Exercises

Solutions to "starred" exercises appear in Appendix B.

✪ 5.1 [15/15/12/12] <5.1, 5.2> Let's try to show how you can make *unfair* benchmarks. Here are two machines with the same processor and main memory but different cache organizations. Assume that both processors run at 2 GHz, have a CPI of 1, and have a cache (read) miss time of 100 ns. Further, assume that writing a 32-bit word to main memory requires 100 ns (for the write-through cache), and that writing a 32-byte block requires 200 ns (for the write-back cache). The caches are unified—they contain both instructions and data, and each cache has a total capacity of 64 KB, not including tags and status bits.

The cache on system A is two-way set associative and has 32-byte blocks. It is write through and does not allocate a block on a write miss.

The cache on system B is direct mapped and has 32-byte blocks. It is write back and allocates a block on a write miss.

a. [15] <5.1, 5.2> Describe a program that makes system A run as fast as possible relative to system B's speed.

b. [15] <5.1, 5.2> Describe a program that makes system B run as fast as possible relative to system A's speed.

c. [12] <5.1, 5.2> How much faster is the program in part (a) on system A as compared to system B?

d. [12] <5.1, 5.2> How much faster is the program in part (b) on system B as compared to system A?

5.2 [15/10/12/12/12/12/12/12/12/12/12] <5.3, 5.4> In this exercise, we will run a program to evaluate the behavior of a memory system. The key is having accurate timing and then having the program stride through memory to invoke different levels of the hierarchy. Below is the code in C for UNIX systems. The first part is a procedure that uses a standard UNIX utility to get an accurate measure of the user CPU time; this procedure may need to change to work on some systems. The second part is a nested loop to read and write memory at different strides and cache sizes. To get accurate cache timing, this code is repeated many times. The third part times the nested loop overhead only so that it can be subtracted from overall measured times to see how long the accesses were. The last part prints the time per access as the size and stride varies. You may need to change `CACHE_MAX` depending on the question you are answering and the size of memory on the system you are measuring. The code below was taken from a program written by Andrea Dusseau of U.C. Berkeley, and was based on a detailed description found in Saavedra-Barrera [1992].

```
#include <stdio.h>
#include <sys/times.h>
#include <sys/types.h>
#include <time.h>
#define CACHE_MIN (1024) /* smallest cache */
```

```
#define CACHE_MAX (1024*1024) /* largest cache */
#define SAMPLE 10 /* to get a larger time sample */
#ifndef CLK_TCK
#define CLK_TCK 60 /* number clock ticks per second */
#endif
int x[CACHE_MAX]; /* array going to stride through */
double get_seconds() { /* routine to read time */
    struct tms rusage;
    times(&rusage); /* UNIX utility: time in clock ticks */
    return (double) (rusage.tms_utime)/CLK_TCK;
}
void main() {
int register i, index, stride, limit, temp;
int steps, tsteps, csize;
double sec0, sec; /* timing variables */
for (csize=CACHE_MIN; csize <= CACHE_MAX; csize=csize*2)
    for (stride=1; stride <= csize/2; stride=stride*2) {
        sec = 0; /* initialize timer */
        limit = csize-stride+1; /* cache size this loop */
        steps = 0;
            do { /* repeat until collect 1 second */
        sec0 = get_seconds(); /* start timer */
        for (i=SAMPLE*stride;i!=0;i=i-1) /* larger sample */
            for (index=0; index < limit; index=index+stride)
                x[index] = x[index] + 1; /* cache access */
        steps = steps + 1; /* count while loop iterations */
        sec = sec + (get_seconds() - sec0);/* end timer */
        } while (sec < 1.0); /* until collect 1 second */

        /* Repeat empty loop to subtract loop overhead */
        tsteps = 0; /* used to match no. while iterations */
            do { /* repeat until same no. iterations as above */
        sec0 = get_seconds(); /* start timer */
        for (i=SAMPLE*stride;i!=0;i=i-1) /* larger sample */
           for (index=0; index < limit; index=index+stride)
                temp = temp + index; /* dummy code */
        tsteps = tsteps + 1; /* count while iterations */
        sec = sec - (get_seconds() - sec0);/* - overhead */
        } while (tsteps<steps); /* until = no. iterations */
        printf("Size:%7d Stride:%7d read+write:%14.0f ns\n",
            csize*sizeof(int), stride*sizeof(int), (double)
        sec*1e9/(steps*SAMPLE*stride*((limit-1)/stride+1)));
    }; /* end of both outer for loops */
}
```

The program above assumes that program addresses track physical addresses, which is true on the few machines that use virtually addressed caches, such as the

Alpha 21264. In general, virtual addresses tend to follow physical addresses shortly after rebooting, so you may need to reboot the machine in order to get smooth lines in your results.

