CMSC 611: Advanced Computer Architecture

Cache and Memory

Classification of Cache Misses

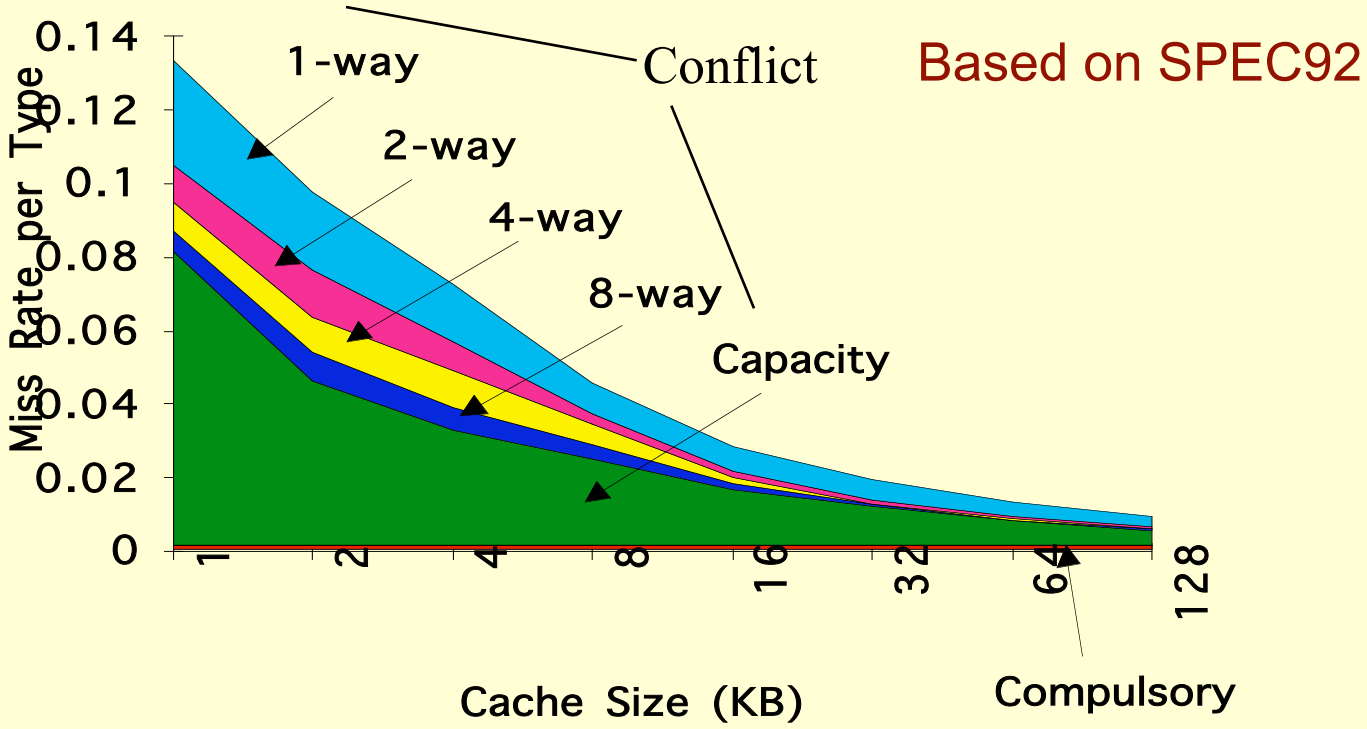
- Compulsory
  - The first access to a block is never in the cache. Also called cold start misses or first reference misses.
    (Misses in even an Infinite Cache)
- Capacity
  - If the cache cannot contain all the blocks needed during execution of a program, blocks must be discarded and later retrieved.
    (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



