Lecture 12: Memory Hierarchy -- Cache Optimizations

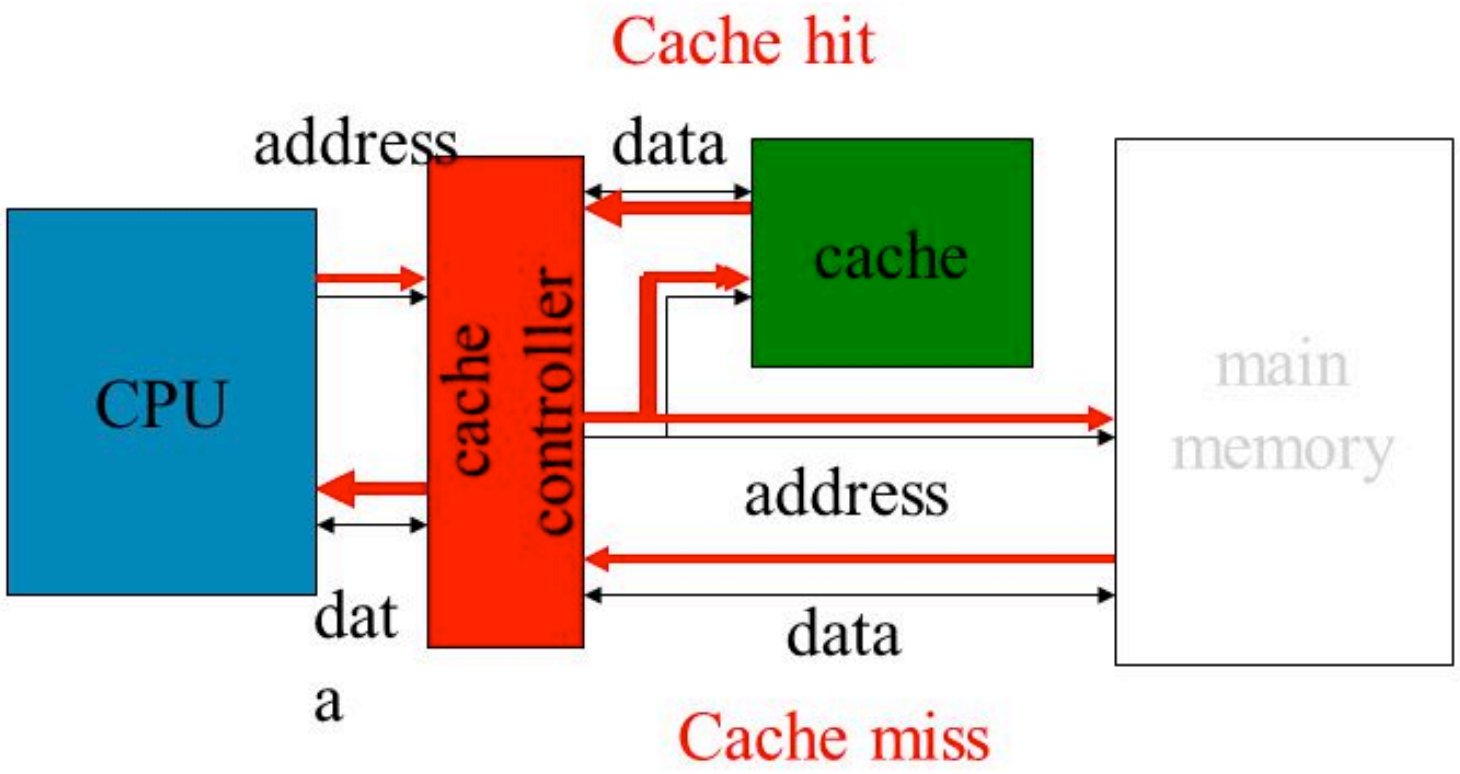
CSCE 513 Computer Architecture

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
https://passlab.github.io/CSCE513

Topics for Memory Hierarchy

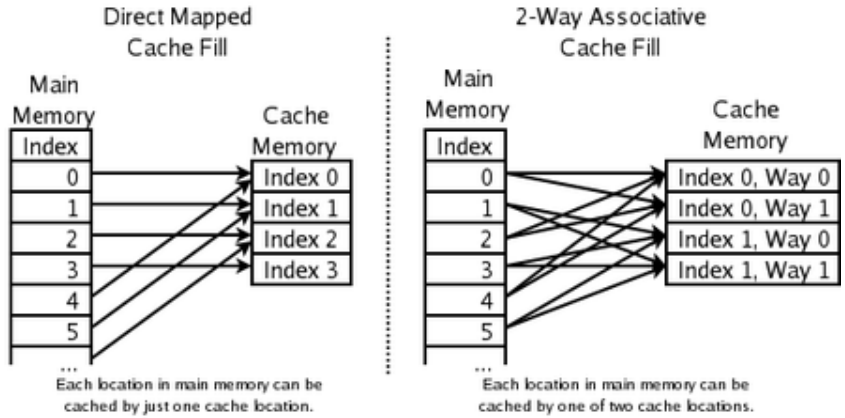
- Memory Technology and Principal of Locality
  - CAQA: 2.1, 2.2, B.1
  - COD: 5.1, 5.2
- Cache Organization and Performance
  - CAQA: B.1, B.2
  - COD: 5.2, 5.3
- ☛ Cache Optimization
  - 6 Basic Cache Optimization Techniques
    - CAQA: B.3
  - 10 Advanced Optimization Techniques
    - CAQA: 2.3
- Virtual Memory and Virtual Machine
  - CAQA: B.4, 2.4; COD: 5.6, 5.7
  - Skip for this course

Cache and Memory Access (ld/st instructions)



A Summary on Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - **Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant**
- **Conflict** (collision):
  - Multiple memory locations mapped to the same cache location
  - **Solution 1: increase cache size**
  - **Solution 2: increase associativity**



- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - **Solution: increase cache size**

- Coherence (Invalidation): other process (e.g., I/O) updates memory
  - We will cover it later on.

Memory Hierarchy Performance

- Two indirect performance measures have waylaid many a computer designer.
  - ***Instruction count* is independent of the hardware;**
  - ***Miss rate* could be independent of the hardware mostly**

$$CPU\ Time = IC * \left(CPI_{Execution} + \frac{Memory\ Accesses}{Instruction} \times Miss\ Rate \times Miss\ Penalty\right) \times Clock\ Cycle\ Time$$

- A better measure of memory hierarchy performance is the ***Average Memory Access Time* (AMAT) per instructions**

$$AMAT = Hit\ time + Miss\ rate \times Miss\ penalty$$

Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle)
  - CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instruction accesses get same miss penalty

Ideal CPI	1.1
Data Miss	1.5
Inst Miss	0.5

- CPI = ideal CPI + average stalls per instruction

  = 1.1(cycles/ins)

  + [**0.30** (DataMops/ins) x 0.10 (miss/DataMop) x 50 (cycle/miss)]

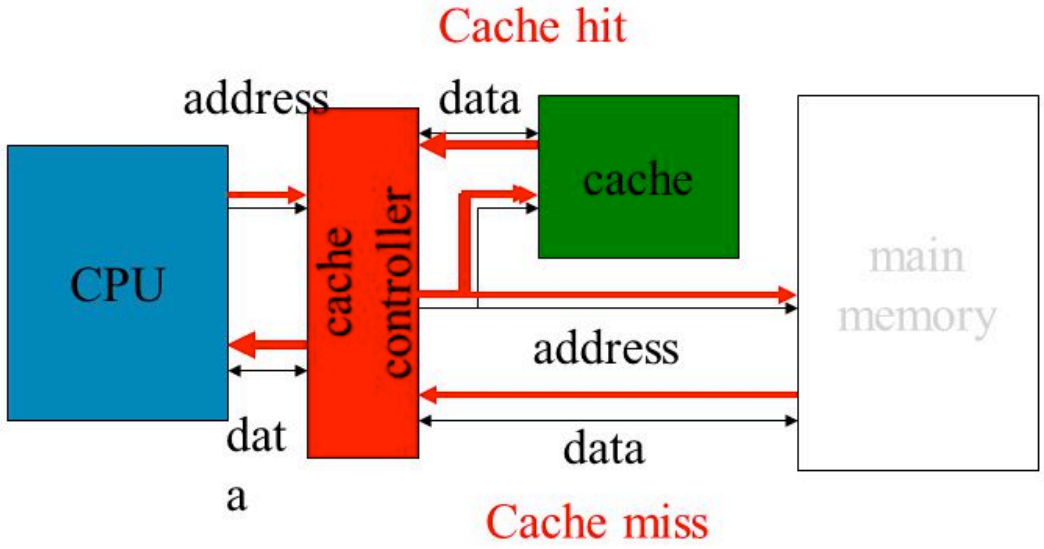
  + [**1** (InstMop/ins) x 0.01 (miss/InstMop) x 50 (cycle/miss)]

  = (1.1 + 1.5 + .5) cycle/ins = 3.1

- **2/3.1 (64.5%) of the time the proc is stalled waiting for memory!**

Improving Cache Performance

Average Memory Access Time = Hit Time + Miss Rate * Miss Penalty



Goals	Basic Approaches
Reducing Miss Rate	Larger block size, larger cache size and higher associativity
Reducing Miss Penalty	Multilevel caches, and higher read priority over writes
Reducing Hit Time	Avoid address translation when indexing the cache