To answer the questions below, assume that the sizes of all components of the memory hierarchy are powers of 2.

a. [15] <5.3, 5.4> Plot the experimental results with elapsed time on the *y*-axis and the memory stride on the *x*-axis. Use logarithmic scales for both axes, and draw a line for each cache size.

b. [10] <5.3, 5.4> How many levels of cache are there?

c. [12] <5.3, 5.4> What are the overall size and block size of the first-level cache? *Hint:* Assume that the size of the page is much larger than the size of a block in a second-level cache (if there is one), and the size of a second-level cache block is greater than or equal to the size of a block in a first-level cache.

d. [12] <5.3, 5.4> What are the overall size and block size of the second-level cache, if there is one?

e. [12] <5.3, 5.4> What is the associativity of the first-level cache? What is the associativity of the second-level cache, if there is one?

f. [12] <5.3, 5.4> What is the system page size?

g. [12] <5.3, 5.4> How many entries are there in the TLB?

h. [12] <5.3, 5.4> What are the miss penalties for the first-level cache and (if present) second-level cache?

i. [12] <5.3, 5.4> What is the time for a page fault to secondary memory (i.e., disk)? *Hint:* Disk accesses have latencies measured in milliseconds.

j. [12] <5.3, 5.4> What is the miss penalty for the TLB?

k. [12] <5.3, 5.4> Is there anything else you have discovered about the memory hierarchy from these measurements?

5.3 [10/10/10] <5.2> Figure 5.59 shows the output from running the program in Exercise 5.2 on a Sun Blade 1000 server, which has separate L1 instruction and data caches and a unified L2 cache.

a. [10] <5.2> How big is the cache?

b. [10] <5.2> What is the block size in the cache?

c. [15] <5.2> What is the miss penalty for the L1 cache? L2 cache? Additional memory hierarchy level(s)?

d. [20] <5.2> What might explain the access time behavior shown for very large strides?

5.4 [10/10/10/10/15/20] <5.2> You are building a system around a processor with in-order execution that runs at 1.1 GHz and has a CPI of 0.7 excluding memory accesses. The only instructions that read or write data from memory are loads (20% of all instructions) and stores (5% of all instructions).

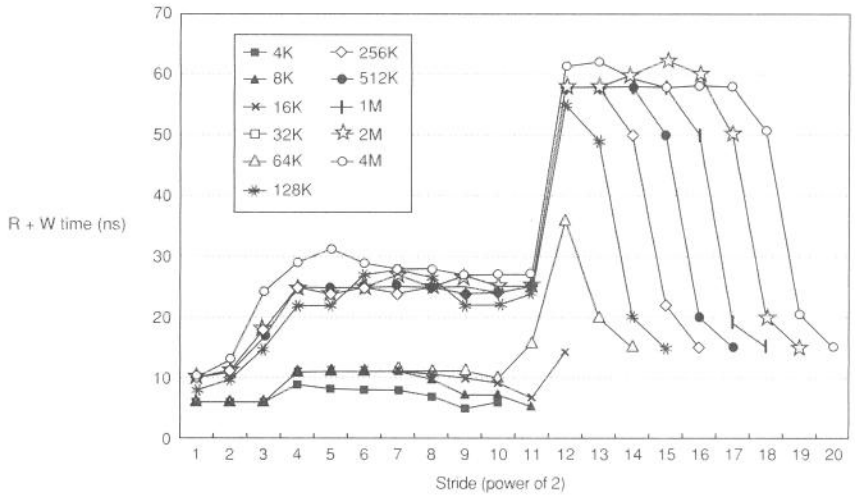

Figure 5.59 Results of running program in Exercise 5.2 on a Sun Blade 1000.