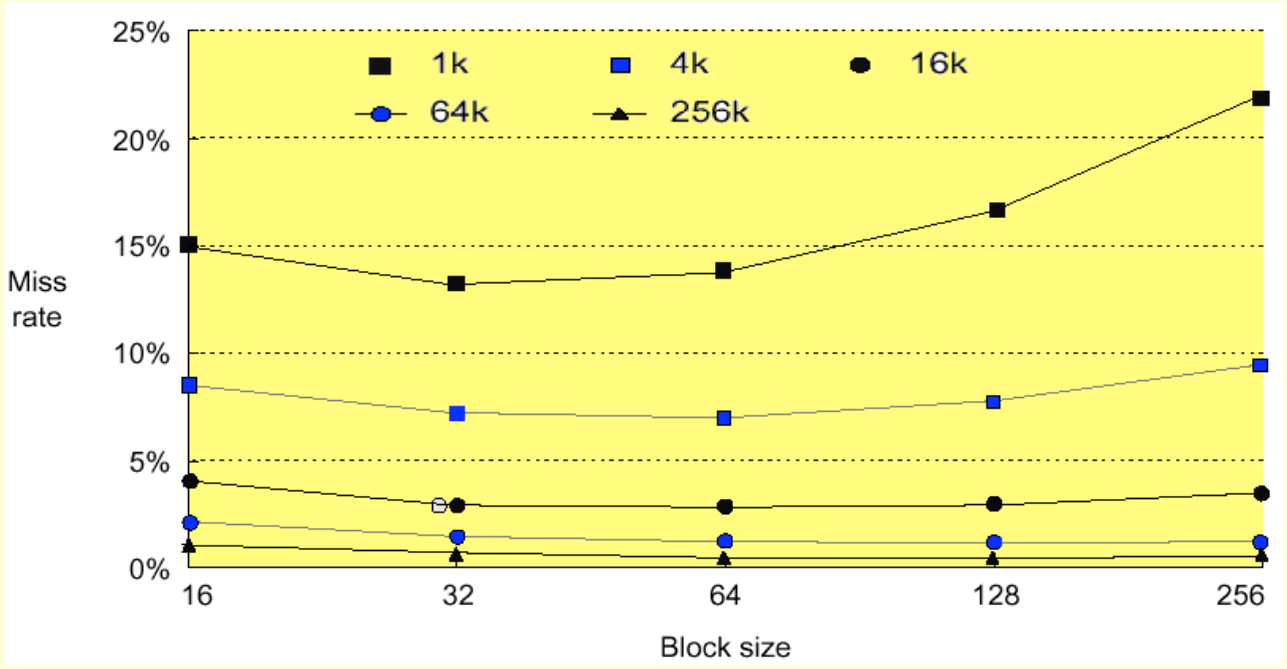
Techniques for Reducing Misses

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

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

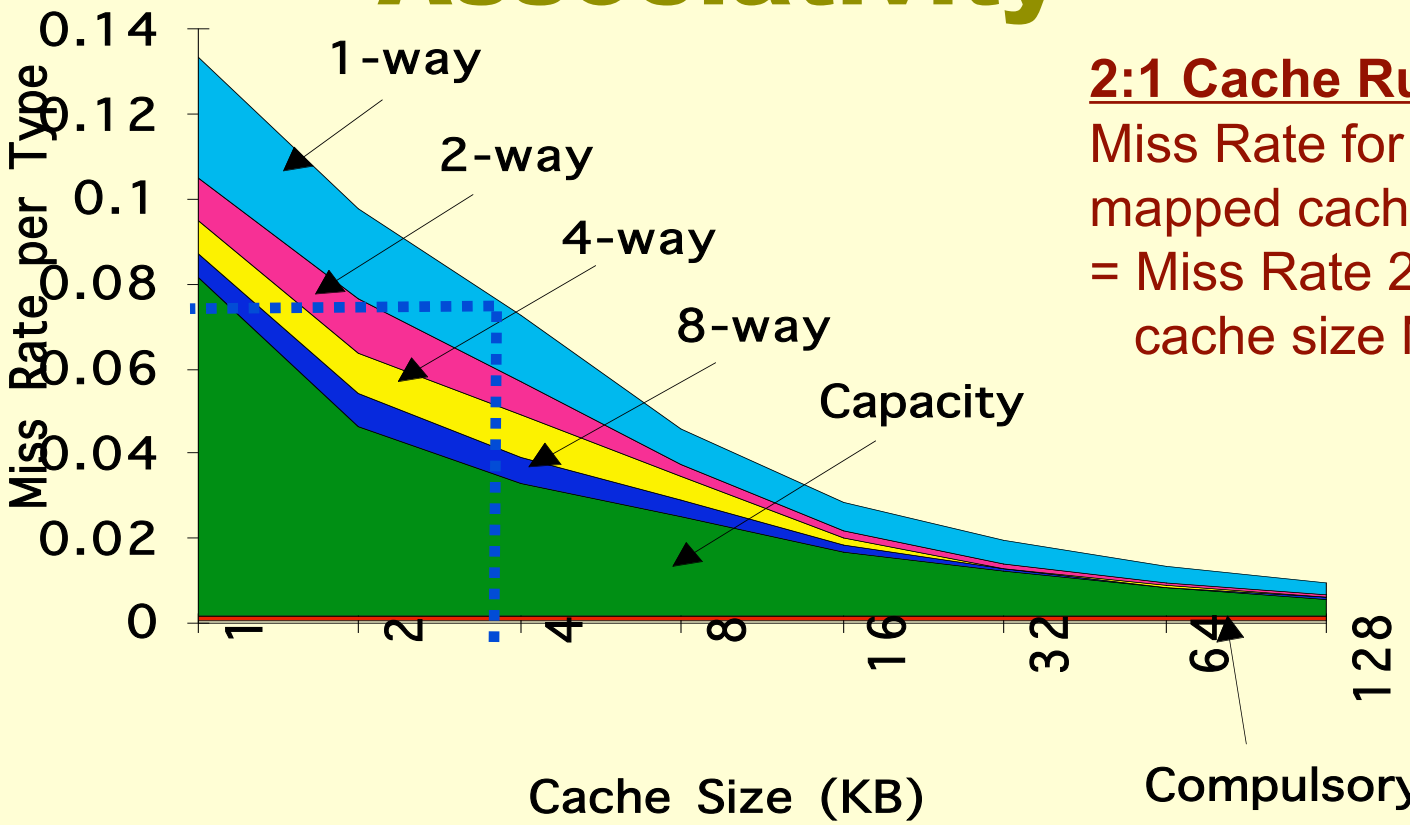
Slide: Dave Patterson

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

Compiler-based Cache Optimizations

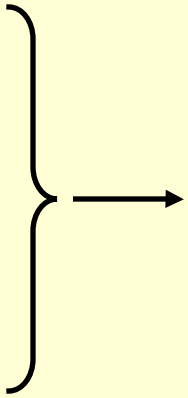
- Complier-based cache optimization reduces the miss rate without any hardware change or complexity
- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- For Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to 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];
```

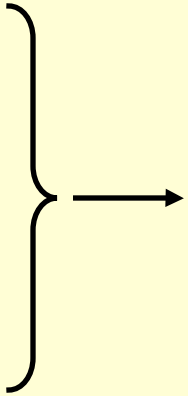


```
/* After: 1 array of stuctures */
struct merge {
        int val;
        int key;
};
struct merge merged_array[SIZE];
```

- 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];
```



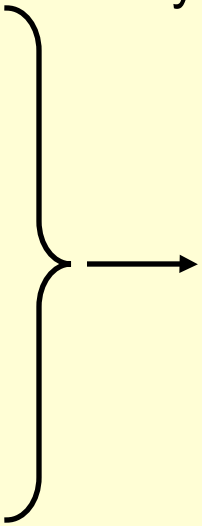
```
/* After */
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];
```

- 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];
```



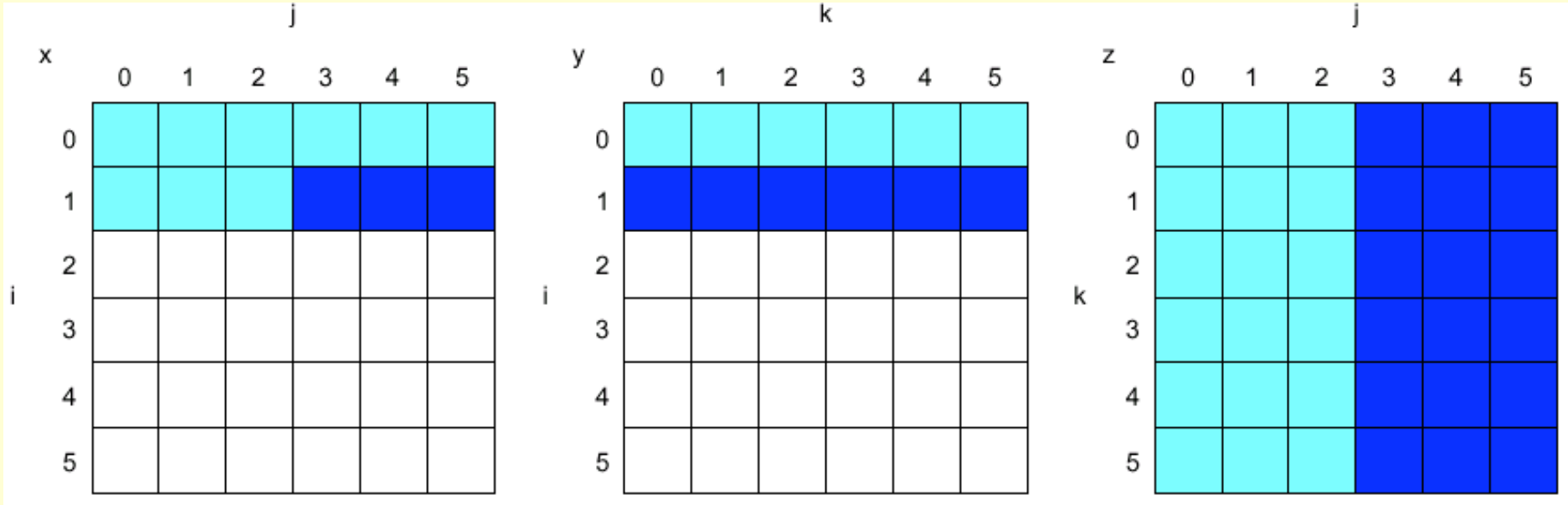
```
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1) {
        a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
}
```

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;
   }
```

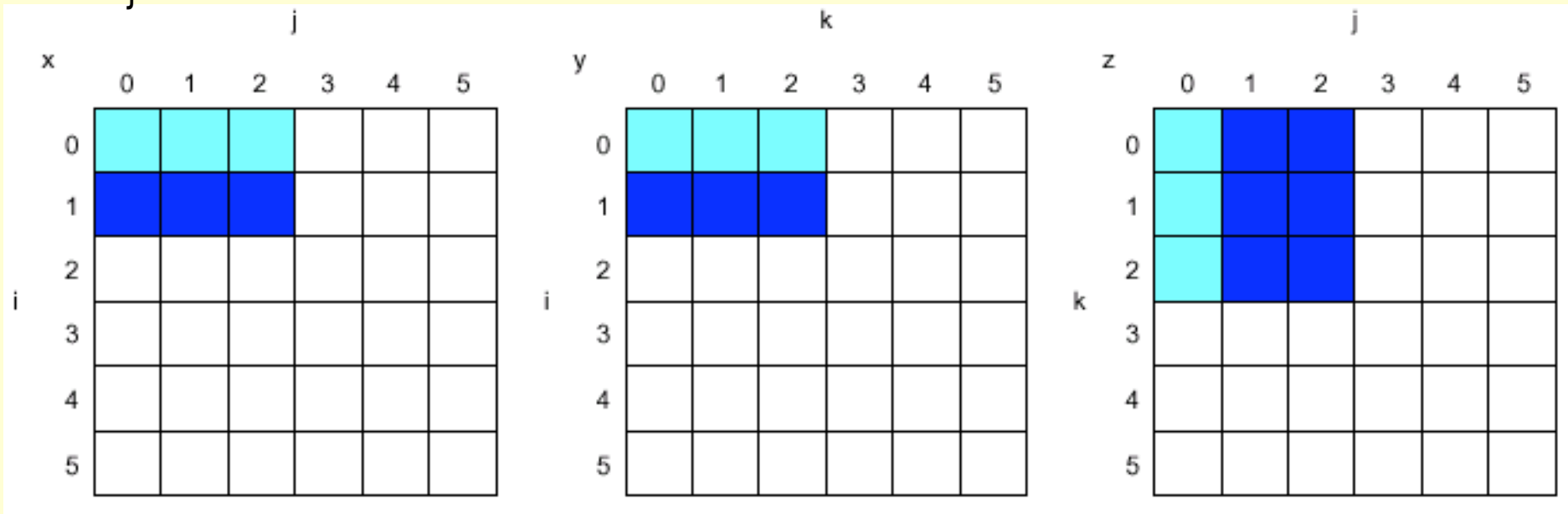
- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]
- Capacity Misses a function of N & Cache Size:
  - 3 N N 4 bytes => no capacity misses;
- Idea: compute on B B sub-matrix that fits



