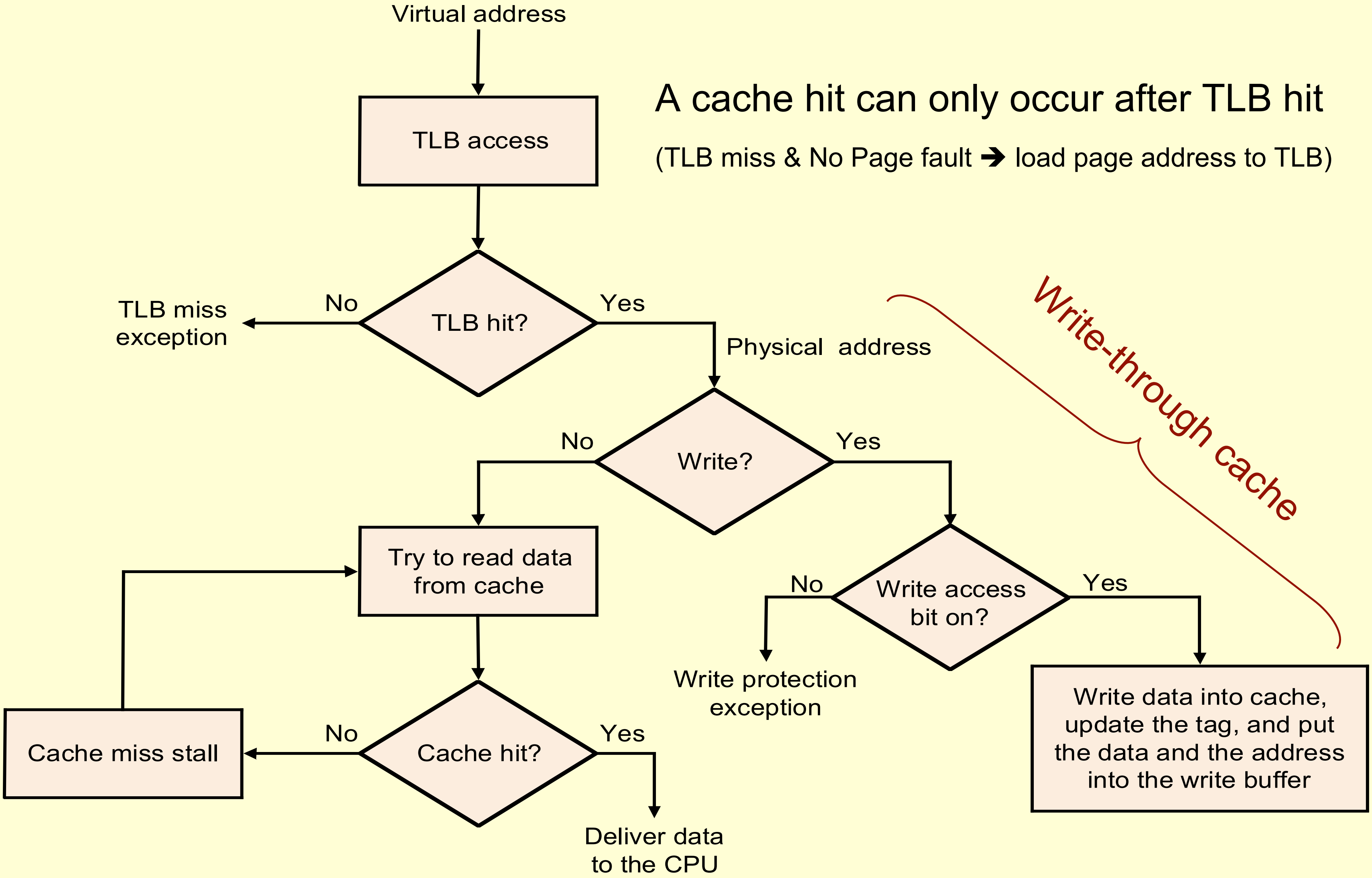
Direct-mapped Cache



TLB and Cache in MIPS

A cache hit can only occur after TLB hit

(TLB miss & No Page fault ➔ load page address to TLB)



Virtual address

TLB access

TLB hit?

No

TLB miss exception

Yes

Physical address

Write?

No

Try to read data from cache

Cache hit?

No

Cache miss stall

Yes

Deliver data to the CPU

Yes

Write access bit on?

No

Write protection exception

Yes

Write data into cache, update the tag, and put the data and the address into the write buffer

Write-through cache

Memory Related Exceptions

Possible exceptions:

Cache miss: referenced block not in cache and needs to be fetched from main memory

TLB miss: referenced page of virtual address needs to be checked in the page table

Page fault: referenced page is not in main memory and needs to be copied from disk

Cache	TLB	Page fault	Possible? If so, under what condition
miss	hit	hit	Possible, although the page table is never really checked if TLB hits
hit	miss	hit	TLB misses, but entry found in page table and data found in cache
miss	miss	hit	TLB misses, but entry found in page table and data misses in cache
miss	miss	miss	TLB misses and followed by page fault. Data must miss in cache
miss	hit	miss	Impossible: cannot have a translation in TLB if page is not in memory
hit	hit	miss	Impossible: cannot have a translation in TLB if page is not in memory
hit	miss	miss	Impossible: data is not allowed in cache if page is not in memory

Memory Protection

- It is always desirable to prevent a process from corrupting allocated memory space of other processes
- The processor must support processes in a non-privileged mode to avoid messing up memory protection
- Implementation can be by mapping independent virtual pages to separate physical pages
- Write protection bits would be included in the page table for authentication
- Sharing pages can be facilitated by the operating system through mapping virtual pages of different processes to same physical pages
- To enable the operating system to implement protection, the hardware must provide at least the following capabilities:
  - Support at least two mode of operations, one of them is a user mode
  - Provide a portion of CPU state that a user process can read but not write, e.g. page pointer and TLB
  - Enable change of operation modes through special instructions

Handling TLB Misses & Page Faults

- <u>**TLB Miss**</u>: *(hardware-based handling)*
  - Check if the page is in memory (valid bit) → update the TLB
  - Generate page fault exception if page is not in memory
- <u>**Page Fault**</u>: *(handled by operating system)*
  - Transfer control to the operating system
  - Save processor status: registers, program counter, page table pointer, etc.
  - Lookup the page table and find the location of the reference page on disk
  - Choose a physical page to host the referenced page, if the candidate physical page is modified (dirty bit is set) the page needs to be written back
  - Start reading the referenced page from disk to the assigned physical page
- The processor needs to support "restarting" instructions in order to guarantee correct execution
  - The user process causing the page fault will be suspended by the operating system until the page is readily available in main memory
  - Protection violations are handled by the operating system similarly but without automatic instruction restarting