1. Reduce Miss Rate via Larger Block Size

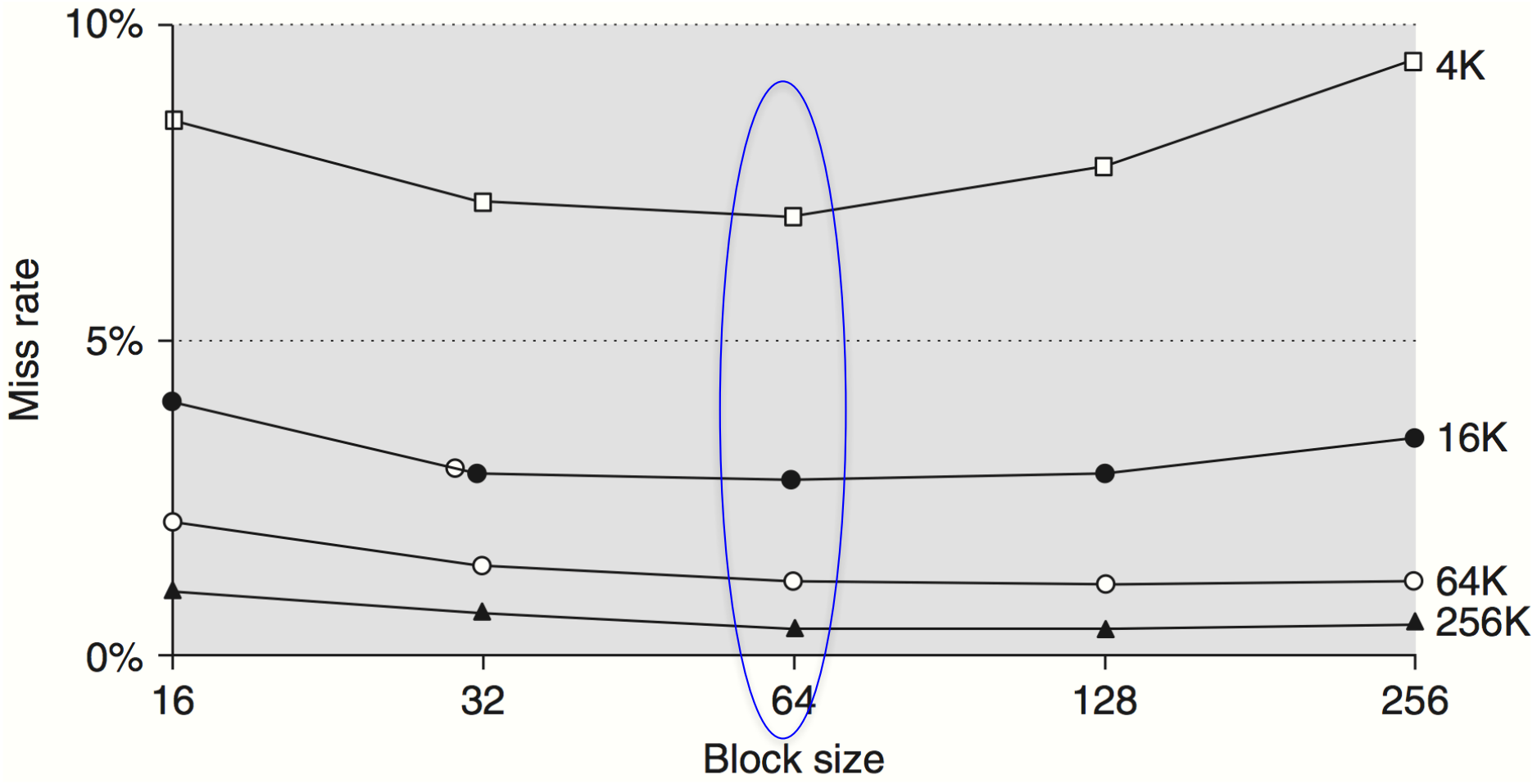


Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so

Much Larger Block Size: → Increase Miss Rate

	Cache size			
Block size	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Figure B.11 Actual miss rate versus block size for the five different-sized caches in Figure B.10. Note that for a 4 KB cache, 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KB in order for a 256-byte block to decrease misses.

Example: Miss Rate vs Reduce AMAT

Example Figure B.11 shows the actual miss rates plotted in Figure B.10. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block size has the smallest average memory access time for each cache size in Figure B.11?

Answer Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

If we assume the hit time is 1 clock cycle independent of block size, then the access time for a 16-byte block in a 4 KB cache is

$$\text{Average memory access time} = 1 + (8.57\% \times 82) = 8.027 \text{ clock cycles}$$

and for a 256-byte block in a 256 KB cache the average memory access time is

$$\text{Average memory access time} = 1 + (0.49\% \times 112) = 1.549 \text{ clock cycles}$$

Larger Block Size: → Increase Miss Penalty
Choose a Block Size Based on AMAT

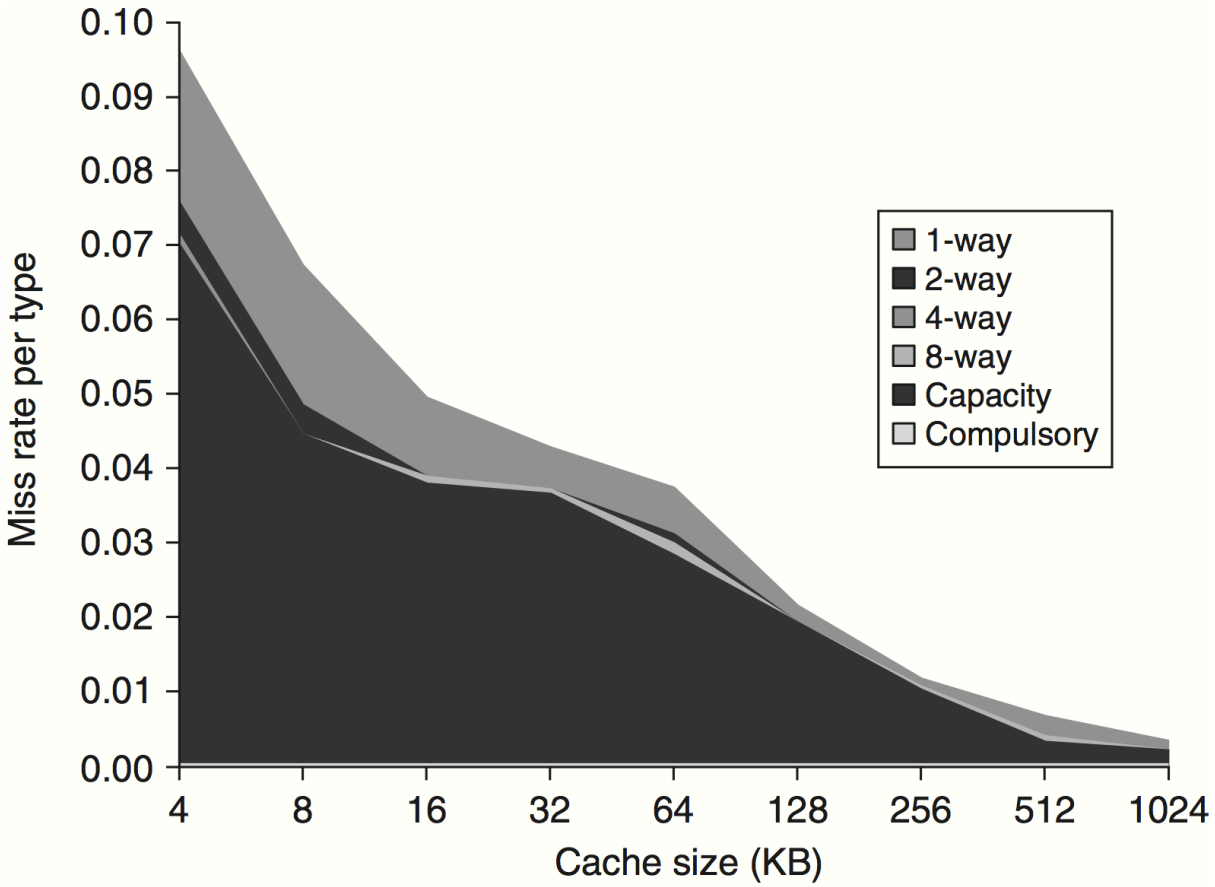
Block size	Miss penalty	Cache size 4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

Figure B.12 Average memory access time versus block size for five different-sized caches in Figure B.10. Block sizes of 32 and 64 bytes dominate. The smallest average time per cache size is boldfaced.

$$\text{Average Memory Access Time} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

2. Reduce Miss Rate via Larger Cache

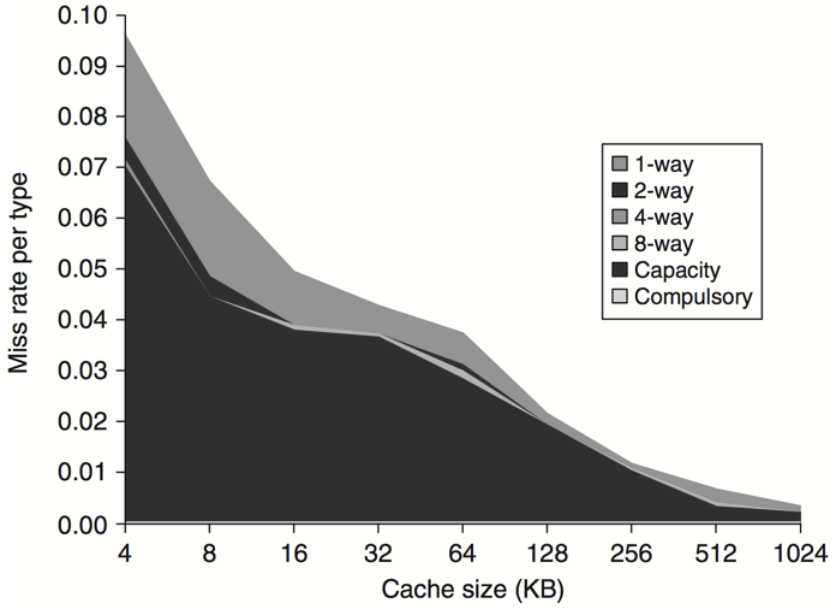
- Increasing capacity of cache reduces capacity misses
- May be longer hit time and higher cost
- Trends: Larger L2 or L3 off-chip caches



3. Reduce Miss Rate via Higher Associativity

- 2:1 Cache Rule:
  - Miss Rate DM cache size N = Miss Rate 2-way cache size N/2
- 8-way set associative is as effective as fully associative for practical purposes
- Tradeoff: higher associative cache complicates the circuit
  - May have longer clock cycle

- Beware: Execution time is the only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%



Associativity ←→ AMAT

Example Assume that higher associativity would increase the clock cycle time as listed below:

$$\text{Clock cycle time}_{2\text{-way}} = 1.36 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{4\text{-way}} = 1.44 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{8\text{-way}} = 1.52 \times \text{Clock cycle time}_{1\text{-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to a level 2 cache (see next subsection) that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure B.8 for miss rates, for which cache sizes are each of these three statements true?

$$\text{Average memory access time}_{8\text{-way}} < \text{Average memory access time}_{4\text{-way}}$$
$$\text{Average memory access time}_{4\text{-way}} < \text{Average memory access time}_{2\text{-way}}$$
$$\text{Average memory access time}_{2\text{-way}} < \text{Average memory access time}_{1\text{-way}}$$

	Associativity			
Cache size (KB)	**1-way**	**2-way**	**4-way**	**8-way**
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**

Associativity ←→ AMAT: High Associativity Leads to Higher Access Time

Answer Average memory access time for each associativity is

$$\text{Average memory access time}_{8\text{-way}} = \text{Hit time}_{8\text{-way}} + \text{Miss rate}_{8\text{-way}} \times \text{Miss penalty}_{8\text{-way}}$$
$$= 1.52 + \text{Miss rate}_{8\text{-way}} \times 25$$
$$\text{Average memory access time}_{4\text{-way}} = 1.44 + \text{Miss rate}_{4\text{-way}} \times 25$$

	Associativity			
Cache size (KB)	**1-way**	**2-way**	**4-way**	**8-way**
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. Boldface type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

4. Reduce Miss Penalty via Multilevel Caches

- Approaches
  - Make the cache faster to keep pace with the speed of CPUs
  - Make the cache larger to overcome the widening gap
- **L1: fast hits, L2: fewer misses**
- L2 Equations

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

so

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

- $\text{Hit Time}_{L1} << \text{Hit Time}_{L2} << \ldots << \text{Hit Time}_{Mem}$
- $\text{Miss Rate}_{L1} < \text{Miss Rate}_{L2} < \ldots$