The memory system for this computer is composed of a split L1 cache that imposes no penalty on hits. Both the I-cache and D-cache are direct mapped and hold 32 KB each. The I-cache has a 2% miss rate and 32-byte blocks, and the D-cache is write through with a 5% miss rate and 16-byte blocks. There is a write buffer on the D-cache that eliminates stalls for 95% of all writes.

The 512 KB write-back, unified L2 cache has 64-byte blocks and an access time of 15 ns. It is connected to the L1 cache by a 128-bit data bus that runs at 266 MHz and can transfer one 128-bit word per bus cycle. Of all memory references sent to the L2 cache in this system, 80% are satisfied without going to main memory. Also, 50% of all blocks replaced are dirty.

The 128-bit-wide main memory has an access latency of 60 ns, after which any number of bus words may be transferred at the rate of one per cycle on the 128-bit-wide 133 MHz main memory bus.

a. [10] <5.2> What is the average memory access time for instruction accesses?

b. [10] <5.2> What is the average memory access time for data reads?

c. [10] <5.2> What is the average memory access time for data writes?

d. [10] <5.2> What is the overall CPI, including memory accesses?

e. [15] <5.2> You are considering replacing the 1.1 GHz CPU with one that runs at 2.1 GHz, but is otherwise identical. How much faster does the system run with a faster processor? Assume the L1 cache still has no hit penalty, and that the speed of the L2 cache, main memory, and buses remains the same in absolute terms (e.g., the L2 cache still has a 15 ns access time and a 266 MHz bus connecting it to the CPU and L1 cache).

f. [20] <5.2> If you want to make your system run faster, which part of the memory system would you improve? Graph the change in overall system performance holding all parameters fixed except the one that you're improving. Parameters you might consider improving include L2 cache speed, bus speeds, main memory speed, and L1 and L2 hit rates. Based on these graphs, how could you best improve overall system performance with minimal cost?

5.5 [10/15/15/25] <3, 4, 5.2> Converting miss rate (misses per reference) into misses per instruction relies upon two factors: references per instruction fetched and the fraction of fetched instructions that actually commits.

a. [10] <3, 4, 5.2> The formula for misses per instruction on page 396 is written first in terms of three factors: miss rate, memory accesses, and instruction count. Each of these factors represents actual events. What is different about writing misses per instruction as miss rate times the factor *memory accesses per instruction*?

b. [15] <5.2> Speculative processors will fetch instructions that do not commit. The formula for misses per instruction on page 396 refers to misses per instruction on the execution path, that is, only the instructions that must actually be executed to carry out the program. Convert the formula for misses per instruction on page 396 into one that uses only miss rate, references per instruction fetched, and fraction of fetched instructions that commit. Why rely upon these factors rather than those in the formula on page 396?

c. [15] <3, 4, 5.2> The conversion in part (b) could yield an incorrect value to the extent that the value of the factor references per instruction fetched is not equal to the number of references for any particular instruction. Rewrite the formula of part (b) to correct this deficiency.

d. [25] <3, 4, 5.2> Simulate a SPEC95 or SPEC2000 benchmark using SimpleScalar (*www.cs.wisc.edu/~mscalar/simplescalar.html*), first using in-order execution and then using out-of-order execution. Does miss rate vary between the two executions?

✪ 5.6 [20] <5.1, 5.3> In systems with a write-through L1 cache backed by a write-back L2 cache instead of main memory, a merging write buffer can be simplified. Explain how this can be done. Are there situations where having a full write buffer (instead of the simple version you've just proposed) could be helpful?

5.7 [20] <5.3, 5.4> A cache may use a write buffer to reduce write latency and a victim cache to hold recently evicted (nondirty) blocks. Would there be any advantages to combining the two into a single piece of hardware? Would there be any disadvantages?

5.8 [20] <5.6> Improve on the compiler prefetch example found on pages 440–441. Try to eliminate both the number of extraneous prefetches and the number of nonprefetched cache misses. Calculate the performance of this refined version using the parameters in the example.