Blocking Example

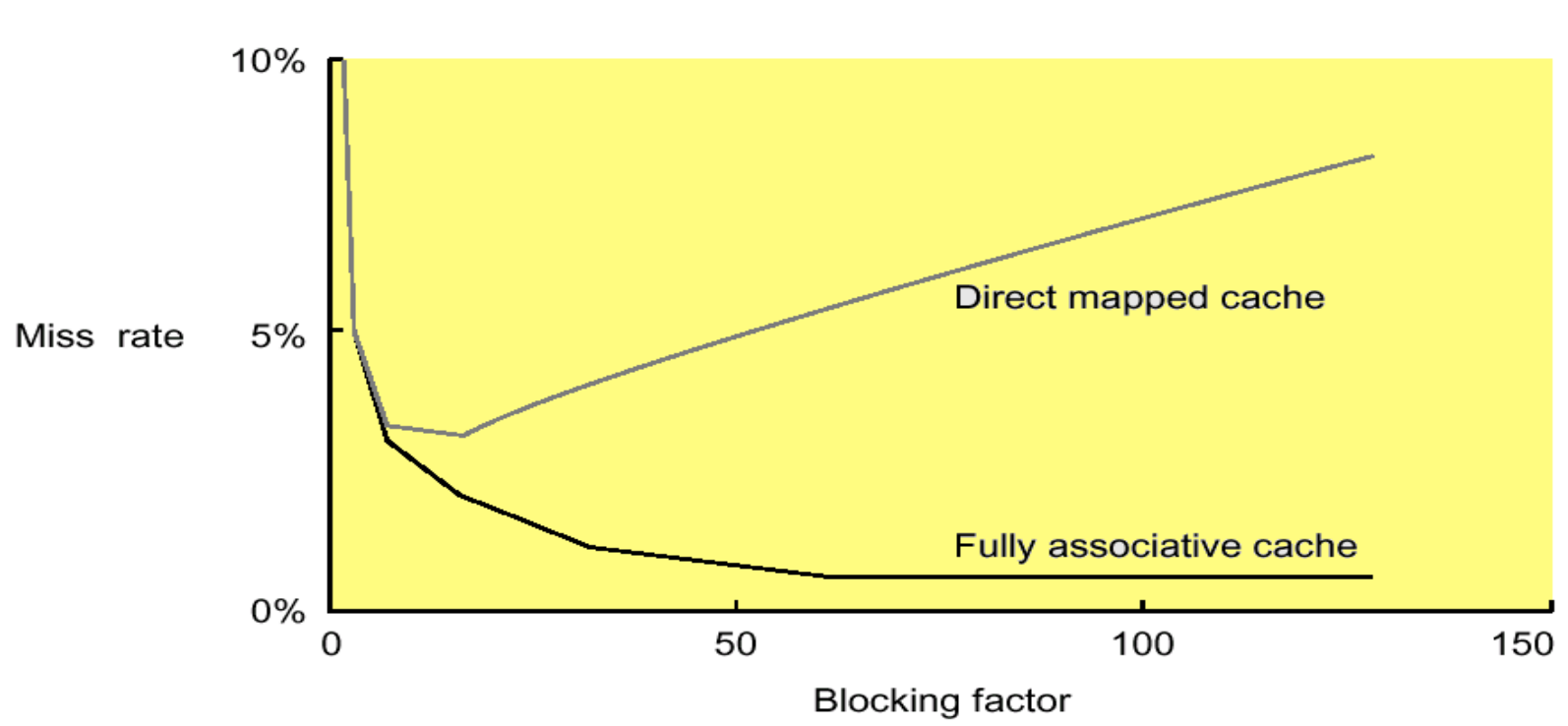
```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1) {
        r = 0;
        for (k = kk; k < min(kk+B-1,N); k = k+1) {
            r = r + y[i][k] * z[k][j];};
            x[i][j] = x[i][j] + r;
   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

- 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



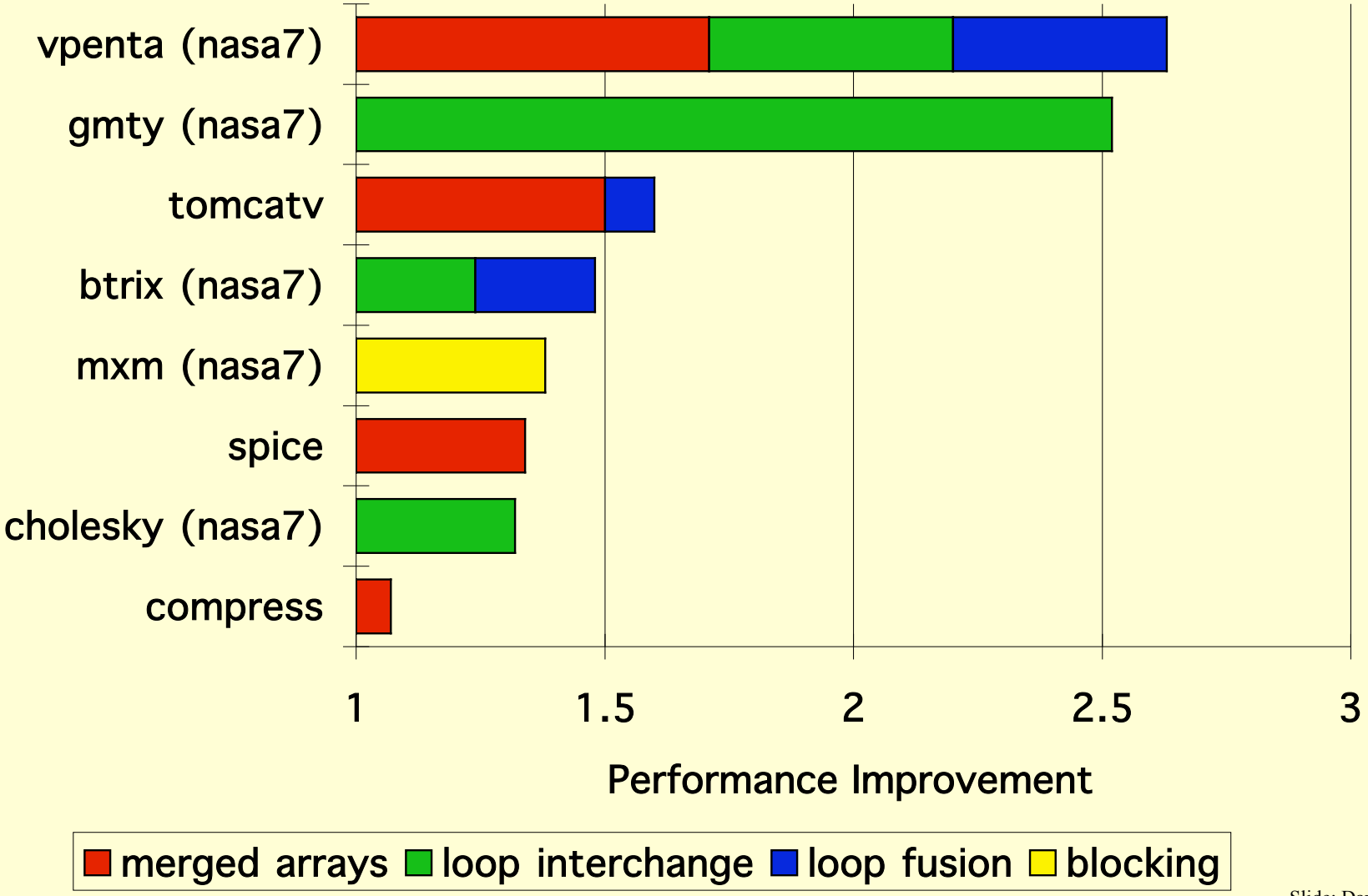
Blocking Factor

- 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

$$CPUtime = IC \quad CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \quad \textbf{Miss rate} \quad Miss\ penalty \quad Clock\ cycle\ time$$

- 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

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$_{L1}$ +