Miss Rate in Multilevel Caches

- **Local miss rate**— misses in this cache divided by the total number of memory accesses to this cache (Miss rateL1 , Miss rateL2)
  - L1 cache skims the cream of the memory accesses
- **Global miss rate**—misses in this cache divided by the total number of memory accesses generated by the CPU (Miss rateL1, Miss RateL1 x Miss RateL2)
  - Indicate what fraction of the memory accesses that leave the CPU go all the way to memory

Miss Rates in Multilevel Caches

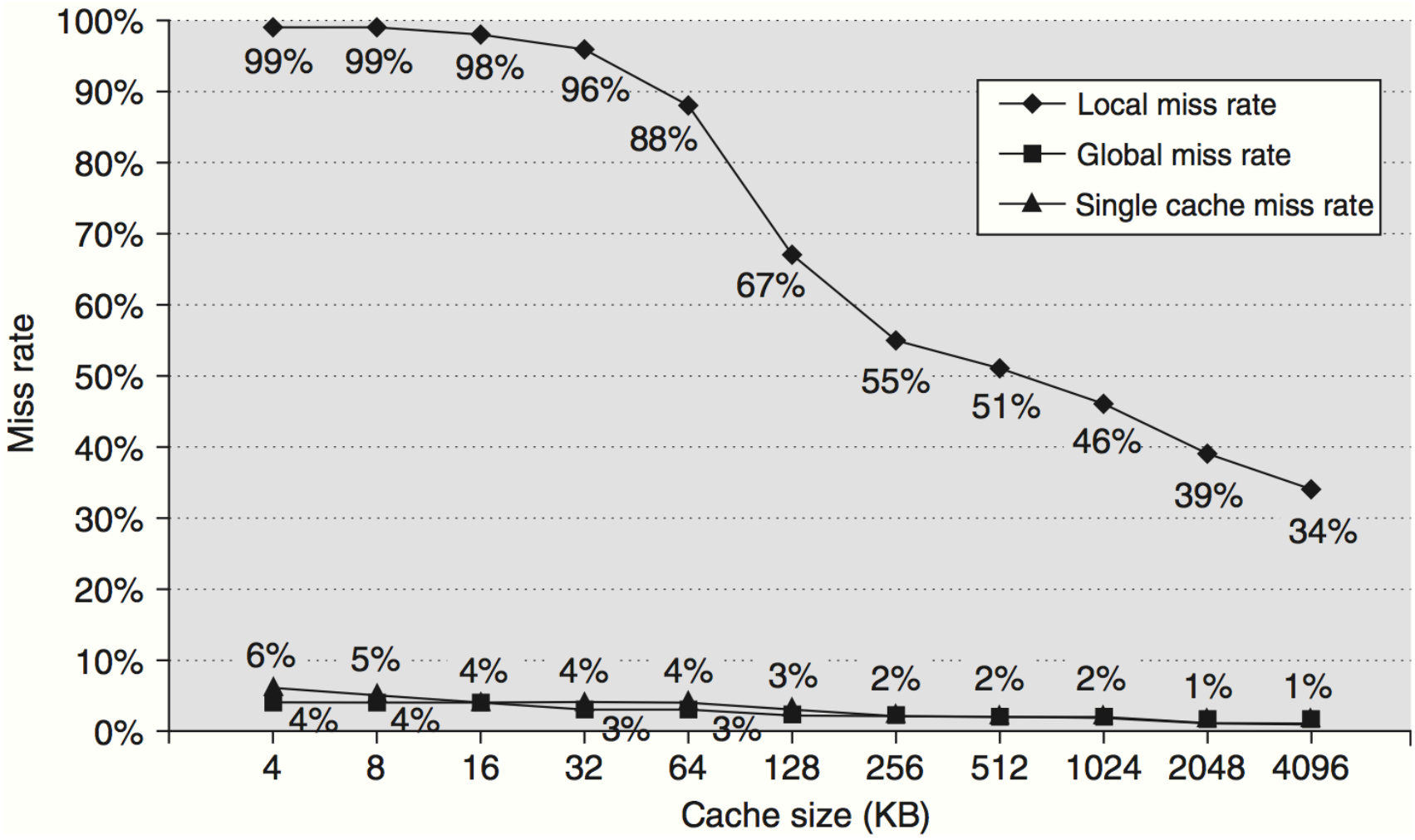


Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64 KB first-level caches make little sense, as reflected in the high miss rates. After 256 KB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and

Multilevel Caches: Design of L2

- Size
  - Since everything in L1 cache is likely to be in L2 cache, L2 cache should be much bigger than L1
- Whether data in L1 is in L2
  - novice approach: design L1 and L2 independently
  - multilevel inclusion: L1 data are always present in L2
    - Advantage: easy for consistency between I/O and cache (checking L2 only)
    - Drawback: L2 must invalidate all L1 blocks that map onto the 2nd-level block to be replaced => slightly higher 1st-level miss rate
      - i.e. Intel Pentium 4: 64-byte block in L1 and 128-byte in L2
  - multilevel exclusion: L1 data is never found in L2
    - A cache miss in L1 results in a swap of blocks between L1 and L2
    - Advantage: prevent wasting space in L2
      - i.e. AMD Athlon: 64 KB L1 and 256 KB L2

Multilevel Caches: Example

Example Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

Answer The miss rate (either local or global) for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%. Then

$$\begin{aligned}\text{Average memory access time} &= \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ clock cycles}\end{aligned}$$

Multilevel Caches: Example

To see how many misses we get per instruction, we divide 1000 memory references by 1.5 memory references per instruction, which yields 667 instructions. Thus, we need to multiply the misses by 1.5 to get the number of misses per 1000 instructions. We have 40 × 1.5 or 60 L1 misses, and 20 × 1.5 or 30 L2 misses, per 1000 instructions. For average memory stalls per instruction, assuming the misses are distributed uniformly between instructions and data:

$$\begin{aligned}\text{Average memory stalls per instruction} &= \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \\ &\quad \times \text{Miss penalty}_{L2} \\ &= (60/1000) \times 10 + (30/1000) \times 200 \\ &= 0.060 \times 10 + 0.030 \times 200 = 6.6 \text{ clock cycles}\end{aligned}$$

If we subtract the L1 hit time from the average memory access time (AMAT) and then multiply by the average number of memory references per instruction, we get the same average memory stalls per instruction:

$$(5.4 - 1.0) \times 1.5 = 4.4 \times 1.5 = 6.6 \text{ clock cycles}$$

As this example shows, there may be less confusion with multilevel caches when calculating using misses per instruction versus miss rates.

5. Reduce Miss Penalty by Giving Priority to Read Misses over Writes

- Serve reads before writes have been completed
- Write through with write buffers

```
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)
```

  Problem: 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

  Suppose a read miss will replace a dirty block
  - Normal: Write dirty block to memory, and then do the read
  - Instead: Copy the dirty block to a write buffer, do the read, and then do the write
  - CPU stall less since restarts as soon as do read

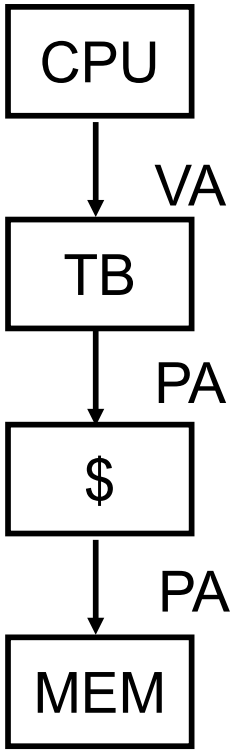
6. Reduce Hit Time by Avoiding Address Translation during Indexing of the Cache

- Importance of cache hit time
  - Average Memory Access Time = **Hit Time** + Miss Rate * Miss Penalty
  - More importantly, cache access time **limits the clock cycle rate** in many processors today!
- Fast hit time:
  - Quickly and efficiently find out if data is in the cache, and
  - if it is, get that data out of the cache
- Four techniques:
  1. Small and simple caches
  2. Avoiding address translation during indexing of the cache
  3. Pipelined cache acces
  4. Trace caches

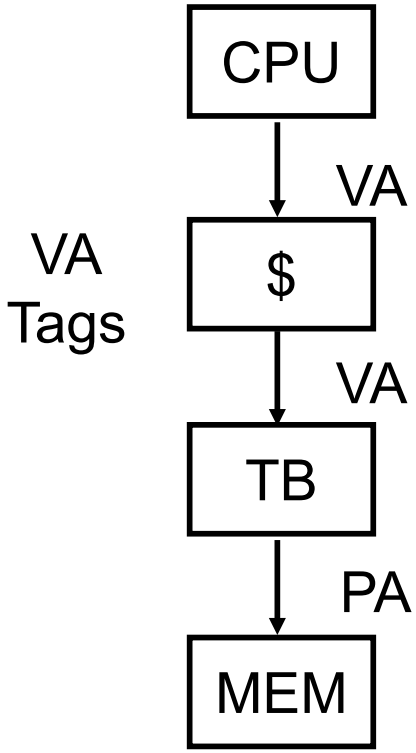
Avoiding address translation during cache indexing

- Two tasks: indexing the cache and comparing addresses
- virtually vs. physically addressed cache
  - –virtual cache: use virtual address (VA) for the cache
  - –physical cache: use physical address (PA) after translating virtual address
- Challenges to virtual cache
  1. **Protection**: page-level protection (RW/RO/Invalid) must be checked
     - –It's checked as part of the virtual to physical address translation
     - –solution: an addition field to copy the protection information from TLB and check it on every access to the cache
  2. **context switching**: same VA of different processes refer to different PA, requiring the cache to be flushed
     - –solution: increase width of cache address tag with process-identifier tag (PID)
  3. **Synonyms or aliases**: two different VA for the same PA
     - –inconsistency problem: two copies of the same data in a virtual cache
     - –hardware ***antialiasing*** solution: guarantee every cache block a unique PA
       - –Alpha 21264: check all possible locations. If one is found, it is invalidated
     - –software ***page-coloring*** solution: forcing aliases to share some address bits
       - –Sun's Solaris: all aliases must be identical in last 18 bits => no duplicate PA
  4. **I/O**: typically use PA, so need to interact with cache (see Section 5.12)

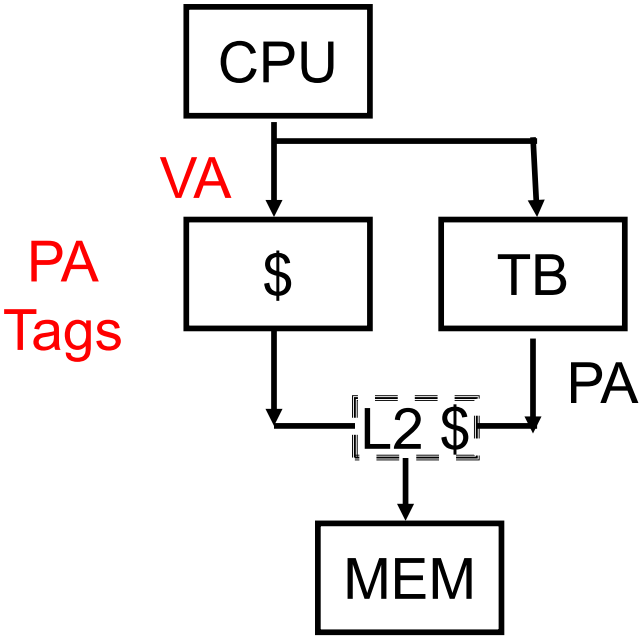
Virtually indexed, physically tagged cache



Conventional Organization



Virtually Addressed Cache
Translate only on miss
Synonym Problem



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

Summary of the 6 Basic Cache Optimization Techniques (Textbook B.3)

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4 L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

Figure B.18 Summary of basic cache optimizations showing impact on cache performance and complexity for the techniques in this appendix. Generally a technique helps only one factor. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

Advanced Cache Optimization Techniques

1. ***Reducing the hit time***—Small and simple first-level caches and way-prediction. Both techniques also generally decrease power consumption.
2. ***Increasing cache bandwidth***—Pipelined caches, multibanked caches, and nonblocking caches. These techniques have varying impacts on power consumption.
3. ***Reducing the miss penalty***—Critical word first and merging write buffers. These optimizations have little impact on power.
4. ***Reducing the miss rate***—Compiler optimizations. Obviously any improvement at compile time improves power consumption.
5. ***Reducing the miss penalty or miss rate via parallelism***—Hardware prefetching and compiler prefetching. These optimizations generally increase power consumption, primarily due to prefetched data that are unused.