5.9 [25/25/25/25] <5.4> Use an instruction simulator and cache simulator such as SimpleScalar (*www.cs.wisc.edu/~mscalar/simplescalar.html*) to calculate the effectiveness of early restart and out-of-order fetch. What is the distribution of block offsets for the first accesses to blocks as block size increases from 4 words to 64 words by factors of 2 for the following (assume two-way set-associative caches):

a. [25] <5.4> A 128 KB instruction-only cache?

b. [25] <5.4> A 128 KB data-only cache?

c. [25] <5.4> A 256 KB unified cache?

d. [25] <5.4> A split 128/128 L1 cache with a 1 MB L2 cache? Assume that the L2 cache has 64-word blocks, and that both the L1 and L2 cache can fetch out of order.

5.10 [30/30] <3, 4, 5> Use an out-of-order, superscalar simulator such as SimpleScalar (*www.cs.wisc.edu/~mscalar/simplescalar.html*) and a simple benchmark for the following:

a. [30] <3, 4, 5> Run the benchmark and find the hit rates for a memory system with a level 1 cache ranging from 64 KB to 256 KB and a level 2 cache ranging from 512 KB to 4 MB assuming that instructions are executed in order and issued at most one per cycle (in other words, the CPU is not really out of order and superscalar).

b. [30] <3, 4, 5> Run the same program on the same range of memory systems on an out-of-order, four-way instruction issue processor with two integer units, one floating-point unit, and one memory unit. How do the hit rates vary from the in-order case?

5.11 [25] <3, 4, 5> Let's study the impact of out-of-order execution on temporal locality in an L1 data cache. Use an out-of-order, superscalar simulator such as SimpleScalar (*www.cs.wisc.edu/~mscalar/simplescalar.html*) and a simple benchmark. Then vary the out-of-orderliness by changing the size of the load and store queues and measure the miss rate as a function of associativity. What do your results indicate about the number of conflict misses and CPU execution model?

5.12 [20/20/15/25] <5.5> Smith and Goodman [1983] found that for a given small size, a direct-mapped instruction cache consistently outperformed a fully associative instruction cache using LRU replacement.

a. [20] <5.5> Explain how this would be possible. (*Hint:* You can't explain this with the three C's model because it "ignores" replacement policy.)

b. [20] <5.5> Explain where replacement policy fits into the three C's model, and explain why this means that misses caused by a replacement policy are "ignored"—or, more precisely, cannot in general be definitively classified—by the three C's model.

c. [15] <5.5> Are there any replacement policies for the fully associative cache that would outperform the direct-mapped cache? Ignore the policy of "do what a direct-mapped cache would do."

d. [25] <5.5> Use a cache simulator to see if Smith and Goodman's results hold for memory reference traces that you have access to. If they do not hold, why not?

5.13 [15/20] <5.2, 5.5> McFarling [1989] found that the best memory hierarchy performance occurred when it was possible to prevent some instructions from entering the cache (see page 432).

a. [15] <5.5> Explain why McFarling's result could be true.

b. [20] <5.2, 5.5> The four memory hierarchy questions (Section 5.2) form a model for describing cache designs. Where does a cache that does not always *read-allocate* fit or not fit into this model?

5.14 [20/15/20/15/15] <1.4, 3.3, 5.5> Way prediction allows an associative cache to provide the hit time of a direct-mapped cache. The MIPS R10K processor uses way prediction to achieve a different goal: reduce the cost of the chip package.

The R10K hardware includes an on-chip L1 cache, on-chip L2 tag comparison circuitry, and an on-chip L2 way prediction table. L2 tag information is brought on chip to detect an L2 hit or miss. The way prediction table contains 8K 1-bit entries, each corresponding to two L2 cache blocks. L2 cache storage is built externally to the processor package, must be two-way associative, and may have one of several block sizes.

a. [20] <1.4, 5.5> How can way prediction reduce the number of pins needed on the R10K package to read L2 tags and data, and what is the impact on performance compared to a package with a full complement of pins to interface to the L2 cache?

b. [15] <5.5> What is the performance drawback of just using the same smaller number of pins but not including way prediction?

c. [20] <5.5> Assume that the R10K uses most-recently used way prediction. What are reasonable design choices for the cache state update(s) to make when the desired data is in the predicted way, the desired data is in the non-predicted way, and the desired data is not in the secondary cache?

d. [15] <5.5> If a 512 KB L2 cache has 64-byte blocks, how many way prediction table entries are needed? How would the R10K support this need?

e. [15] <3.3, 5.5> For a 4 MB L2 cache with 128-byte blocks, how is the usefulness of the R10K way prediction table analogous to that of a branch history table?