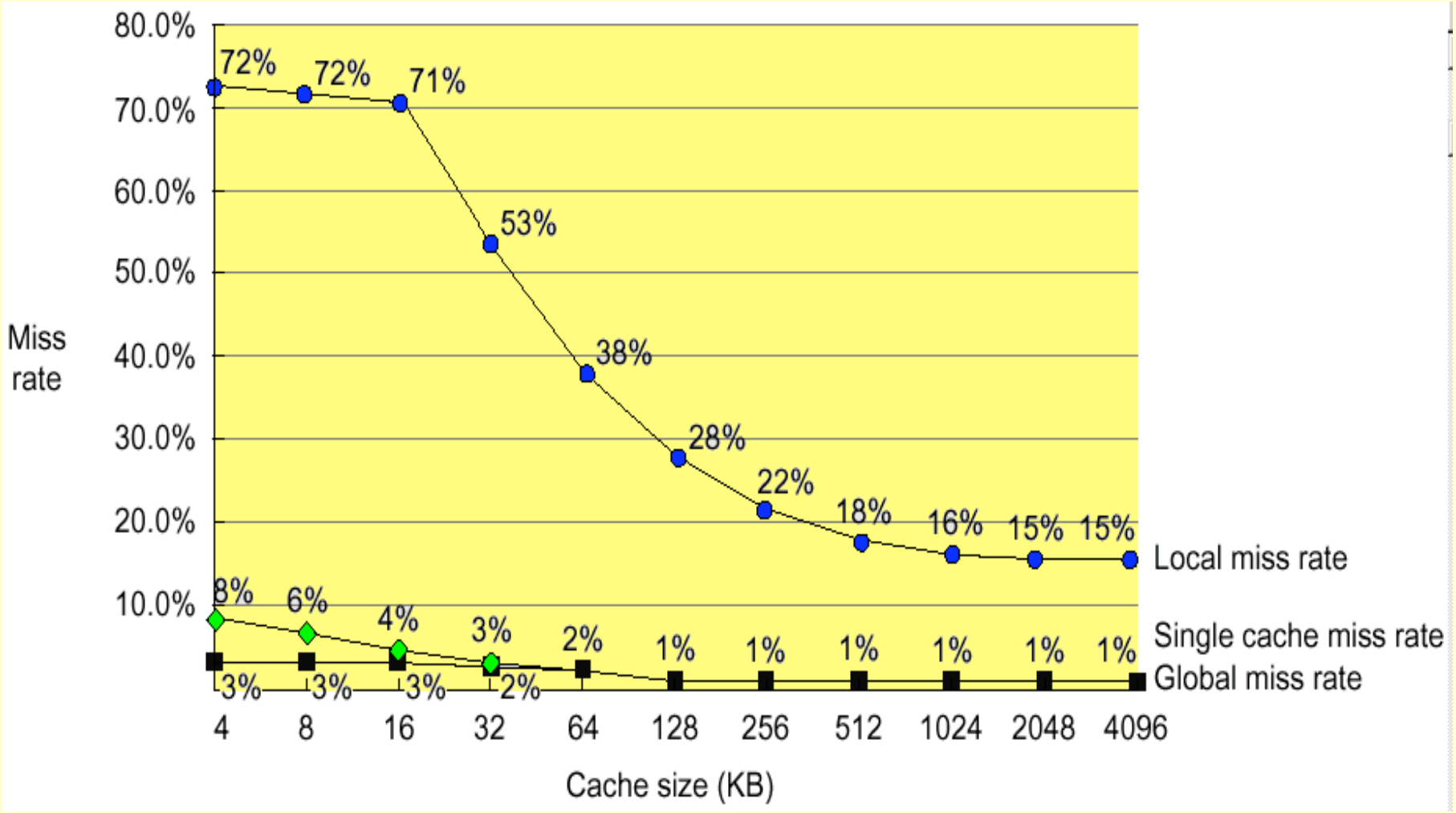
Miss Rate$_{L1}$ x (Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$)

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$_{L2}$)

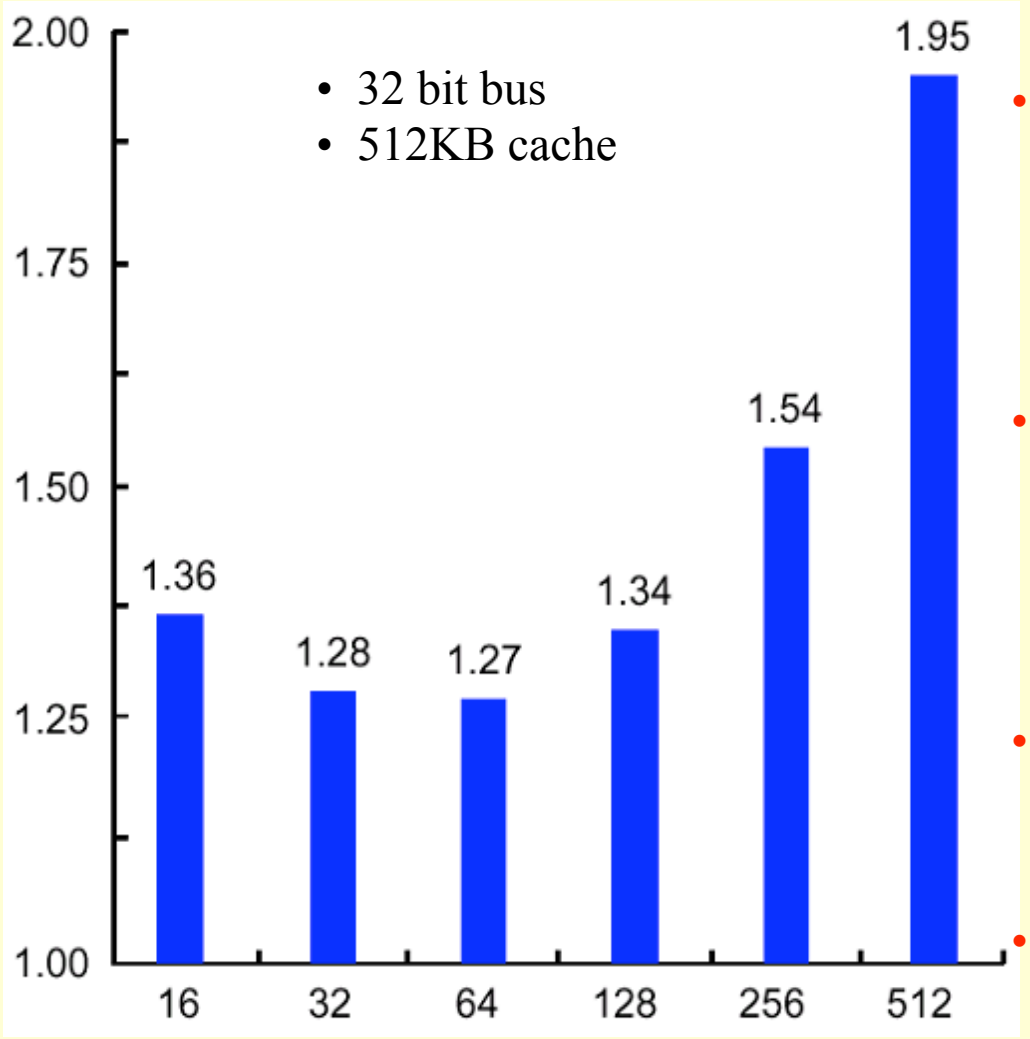
Global Miss Rate is what matters since the local miss rate is a function only of the secondary cache

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

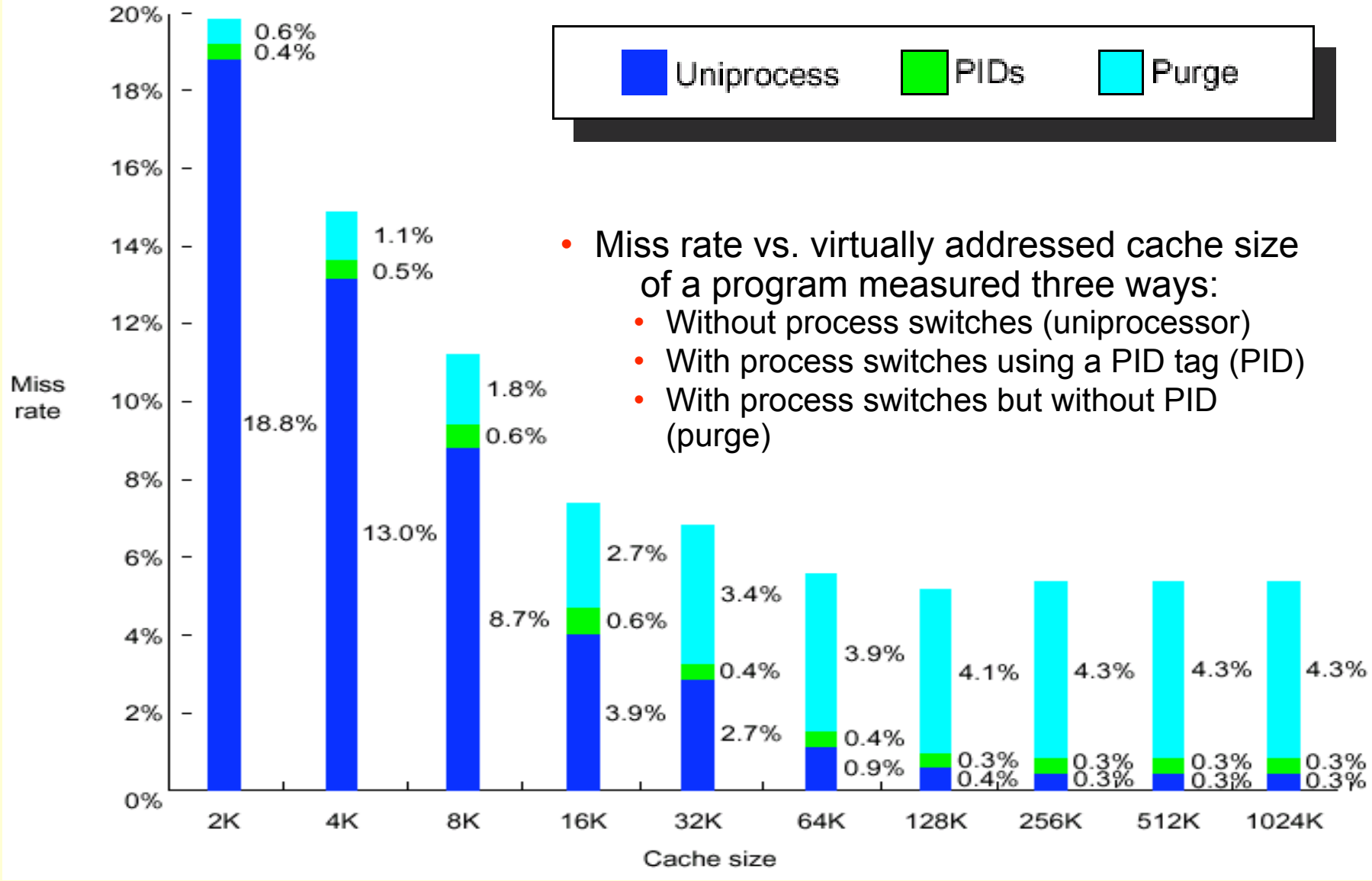
- Design simplicity limits the complexity of the control logic and allows to shorter clock cycles (e.g. direct mapped organization)
- On-chip integration decreases signal propagation delay, thus reducing hit time (small on-chip first level cache and large off-chip L2 cache)