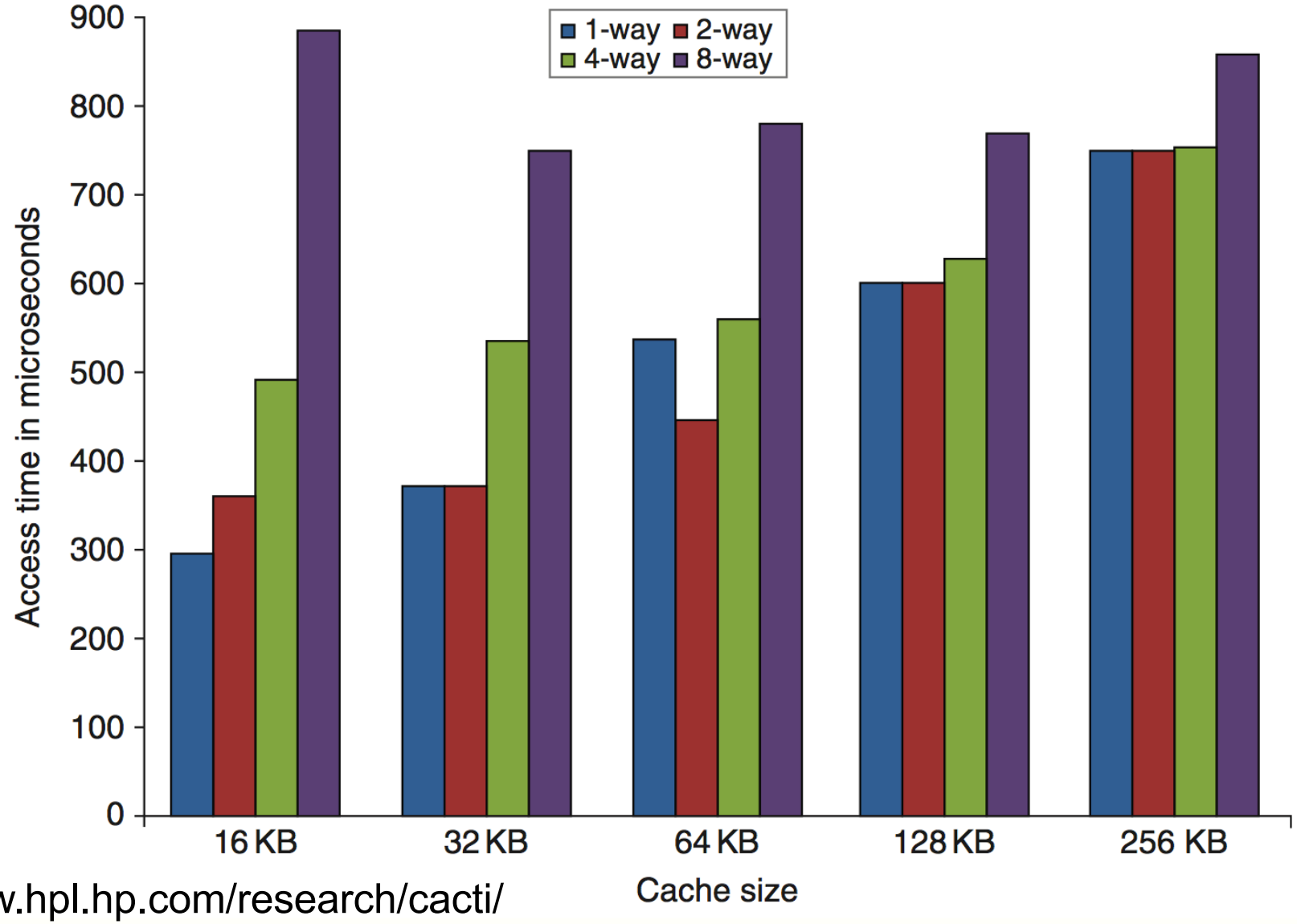
- Increase hardware complexity
- Require sophisticated compiler transformation

1. Small and Simple First-level Caches

- Cache-hit critical path, three steps:
  1. addressing the tag memory using the index portion of the address,
  2. comparing the read tag value to the address, and
  3. setting the multiplexor to choose the correct data item if the cache is set associative.
- Guideline: smaller hardware is faster, Small data cache and thus fast clock rate
  - size of the L1 caches has recently increased either slightly or not at all.
    - Alpha 21164 has 8KB Instruction and 8KB data cache + 96KB second level cache
    - E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
  - Also L2 cache small enough to fit on chip with processor ⇒ avoids time penalty of going off chip
- Guideline: simpler hardware is faster
  - Direct-mapped caches can overlap the tag check with the transmission of the data, effectively reducing hit time.
  - Lower levels of associativity will usually reduce power because fewer cache lines must be accessed.

- General design: small and simple cache for 1st-level cache
  - Keeping the tags on chip and the data off chip for 2nd-level caches
  - One emphasis is on fast clock time while hiding L1 misses with dynamic execution and using L2 caches to avoid going to memory

Cache Size → AMAT



http://www.hpl.hp.com/research/cacti/

Figure 2.3 Access times generally increase as cache size and associativity are increased. These data come from the CACTI model 6.5 by Tarjan, Thoziyoor, and Jouppi

Cache Size → Power Consumption

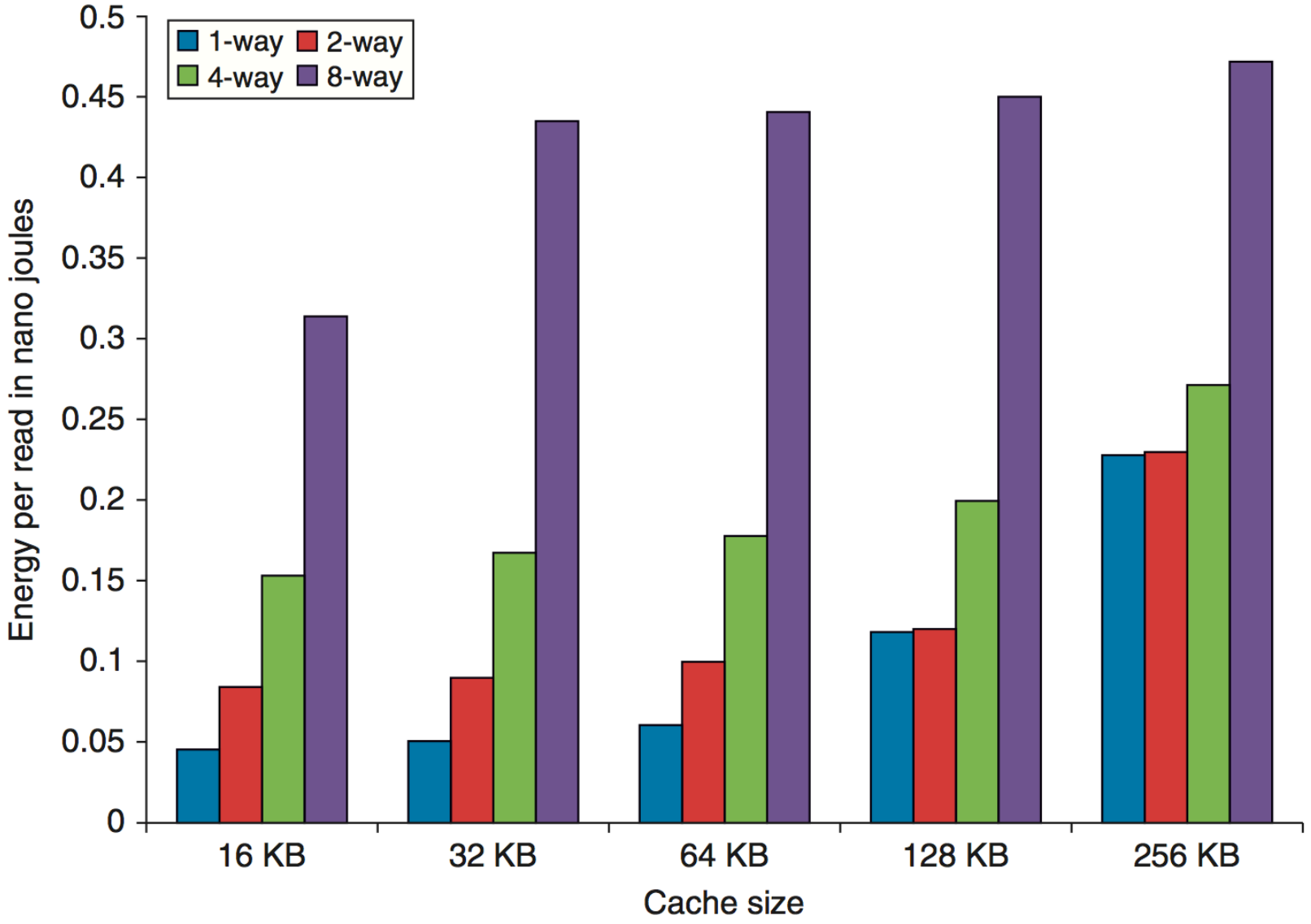


Figure 2.4 **Energy consumption per read increases as cache size and associativity are increased.** As in the previous figure, CACTI is used for the modeling with the same

Examples

Example Using the data in Figure B.8 in Appendix B and Figure 2.3, determine whether a 32 KB four-way set associative L1 cache has a faster memory access time than a 32 KB two-way set associative L1 cache. Assume the miss penalty to L2 is 15 times the access time for the faster L1 cache. Ignore misses beyond L2. Which has the faster average memory access time?