5.15 [25/20] <5.5> Simulating a single process workload gives miss rates that are lower than those seen in real computer systems.

a. [25] <5.5> Write a program that merges traces written by an instruction simulator such as SimpleScalar (*www.cs.wisc.edu/~mscalar/simplescalar.html*) and produces a stream of references that more accurately reflects a real computer workload. SimpleScalar has a precompiled set of SPEC95 benchmarks that may be useful for this exercise.

b. [20] <5.5> Run the resulting traces through a cache simulator for various cache sizes. How does the miss rate for the merged traces differ from the miss rate from running the programs sequentially?

✪ 5.16 [15/15] <5.5, 5.6> As caches increase in size, blocks often increase in size as well.

a. [15] <5.5, 5.6> If a large instruction cache has larger data blocks, is there still a need for prefetching? Explain the interaction between prefetching and increased block size in instruction caches.

b. [15] <5.5, 5.6> Is there a need for data prefetch instructions when data blocks get larger? Explain.

5.17 [15/20] <5.10, 5.13> The Alpha 21264 uses a fully associative cache with 128 entries for its TLB. This arrangement is very flexible, but can lead to performance issues.

a. [15] <5.10> Fully associative caches are often slow and/or difficult to build. How could you build a TLB from a two-way set-associative cache? What drawbacks are there to using a set-associative cache rather than a fully associative cache for the TLB?

b. [20] <5.10, 5.13> The 21264 TLB supports multiple page sizes. Could a memory system using a two-way set-associative TLB support multiple page sizes? Explain.

5.18 [15/15] <5.7, 5.10> The 21264 uses a virtually addressed instruction cache, removing the TLB from the critical path on instruction fetches. The use of virtually addressed caches can reduce time to fetch data from the cache, but can lead to problems (as discussed in Section 5.7).

a. [15] <5.7, 5.10> The 21264 eliminates aliases in the virtually addressed cache by checking eight different locations on each access. Other systems use page coloring to do the same thing. Is this really necessary for instruction caches? Explain.

b. [15] <5.7, 5.10> Virtually addressed caches often need to have more tag bits than physically addressed caches, both because virtual addresses are often longer than physical addresses and because virtually addressed caches need to store additional tag bits to distinguish cached blocks from different processes. How much added overhead does this contribute? Assume 64-bit virtual addresses, 8-bit process identifiers, and a physical memory that can hold up to 64 GB of main memory (i.e., physical tags need only be large enough to handle 36-bit physical addresses). How does overhead vary for different cache block sizes?

✪ 5.19 [15/15/15/15/15/15] <5.10, 5.11> Some memory systems handle TLB misses in software (as an exception), while others use hardware for TLB misses.

a. [15] <5.10, 5.11> What are the trade-offs between these two methods for handling TLB misses?

b. [15] <5.10, 5.11> Will TLB miss handling in software always be slower than TLB miss handling in hardware? Explain.

c. [15] <5.10, 5.11> Are there page table structures that would be difficult to handle in hardware, but possible in software? Are there any such structures that would be difficult for software to handle but easy for hardware to manage?

d. [15] <5.10, 5.11> Use the data from Figure 5.45 to calculate the penalty to CPI for TLB misses on the following workloads assuming hardware TLB handlers require 10 cycles per miss and software TLB handlers take 30 cycles per miss: (50% gcc, 25% perl, 25% ijpeg), (30% swim, 30% wave5, 20% hydro2d, 10% gcc).

e. [15] <5.10, 5.11> Are the TLB miss times in part (d) realistic? Discuss.

f. [15] <5.10, 5.11> Why are TLB miss rates for floating-point programs generally higher than those for integer programs?