  - Alpha 21164 has 8KB Instruction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

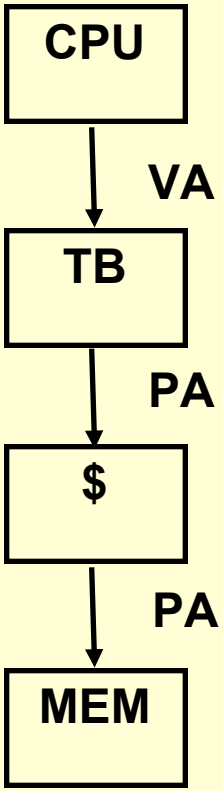
Impact of Using Process ID



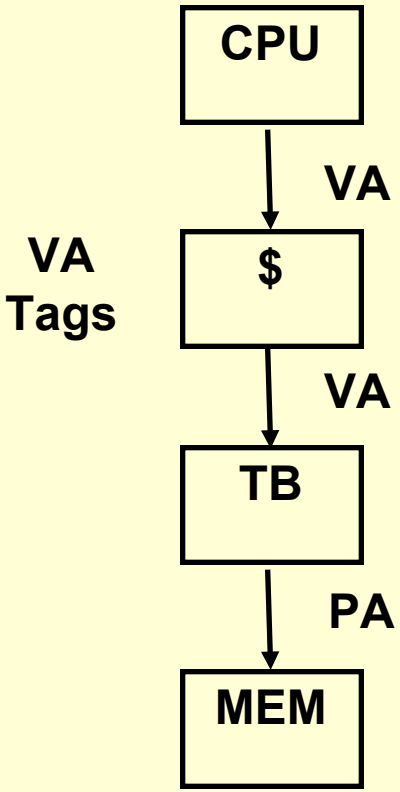
- Miss rate vs. virtually addressed cache size of a program measured three ways:
  - Without process switches (uniprocessor)
  - With process switches using a PID tag (PID)
  - With process switches but without PID (purge)

Virtually Addressed Caches

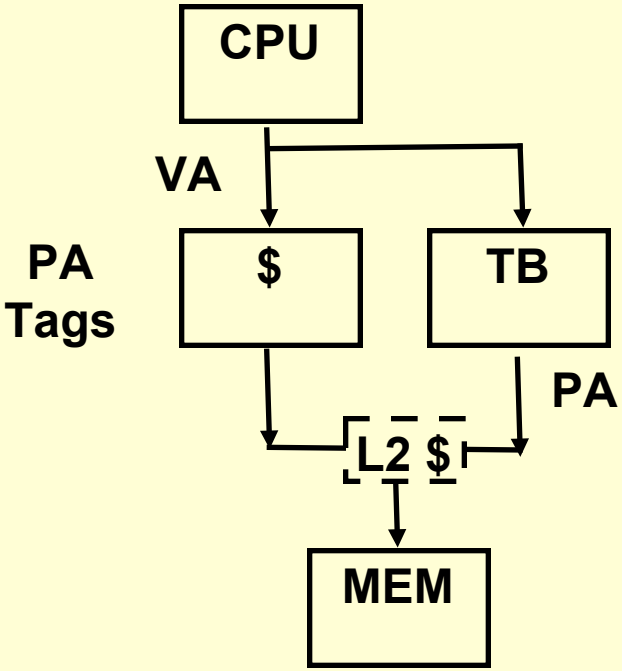
VA: Virtual address **TB**: Translation buffer **PA**: Page address



Conventional Organization



Virtually Addressed Cache
Translate only on miss
Synonym Problem

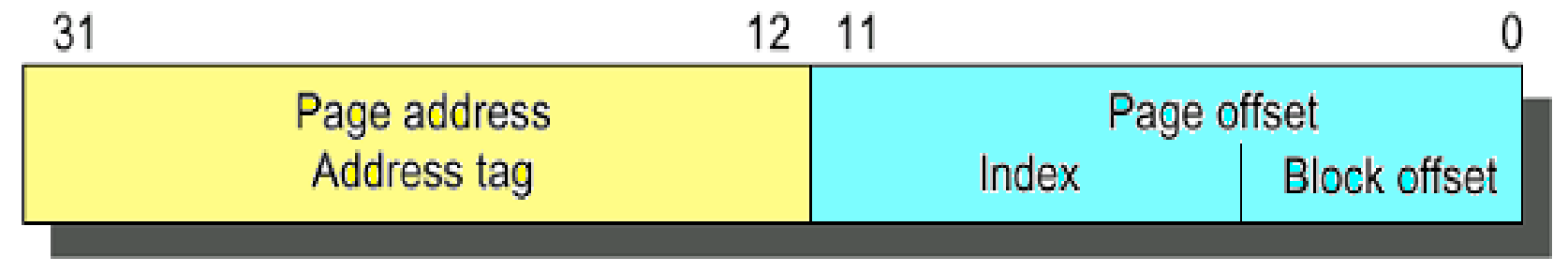