Answer Let the access time for the two-way set associative cache be 1. Then, for the two-way cache:

$$\begin{aligned}\text{Average memory access time}_{\text{2-way}} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \\ &= 1 + 0.038 \times 15 = 1.38\end{aligned}$$

For the four-way cache, the access time is 1.4 times longer. The elapsed time of the miss penalty is 15/1.4 = 10.1. Assume 10 for simplicity:

$$\begin{aligned}\text{Average memory access time}_{\text{4-way}} &= \text{Hit time}_{\text{2-way}} \times 1.4 + \text{Miss rate} \times \text{Miss penalty} \\ &= 1.4 + 0.037 \times 10 = 1.77\end{aligned}$$

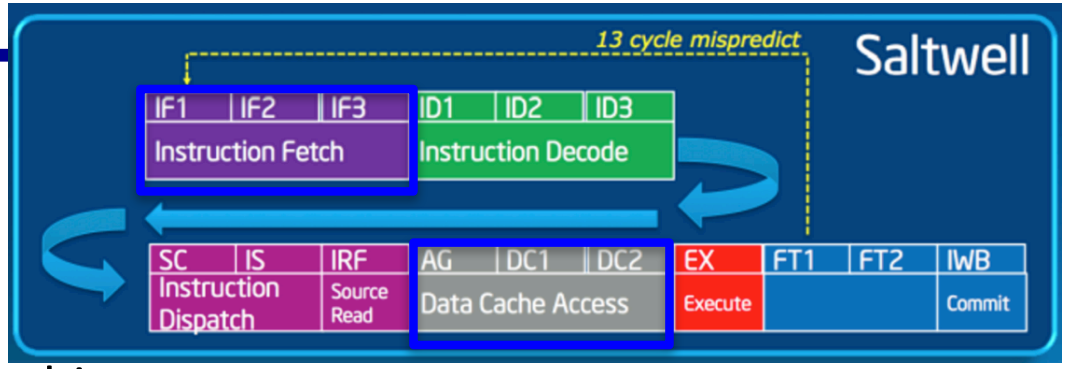
Clearly, the higher associativity looks like a bad trade-off; however, since cache access in modern processors is often pipelined, the exact impact on the clock cycle time is difficult to assess.

2. Fast Hit Times Via Way Prediction

- How to combine fast hit time of direct-mapped with lower conflict misses of 2-way SA cache?
- Way prediction: keep extra bits in cache to predict "way" (block within set) of next access
  - Multiplexer set early to select desired block; only 1 tag comparison done that cycle (in parallel with reading data)
  - Miss ⇒ check other blocks for matches in next cycle

- Accuracy ≈ 85%
- Drawback: CPU pipeline harder if hit time is variable-length

3. Increasing Cache Bandwidth by Pipelining



- Simply to pipeline cache access
  - Multiple clock cycle for 1st-level cache hit
- Advantage: fast cycle time and slow hit
- Example: accessing instructions from I-cache
  - Pentium: **1 clock cycle**, mid-1990
  - Pentium Pro ~ Pentium III: **2 clocks**, mid-1990 to 2000
  - Pentium 4: **4 clocks**, 2000s
  - Intel Core i7: **4 clocks**
- Drawback: Increasing the number of pipeline stages leads to
  - greater penalty on mispredicted branches and
  - more clock cycles between the issue of the load and the use of the data
- Note that it increases the bandwidth of instructions rather than decreasing the actual latency of a cache hit

4. Increasing Cache Bandwidth with Non-Blocking Caches

- Non-blocking or lockup-free cache allows continued cache hits during miss
  - Requires F/E bits on registers or out-of-order execution
  - Requires multi-bank memories
- Hit under miss reduces effective miss penalty by working during miss vs. ignoring CPU requests
- Hit under multiple miss or miss under miss further lowers effective miss penalty by overlapping multiple misses
  - Significantly increases complexity of cache controller since can be many outstanding memory accesses
  - Requires multiple memory banks
  - Penium Pro allows 4 outstanding memory misses

Effectiveness of Non-Blocking Cache

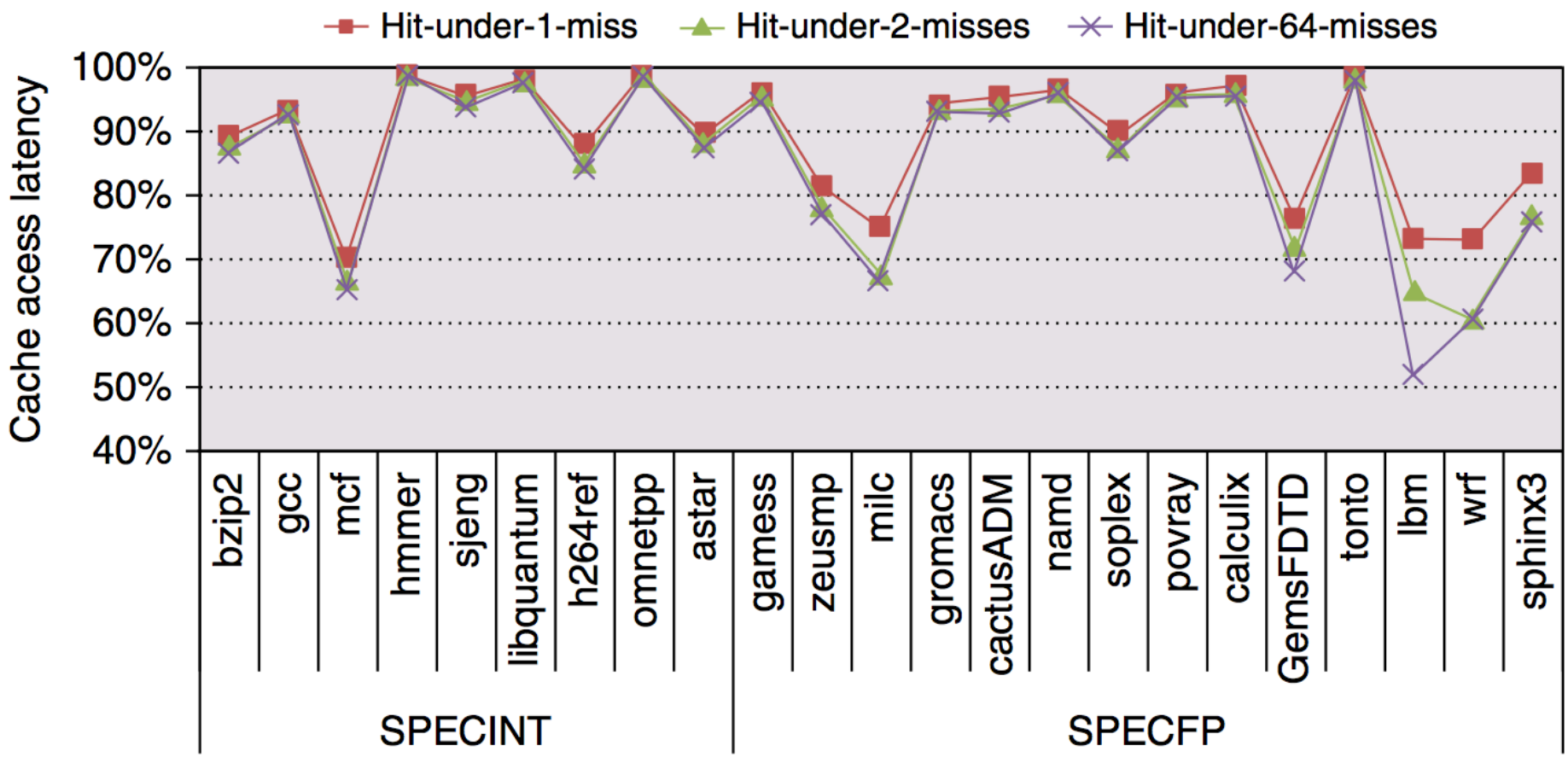


Figure 2.5 The effectiveness of a nonblocking cache is evaluated by allowing 1, 2, or 64 hits under a cache miss with 9 SPECINT (on the left) and 9 SPECFP (on the right) benchmarks. The data memory system modeled after the Intel i7 consists of a 32KB L1 cache with a four cycle access latency. The L2 cache (shared with instructions) is 256 KB with a 10 clock cycle access latency. The L3 is 2 MB and a 36-cycle access latency. All the caches are eight-way set associative and have a 64-byte block size. Allowing one hit

Performance Evaluation of Non-Blocking Caches

- A cache miss does not necessarily stall the processor
  - difficult to judge the impact of any single miss and hence to calculate the average memory access time.
- The effective miss penalty is the non-overlapped time that the processor is stalled.
  - not the sum of the misses

- The benefit of nonblocking caches is complex, depends on
  - the miss penalty when there are multiple misses,
  - the memory reference pattern, and
  - how many instructions the processor can execute with a miss outstanding.
- For out-of-order processors
  - Check textbook

5. Increasing Cache Bandwidth Via Multiple Banks

- Rather than treating cache as single monolithic block, divide into independent banks to support simultaneous accesses
  - The Arm Cortex-A8 supports one to four banks in its L2 cache;
  - the Intel Core i7 has four banks in L1 (to support up to 2 memory accesses per clock), and the L2 has eight banks.

Sequential Interleaving

- Works best when accesses naturally spread across banks ⇒
  - mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is sequential interleaving
  - Spread block addresses sequentially across banks
  - E,g, bank *i* has all blocks with address *i* modulo *n*

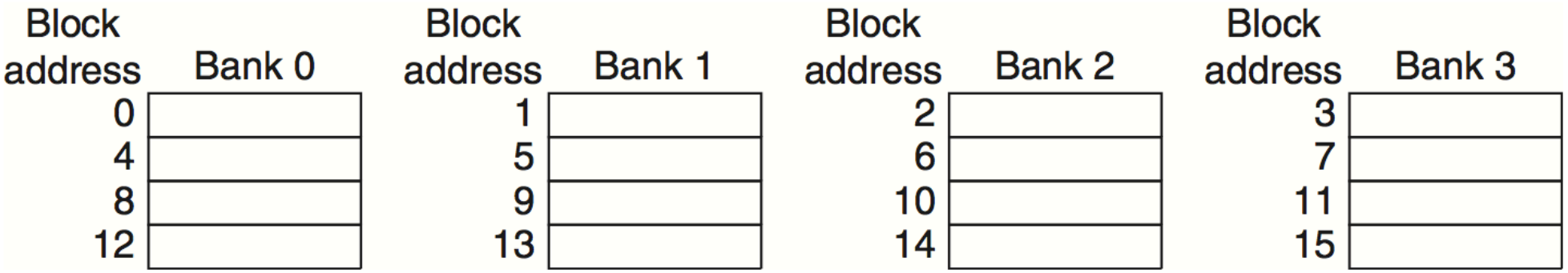


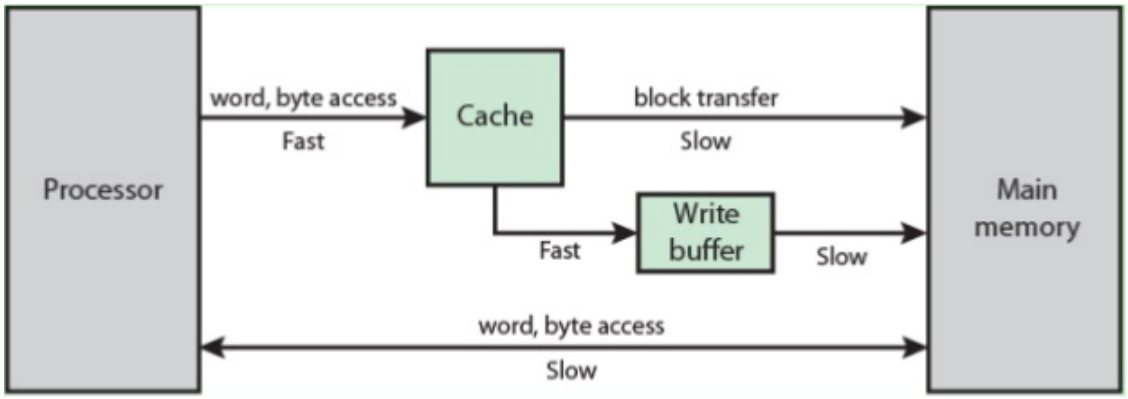
Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