5.20 [25/25/25/25/20] <5.10> How big should a TLB be? TLB misses are usually very fast (fewer than 10 instructions plus the cost of an exception), so it may not be worth having a huge TLB just to lower TLB miss rate a bit. Using the SimpleScalar simulator (*www.cs.wisc.edu/~mscalar/simplescalar.html*) and one or more SPEC95 benchmarks, calculate the TLB miss rate and the TLB overhead (in percentage of time wasted handling TLB misses) for the following TLB configurations. Assume that each TLB miss requires 20 instructions.

a. [25] <5.10> 128 entries, two-way set associative, 4 KB to 64 KB pages (going by powers of 2).

b. [25] <5.10> 256 entries, two-way set associative, 4 KB to 64 KB pages (going by powers of 2).

c. [25] <5.10> 512 entries, two-way set associative, 4 KB to 64 KB pages (going by powers of 2).

d. [25] <5.10> 1024 entries, two-way set associative, 4 KB to 64 KB pages (going by powers of 2).

e. [20] <5.10> What would be the effect on TLB miss rate and overhead for a multitasking environment? How would the context switch frequency affect the overhead?

5.21 [15/20/20] <5.11> It is possible to provide more flexible protection than that in the Intel Pentium architecture by using a protection scheme similar to that used in the HP-PA architecture. In such a scheme each page table entry contains a "protection ID" (key) along with access rights for the page, as shown below. On each reference, the CPU compares the protection ID in the page table entry with those stored in each of four protection ID registers (access to these registers requires that the CPU be in supervisor mode). If there is no match for the protection ID in the page table entry or if the access is not a permitted access (writing to a read-only page, for example), an exception is generated.

Protection ID	Permissions	Physical page number	Valid?	Dirty?	Used?

a. [15] <5.11> How could a process have more than four valid protection IDs at any given time? In other words, suppose a process wished to have 10 protection IDs simultaneously. Propose a mechanism by which this could be done (perhaps with help from software).

b. [20] <5.11> Explain how this model could be used to facilitate the construction of operating systems from relatively small pieces of code that can't overwrite each other (microkernels). What advantages might such an operating system have over a monolithic operating system in which any code in the OS can write to any memory location?

c. [20] <5.11> A simple design change to this system would allow two protection IDs for each page table entry, one for read access and the other for either write or execute access (the field is unused if neither the writable nor executable bit is set). What advantages might there be from having different protection IDs for read and write capabilities? *Hint:* Could this make it easier to share data and code between processes?

Protection ID 1	Protection ID 2	Executable?	Writable?	Physical page number	Valid?	Dirty?	Used?

5.22 [25/25/25] <5.16> One of the common pitfalls in memory system design is simulating too few references. Using your favorite simulator and several different traces, show the following:

a. [25] <5.16> Instruction cache miss rate versus trace length for varying sizes of instruction cache. Use a fixed trace and the same associativity and block size for each cache size—the point here is to measure the effect of using too short of a trace. Discuss the reasons for the different measured miss rates.

b. [25] <5.16> Data cache miss rate versus trace length for varying sizes of data cache. Perform the same analysis as in part (a).

c. [25] <5.16> Using your results, can you give a rule of thumb for how long of a simulation is necessary for a given cache size? How does this answer relate to the issues raised in Exercise 5.15 (i.e., how are short traces related to context switch issues)?

5.23 [15/15/10/10] <5.5, 5.16> Compulsory misses can distort the miss rate measured using a trace that is too short.

a. [15] <5.5, 5.16> The first 100,000 instruction references of a program are simulated on an 8 KB direct-mapped cache with 16-byte blocks, and a total of 600 misses occur. How many of the misses could be compulsory?

b. [15] <5.5, 5.16> Assume that to complete, the program from which the 100,000 initial references were taken executes for another 900,000 references. Assume no compulsory misses occur after the initial 100,000 references; that is, the cache initialization transient ended some time during the first 100,000 references. What range of miss rate is it possible to measure for the combination of this cache and this complete program?

c. [10] <5.5> If the simulated cache design has fewer blocks, what is the effect on the range of possible measured miss rate?

d. [10] <5.5, 5.16> If the simulated memory reference trace is lengthened, what is the effect on the range of possible measured miss rate?