Overlap $ access with VA translation: requires $ index to remain invariant across translation

* 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)

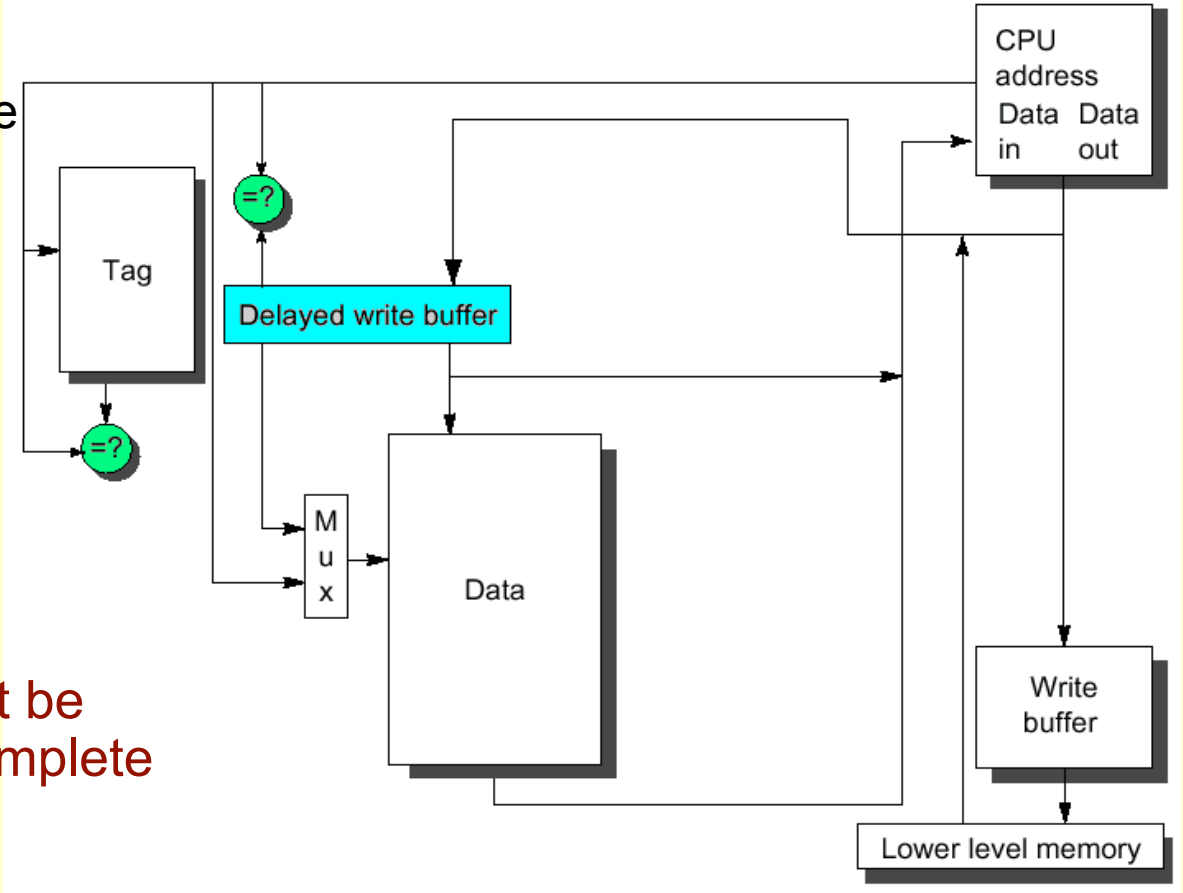


- To support bigger caches and uses same technique, one can:
  - Use higher associativity since the tag size gets smaller (moves barrier towards the most part of the address)
  - The operating system is to implement page coloring since it will fix a few least significant bits in the address (move part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



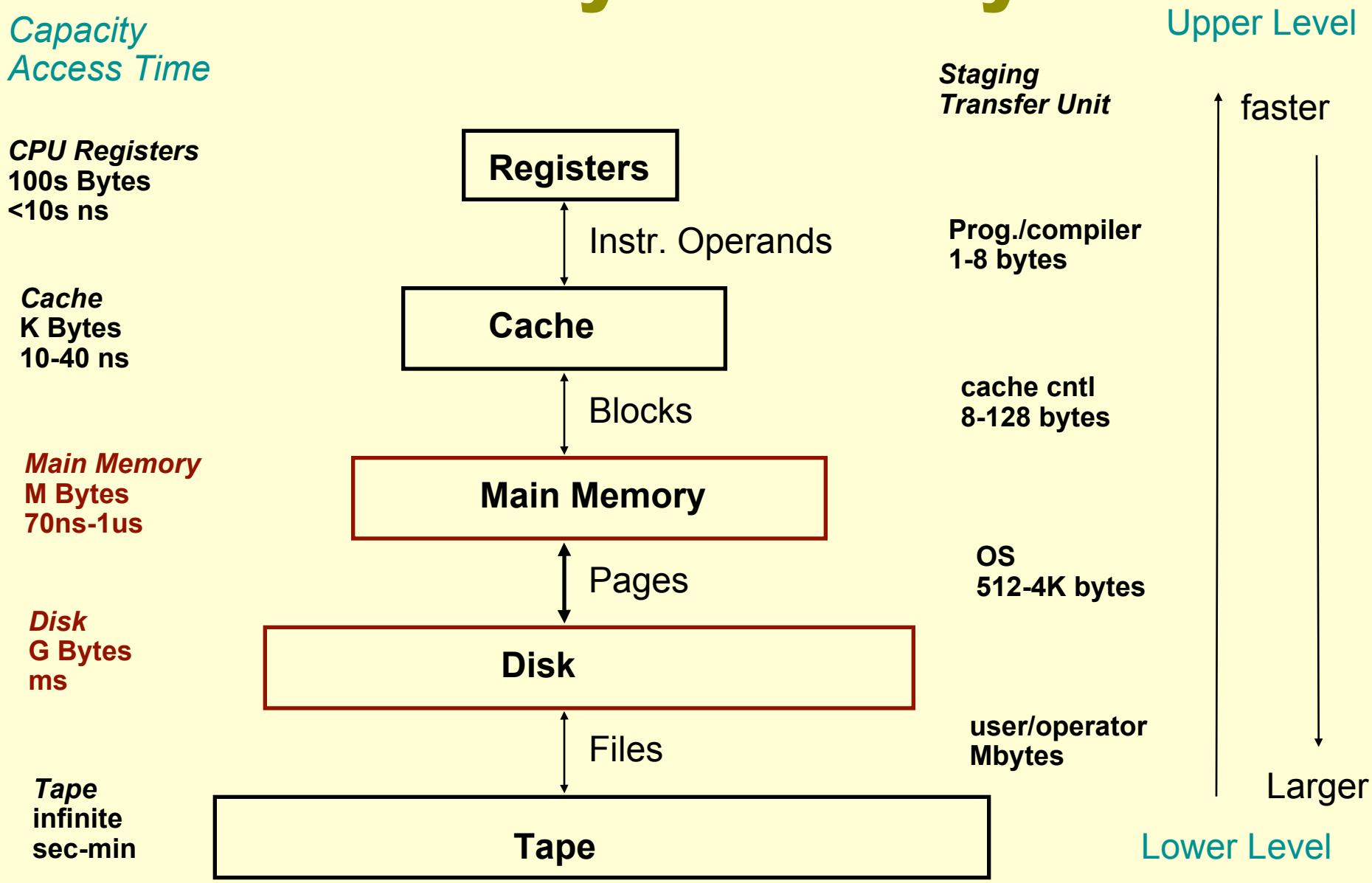
"Delayed Write Buffer"; must be checked on reads; either complete write or read from buffer

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



* Slide is courtesy of Dave Patterson