6. Reduce Miss Penalty: Early Restart and Critical Word First

- Don't wait for full block before restarting CPU
- Critical Word First—Request missed word from memory first, send it to CPU right away; let CPU continue while filling rest of block
  - Large blocks more popular today ⇒ Critical Word 1st widely used
- Early restart—As soon as requested word of block arrives, send to CPU and continue execution
  - Spatial locality ⇒ tend to want next sequential word, so may still pay to get that one

7. Merging Write Buffer to Reduce Miss Penalty

- Write buffer lets processor continue while waiting for write to complete



- Merging write buffer:
  - If buffer contains modified blocks, addresses can be checked to see if new data matches that of some write buffer entry
  - If so, new data combined with that entry
- For sequential writes in write-through caches, increases block size of write (more efficient)
- Sun T1 (Niagara) and many others use write merging

Merge Write Buffer Example

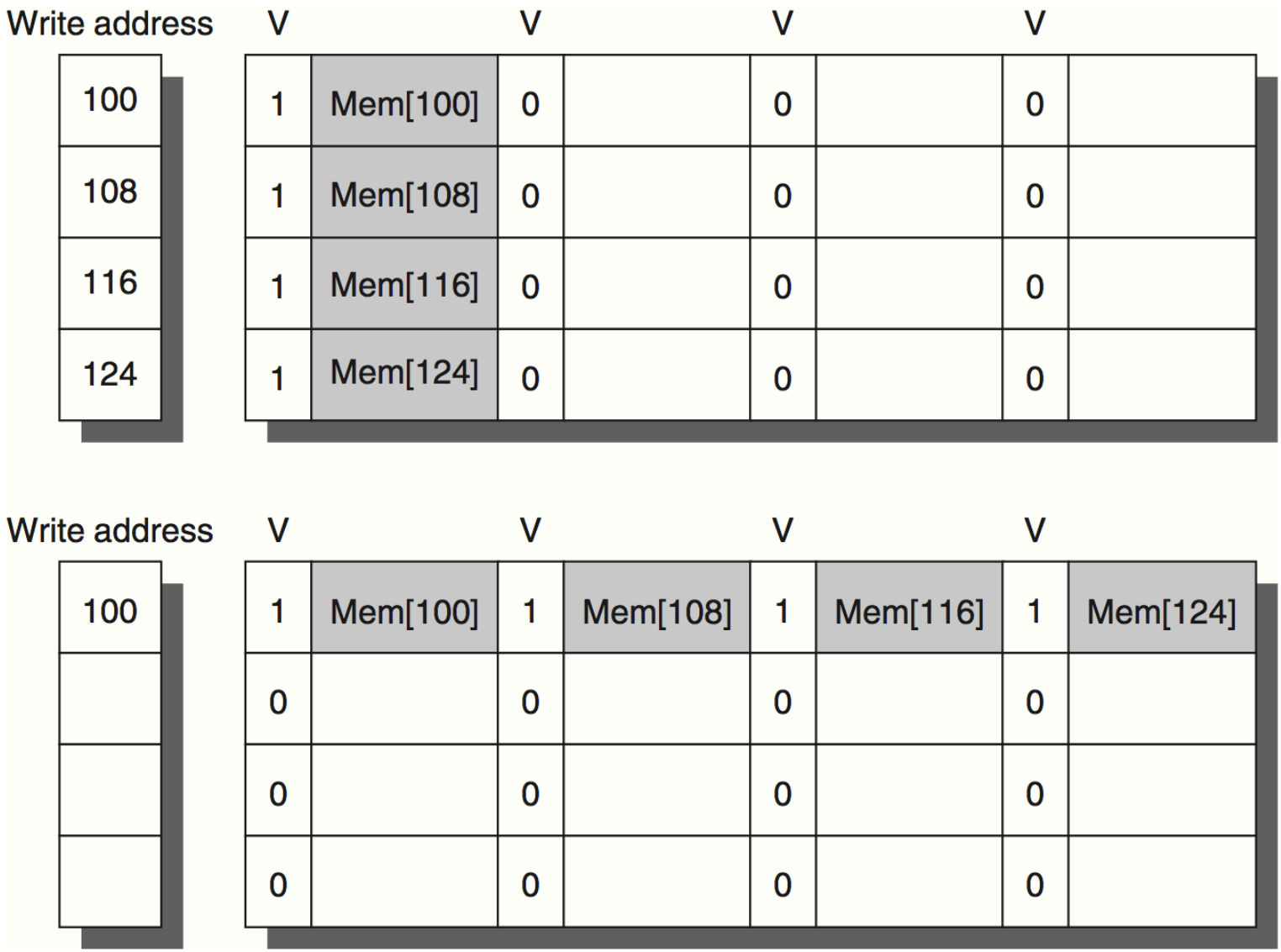
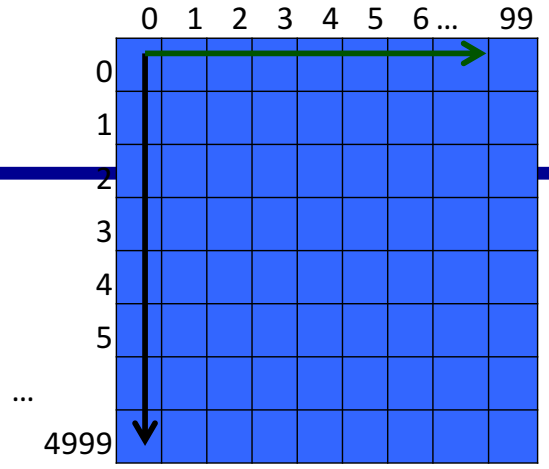


Figure 2.7 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer

8. Reducing Misses by Compiler Optimizations Software-only Approach

- McFarling [1989] reduced misses by 75% in software on 8KB direct-mapped cache, 4 byte blocks
- Instructions
  - Reorder procedures in memory to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - Loop interchange: Change nesting of loops to access data in memory order
  - Blocking: Improve temporal locality by accessing blocks of data repeatedly vs. going down whole columns or rows
  - Merging arrays: Improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop fusion: Combine 2 independent loops that have same looping and some variable overlap

Loop Interchange Example




```
/* 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];
```

Sequence of access:
X[0][0], X[1][0], X[2][0], …

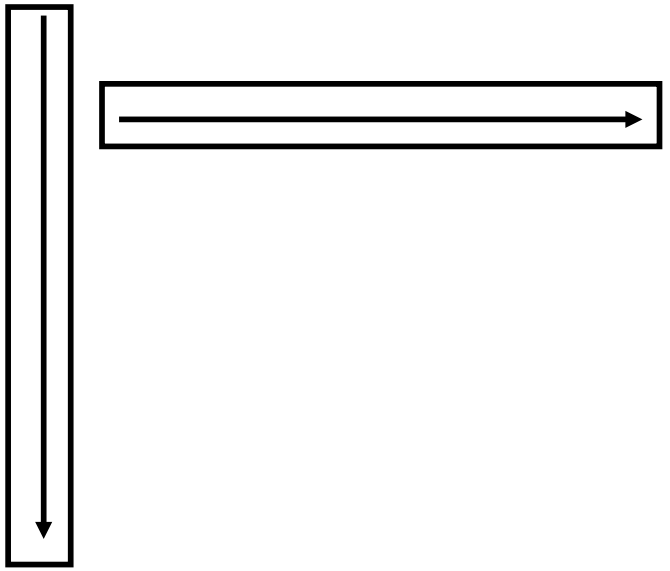
```
/* After */
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
```

Sequence of access:
X[0][0], X[0][1], X[1][2], …

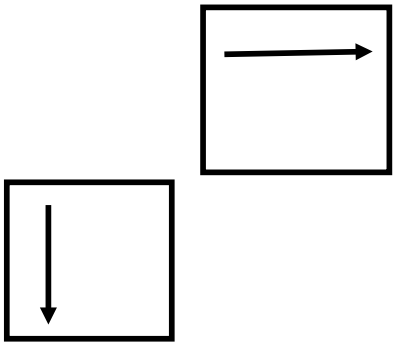
Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Blocking Example

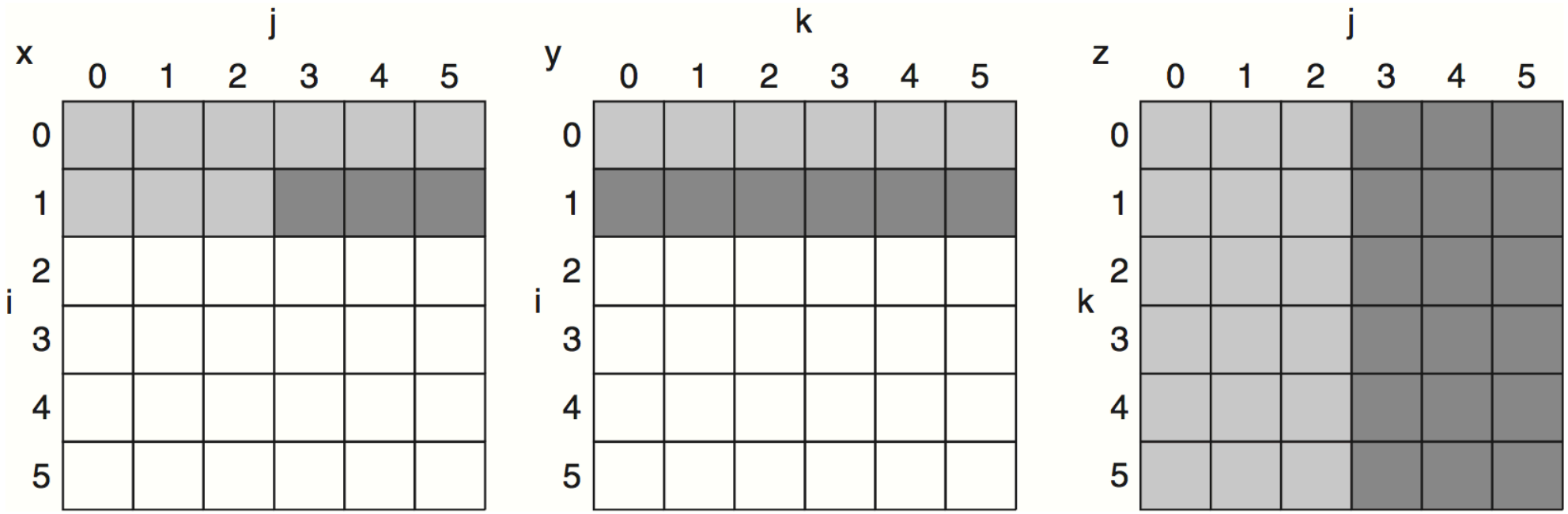
```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
      {r = 0;
       for (k = 0; k < N; k = k+1){
          r = r + y[i][k]*z[k][j];};
       x[i][j] = r;
      };
```



- 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:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

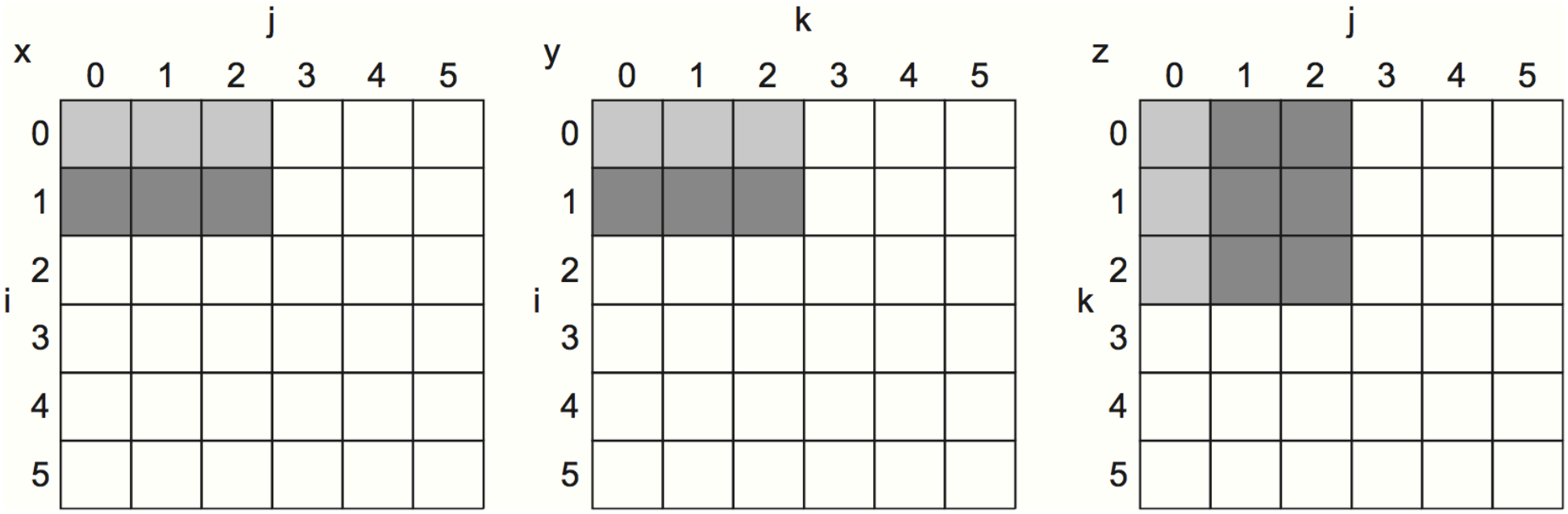


Array Access in Matrix Multiplication



2.8 A snapshot of the three arrays x, y, and z when *N* = 6 and i = 1. The age of accesses to the arr
is indicated by shade: white means not yet touched, light means older accesses, and dark means
es. Compared to Figure 2.9, elements of y and z are read repeatedly to calculate new elements of x. Th
, j, and k are shown along the rows or columns used to access the arrays.

Array Access for Blocking/Tiling Transformation



ıre 2.9 The age of accesses to the arrays x, y, and z when *B* = 3. Note that, in contrast to Figure 2.8, a
ıber of elements is accessed.

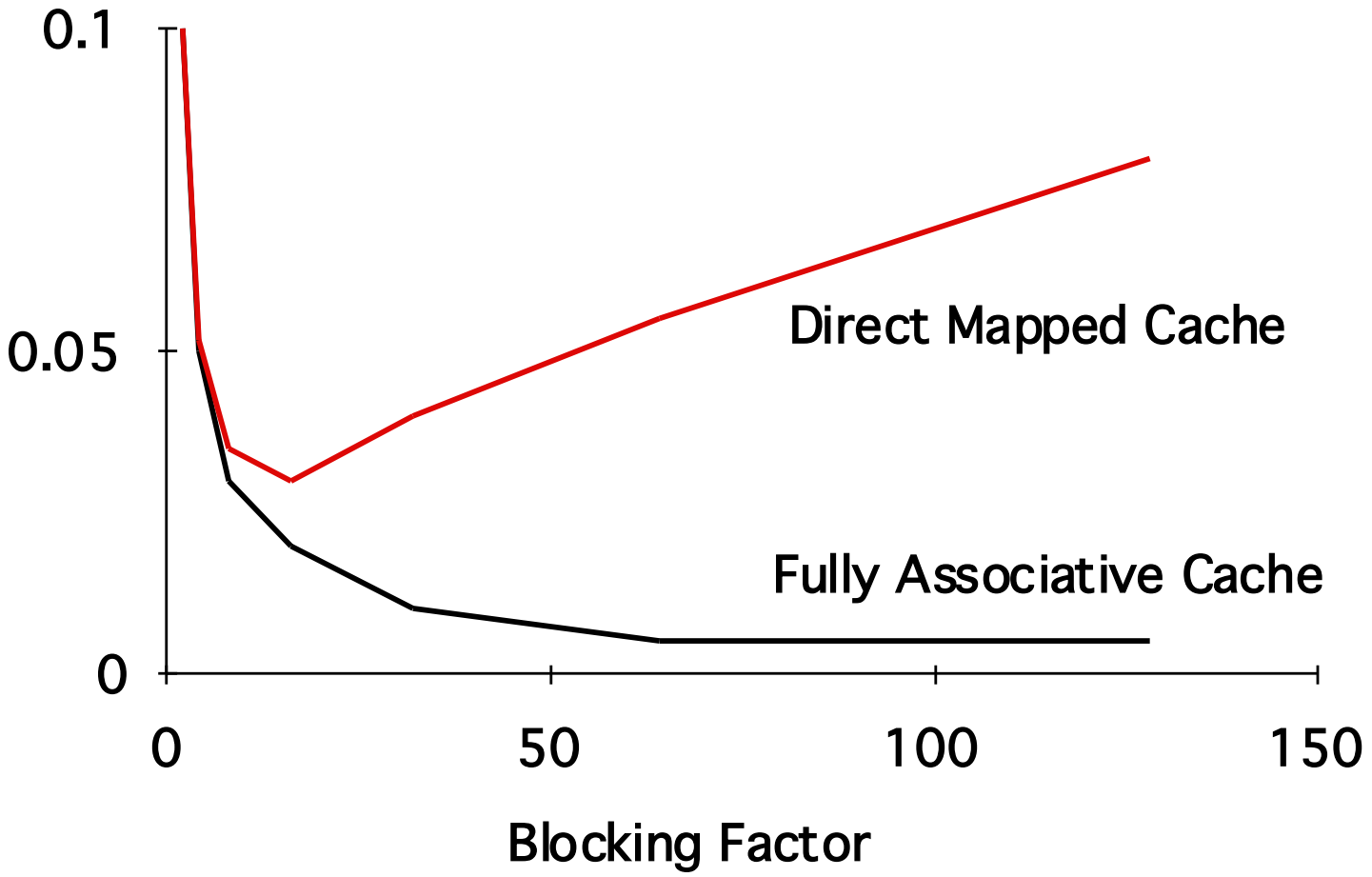
Blocking Example

```
/* 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*
- Capacity misses from $2N^3 + N^2$ to $2N^3/B + N^2$
- Reduce conflict misses too?

Reducing Conflict Misses by Blocking

- Conflict misses in caches not FA vs. blocking size
  - Lam et al [1991]: Blocking factor of 24 had 1/5 the misses vs. 48 despite both fitting in cache



Merging/Spliting Arrays Example

Array of Struct or Struct of Array

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

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

Reduce conflicts between val & key; improve spatial locality

Loop Fusion Example

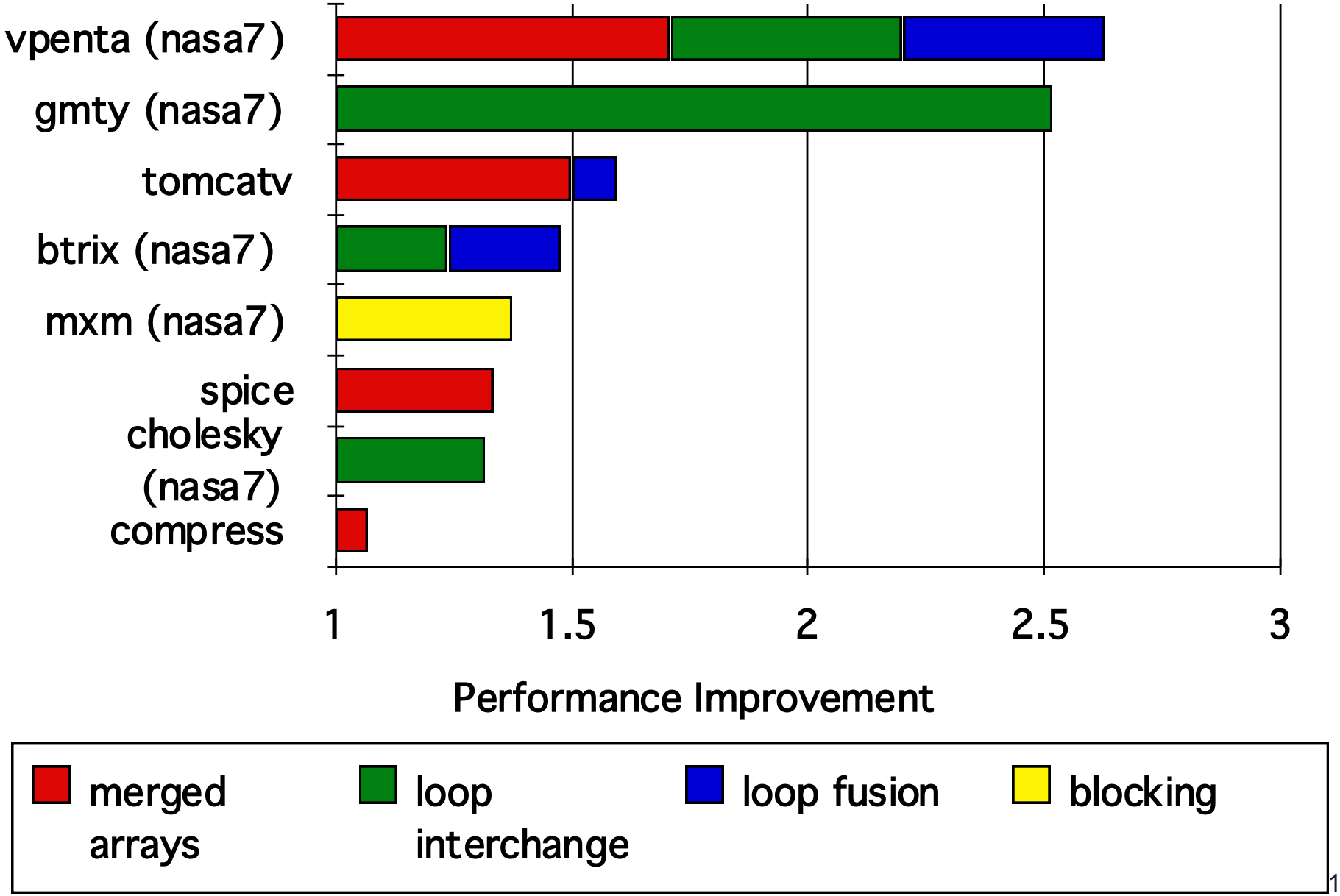
```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
    a[i][j] = 1/b[i][j] * c[i][j];