Main Memory Background

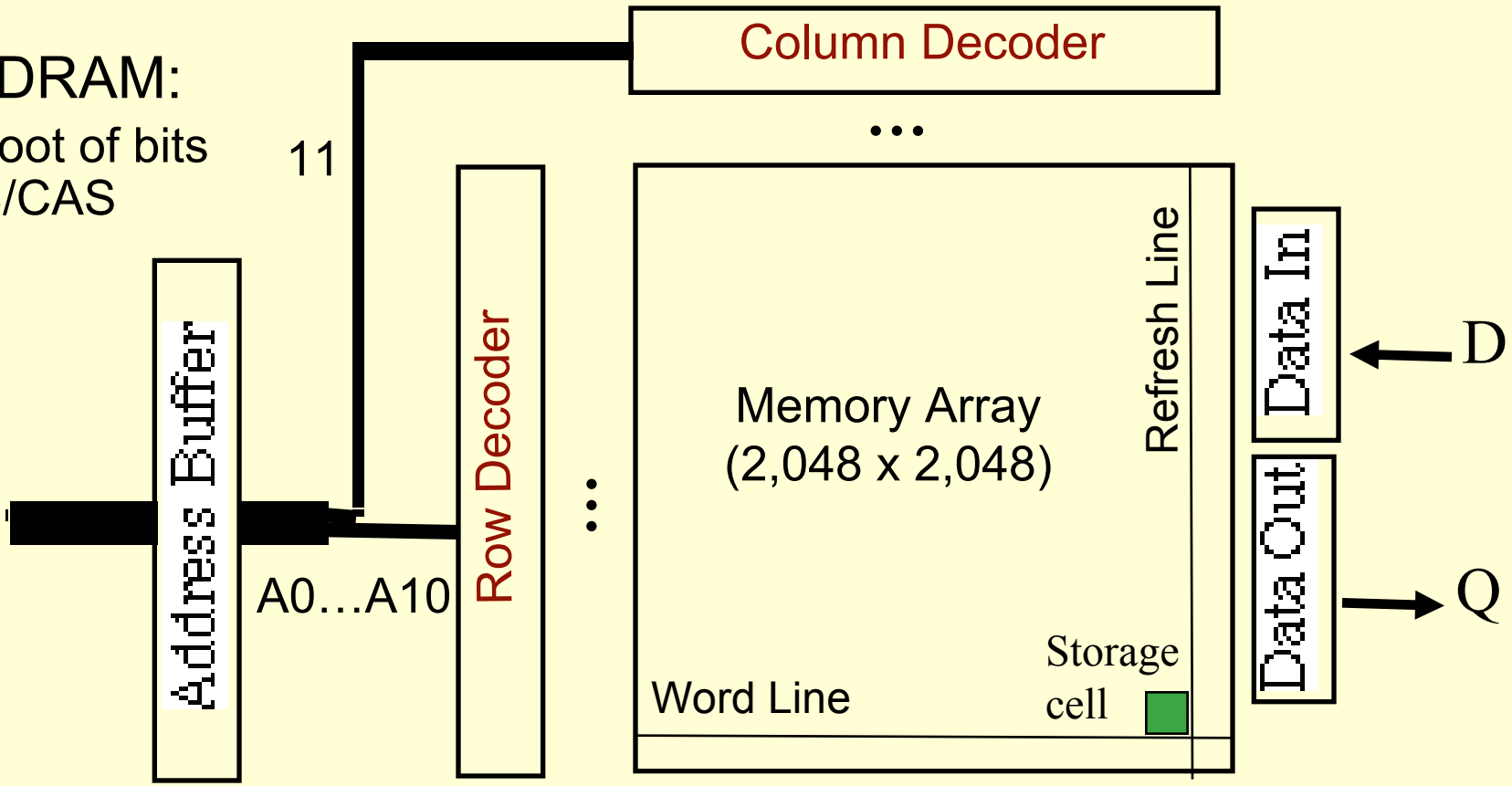
- Performance of Main Memory:
  - Latency: affects cache miss penalty
    - *Access Time*: time between request and word arrives
    - *Cycle Time*: time between requests
  - Bandwidth: primary concern for I/O & large Block Miss Penalty (L2)
- Main Memory is *DRAM*: Dynamic Random Access Memory
  - Dynamic 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

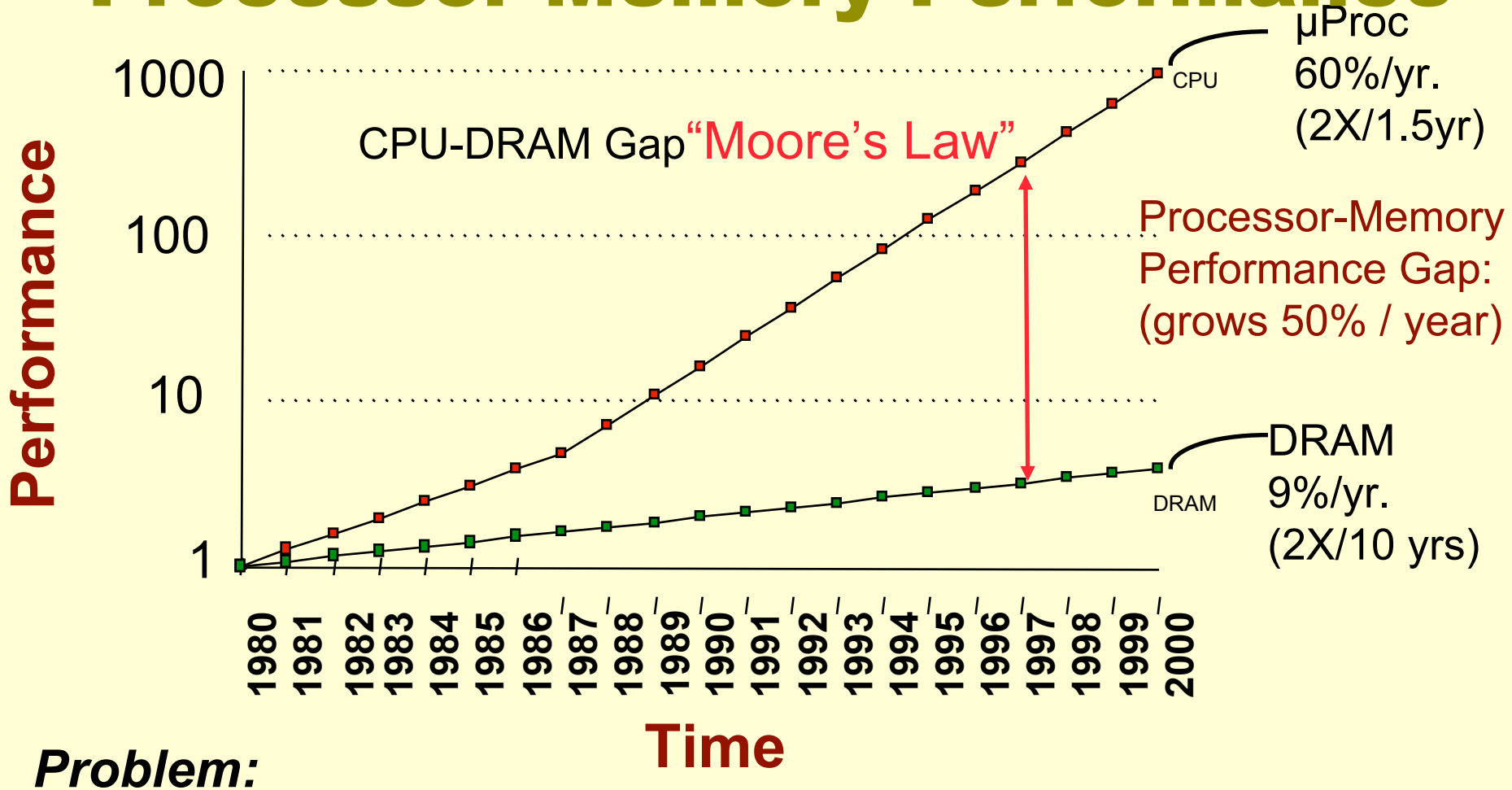
4 Mbit DRAM:

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



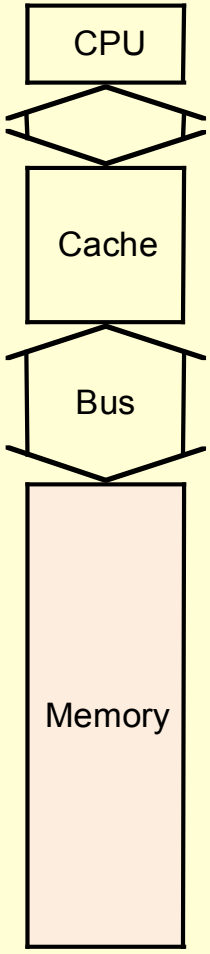
Problem:

Improvements in access time are not enough to catch up

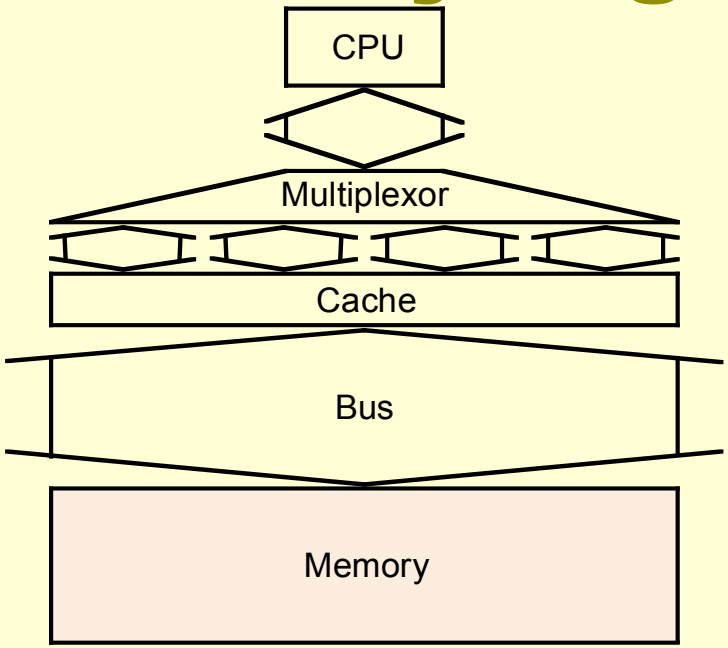
Solution:

Increase the bandwidth of main memory (improve throughput)