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];}
```

2 misses per access to `a` & `c` vs. one miss per access; improve spatial locality

Summary of Compiler Optimizations to Reduce Cache Misses (by hand)



9. Reducing Misses by Hardware Prefetching of Instructions & Data

- **Hardware prefetch items before the processor requests them.**
  - Both instructions and data can be prefetched,
  - Either directly into the caches or into an external buffer that can be more quickly accessed than main memory.
- Instruction prefetching
  - Typically, CPU fetches 2 blocks on miss: requested and next
  - Requested block goes in instruction cache, prefetched goes in instruction stream buffer
- Data prefetching
  - Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages
  - Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes
  - The Intel Core i7 supports hardware prefetching into both L1 and L2 with the most common case of prefetching being accessing the next line.
    - Simpler than before.

Hardware Prefetching

- Relies on utilizing memory bandwidth that otherwise would be unused
  - If it interferes with demand misses it can actually lower performance.
  - When prefetched data are not used or useful data are displaced, prefetching will have a very negative impact on power.

Hardware Prefetching

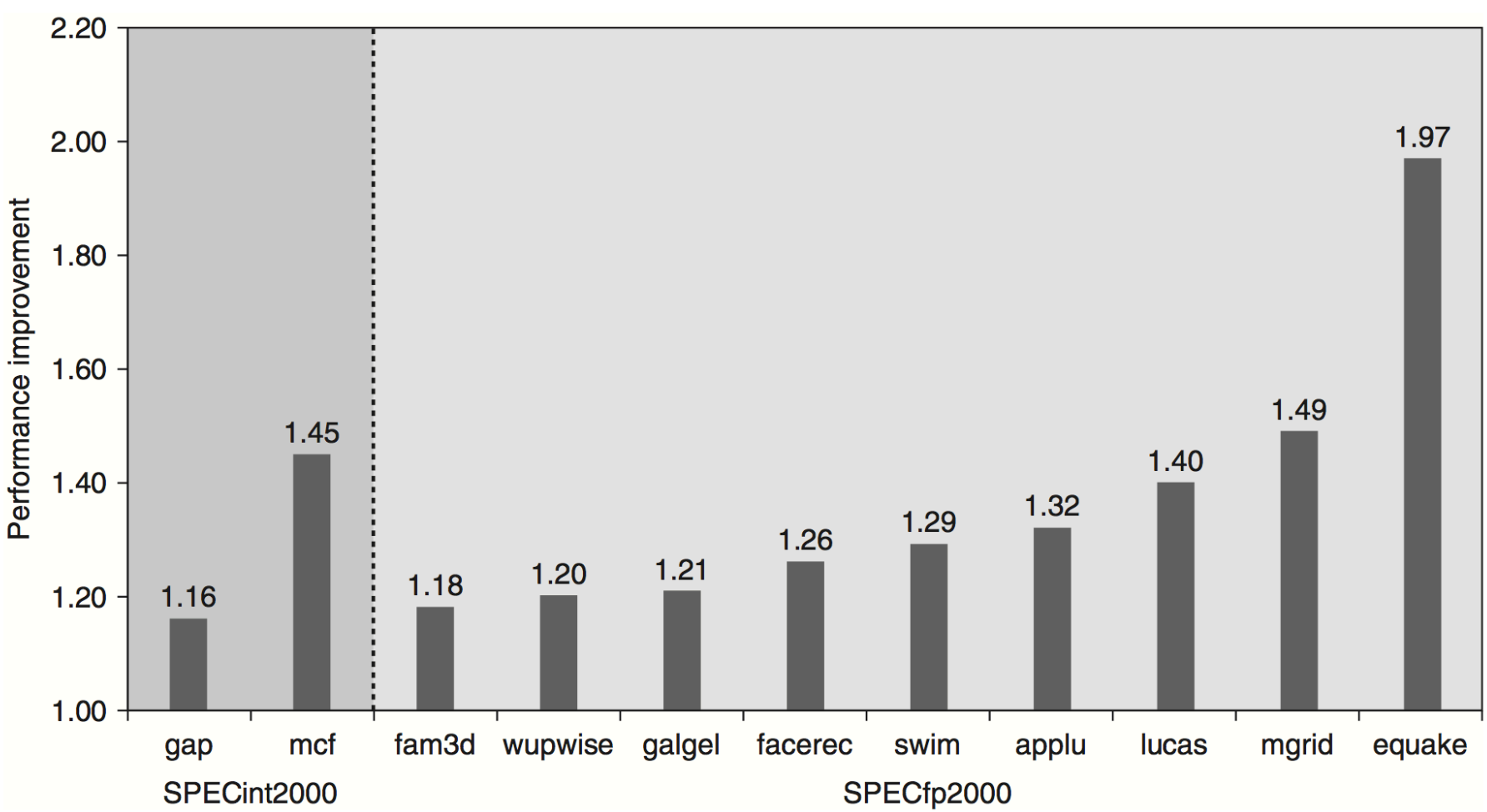


Figure 2.10 Speedup due to hardware prefetching on Intel Pentium 4 with hardware prefetching turned on for 2 of 12 SPECint2000 benchmarks and 9 of 14 SPECfp2000 benchmarks. Only the programs that benefit the most from prefetching are shown; prefetching speeds up the missing 15 SPEC benchmarks by less than 15% [Singhal 2004].

10. Compiler-Controlled Prefetching to Reduce Miss Penalty or Miss Rate

Compiler to insert prefetch instructions to request data before the processor needs it.

- Data prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  - Special prefetching instructions cannot cause faults; form of speculative execution

- Prefetch instructions take time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces problem of issue bandwidth

Compiler-Controlled Prefetching Example

- Page #93

For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided by prefetching. Let's assume we have an 8 KB direct-mapped data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of a and b are 8 bytes long since they are double-precision floating-point arrays. There are 3 rows and 100 columns for a and 101 rows and 3 columns for b. Let's also assume they are not in the cache at the start of the program.

```
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];
```

Compiler-Controlled Prefetching Example

```
for (j = 0; j < 100; j = j+1) {
        prefetch(b[j+7][0]);
        /* b(j,0) for 7 iterations later */
        prefetch(a[0][j+7]);
        /* a(0,j) for 7 iterations later */
        a[0][j] = b[j][0] * b[j+1][0];};
for (i = 1; i < 3; i = i+1)
        for (j = 0; j < 100; j = j+1) {
                prefetch(a[i][j+7]);
                /* a(i,j) for +7 iterations */
                a[i][j] = b[j][0] * b[j+1][0];}
```

Prefetching in GCC Compiler

Built-in Function: void **__builtin_prefetch** (*const void *addr, ...*)

This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to `__builtin_prefetch` into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions are generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of *addr* is the address of the memory to prefetch. There are two optional arguments, *rw* and *locality*. The value of *rw* is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value *locality* must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

```
for (i = 0; i < n; i++)
  {
    a[i] = a[i] + b[i];
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* ... */
  }
```

Data prefetch does not generate faults if *addr* is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` does not fault if `p->next` is not a valid address, but evaluation faults if `p` is not a valid address.

If the target does not support data prefetch, the address expression is evaluated if it includes side effects but no other code is generated and GCC does not issue a warning.

https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

Summary of the 10 Advanced Cache Optimization Techniques

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Review on Cache Performance and Optimizations

Summary of the 6 Basic Cache Optimization Techniques (Textbook B.3)

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4 L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

Figure B.18 Summary of basic cache optimizations showing impact on cache performance and complexity for the techniques in this appendix. Generally a technique helps only one factor. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

Summary of the 10 Advanced Cache Optimization Techniques

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Review of Cache Optimizations

- 3 C's of cache misses
  - Compulsory, Capacity, Conflict
- Write policies
  - Write back, write-through, write-allocate, no write allocate
- Multi-level cache hierarchies reduce miss penalty
  - 3 levels common in modern systems (some have 4!)
  - Can change design tradeoffs of L1 cache if known to have L2
- Prefetching: retrieve memory data before CPU request
  - Prefetching can waste bandwidth and cause cache pollution
  - Software vs hardware prefetching
- Software memory hierarchy optimizations
  - Loop interchange, loop fusion, cache tiling

Design Guideline

- Cache block size: 32 or 64 bytes
  - Fixed size across cache levels
- Cache sizes (per core):
  - L1: Small and fastest for low hit time, 2K to 62K each for D$ and I$ separated
  - L2: Large and faster for low miss rate, 256K – 512K for combined D$ and I$ combined
  - L3: Large and fast for low miss rate: 1MB – 8MB for combined D$ and I$ combined
- Associativity
  - L1: directed, 2/4 way
  - L2: 4/8 way
- Banked, pipelined and no-blocking access

For Software Development

- Explicit or compiler-controlled prefetching
  - Insert prefetching call.
- Explicit or compiler-assisted code optimization for cache performance
  - Loop transformation (interchange, blocking, etc)

Memory Hierarchy Performance, B.2 in Textbook

- Two indirect performance measures have waylaid many a computer designer.
  - *Instruction count* is independent of the hardware;
  - *Miss rate* is independent of the hardware.

$$CPU\ Time = IC * \left(CPI_{Execution} + \frac{Memory\ Accesses}{Instruction} \times Miss\ Rate \times Miss\ Penalty\right) \times Clock\ Cycle\ Time$$

- A better measure of memory hierarchy performance is the *Average Memory Access Time* (AMAT) per instructions

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Equations (B-19, B-21)

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

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

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}$$

Review

- Finished
  - Appendix A, Instruction Set Principles
  - Appendix B, Review of Memory Hierarchy
  - Appendix C, Pipelining: Basic and Intermediate Concepts
  - Chapter 1, Fundamentals of Quantitative Design and Analysis
  - Chapter 2, Memory Hierarchy Design
- Assignment #3 posted, due 10/29
- Second Half
  - Chapter 3, Instruction-Level Parallelism and Its Exploitation
  - Chapter 4, Data-Level Parallelism in Vector, SIMD, and GPU Architectures
  - Chapter 5, Thread-Level Parallelism
  - Chapter 7, Domain-Specific Architectures

Feedback

- Lecture
  - Amount and depth of content
- Assignment
  - Amount and depth
  - Amount and depth of hand-on work and programming
  - Extra questions and bonus points
- Midterm
  - Depth and amount

- Nov 12 and Nov 14 classes (regular class on Nov 19th)
  - Reschedule one to Nov 9 (?)
  - Remote via webconnect (?)