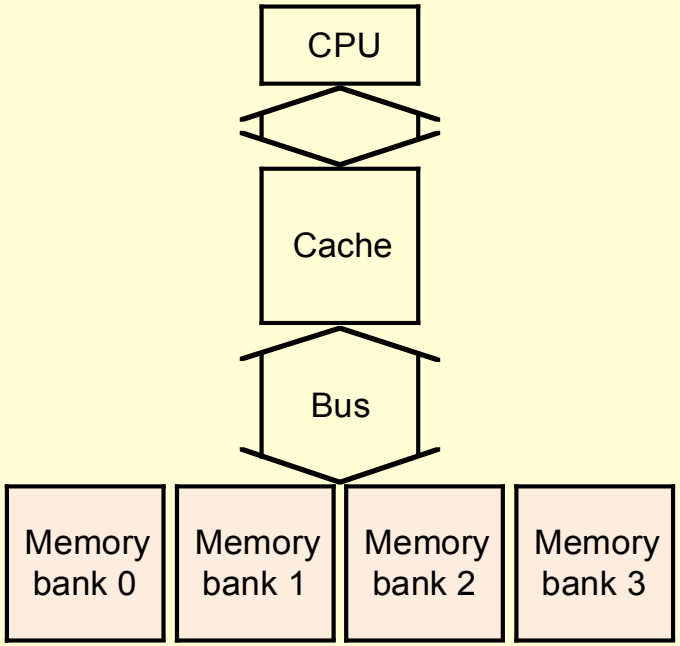
Memory Organization



a. One-word-wide memory organization



b. Wide memory organization



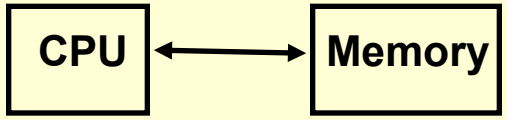
c. Interleaved memory organization

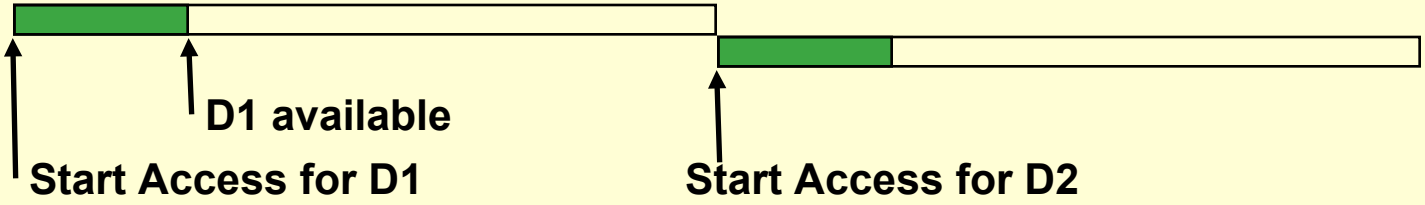
- *Simple*: CPU, Cache, Bus, Memory same width (32 bits)
- *Wide*: CPU/Mux 1 word; Mux/Cache, Bus, Memory N words
- *Interleaved*: CPU, Cache, Bus 1 word: Memory N Modules (4 Modules); example is *word interleaved*

Memory organization would have significant effect on bandwidth

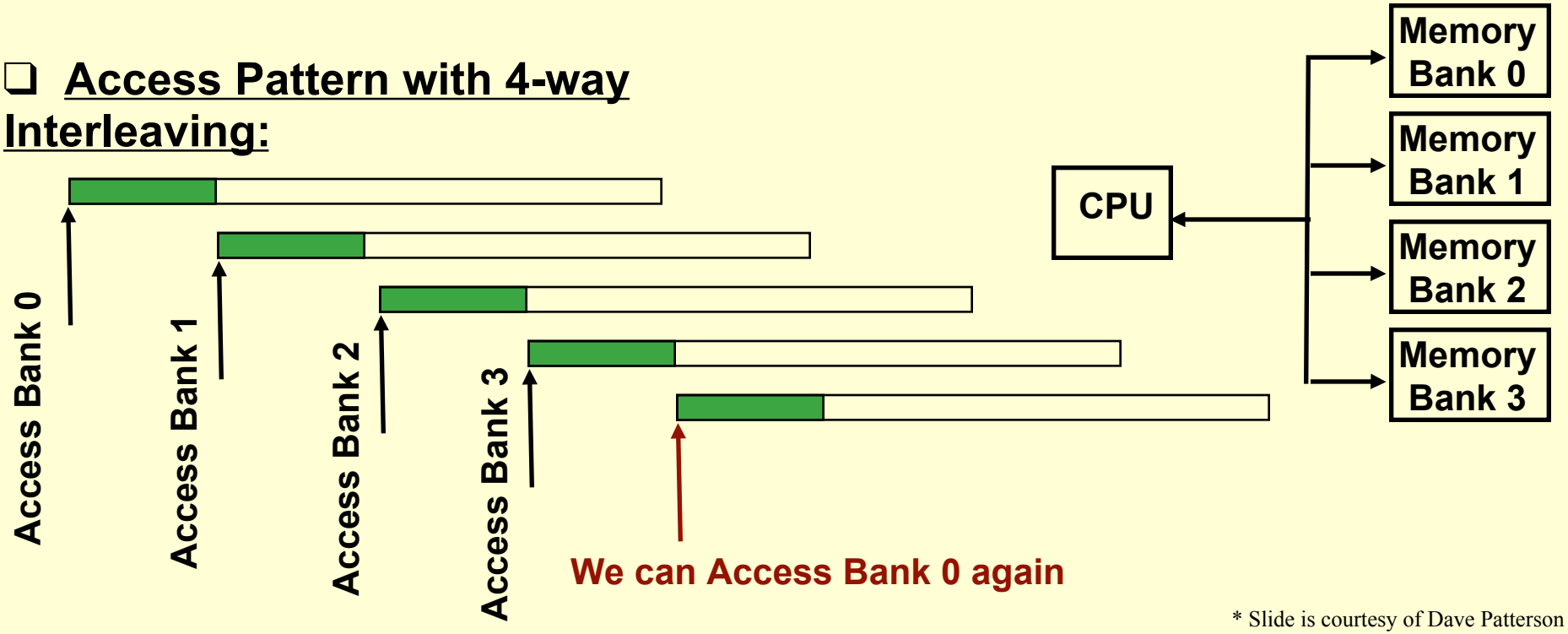
Memory Interleaving

- **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*: 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