5.24 [15/20/25/25/20/15/15] <5.2, 5.16> The miss behavior of a complete program may be quite different from that for any one segment (see Figure 5.53). Further, the entire memory reference trace of a program or set of programs may include billions and billions of addresses, requiring more time or more computing resources to simulate than is available. Under certain assumptions the magnitude of cache simulation work can be substantially reduced while maintaining accurate results. This exercise explores the limits of reduction.

a. [15] <5.16> To reduce simulation time and effort, traces should be as short as possible, but not shorter. How short can a trace be without being so short that it is impossible to establish a miss rate? (*Hint:* State your answer in the form of a necessary trace characteristic rather than as a specific trace length.) Note that to obtain good precision of measured miss rate, a trace should be perhaps a factor of 10 "longer" than your minimum.

b. [20] <5.2, 5.16> The miss rate data reported and discussed in the text are for an entire cache. If a cache has a large number of sets, it may be accurate to assume that any subset of sets has the same miss rate as does the entire cache. How should a trace be prepared to exploit this observation? By what maximum factor could simulation effort be reduced? In practice, 10% of sets are simulated (essentially a 90% reduction in trace size) to obtain good statistical confidence in the results, but how could you quantify the error introduced by this technique?

c. [25] <5.2> Simulation of many cache designs—size, number of sets, associativity—is needed before an informed decision on which one design to build can be made. For a *stacking* replacement policy, the contents of a set for a k-way associative cache includes, at any point during trace simulation, the contents of sets for all j-way associative caches, for $j < k$, provided the caches all have the same block size and number of sets. Further, if the blocks of the k-way set are ordered from "least replaceable" to "most replaceable," then the first j blocks in the k-way set are precisely the blocks that would be held at that exact time in a j-way associative cache, and their order represents exactly the same "least replaceable" to "most replaceable" labeling that the j-way cache would ascribe to each block. Thus, the blocks of the k-way set can be arranged in a stack from which, starting at the top, all j-way set contents can be found. LRU is an example of a stacking replacement policy.

For a fixed block size, a fixed number of sets, and a stacking replacement policy, a single simulation run can produce miss rates for one-way through k-way associative cache designs. Describe an algorithm to accomplish this economical simulation. Detail the data structure(s), any data structure maintenance, hit and miss event counting, and the formula for generating the set of cache

miss rate results of your method. Is there any error introduced by using only a single run? How much reduction in simulation effort might reasonably be expected?

d. [25] <5.2> Consider all blocks that map into one set of a cache of size N. For a cache of size $2N$, these same blocks all map into exactly two sets, assuming block size and associativity are the same. How can you extend the algorithm of part (c) for stacking replacement policy caches to use a single simulation run for multiple numbers of sets as well as multiple associativities? Is there any error introduced by using only a single run? How much reduction in simulation effort might reasonably be expected for this combined run?

e. [20] <5.2> Running a complete memory reference trace through a cache simulator means simulating many cache hits. While hits are good news, it is the misses that we need to know about. Prove that the same number of misses would be counted by a simulation of a k-way set-associative cache with s sets, block size b, and a stacking replacement policy on a given memory reference trace and by a simulation of that same cache on a stripped version of the given trace produced by retaining only those references that would be misses in a direct-mapped cache with s sets and block size b.

f. [15] <5.2> By how much does the trace stripping technique of part (e) reduce the simulation effort required for a k-way cache?

g. [15] <5.2> Can the techniques of parts (b), (d), and (e) be combined? Why or why not?

✪ 5.25 [Discussion] <3, 5> Designing caches for out-of-order (OOO) superscalar CPUs is difficult for several reasons. Clearly, the cache will need to be nonblocking and may need to cope with several outstanding misses. However, the access pattern for OOO superscalar processors differs from that generated by in-order execution. What are the differences, and how might they affect cache design for OOO processors?

5.26 [Discussion] <5.8> Few computers today take advantage of the extra security available with gates and rings found in a CPU like the Intel Pentium. Construct a scenario where the computer industry would switch over to this model of protection. What (if any) would be the cost of doing so, other than the costs of modifying the operating system kernel to use gates and rings?

5.27 [Discussion] <5.7, 5.8> Some people have argued that with increasing capacity of memory storage per chip, virtual memory is an idea whose time has passed, and they expect to see it dropped from future computers. Find reasons for and against this argument.

5.28 [Discussion] <5> A hypothetical new technology, magnetic RAM (MRAM), has been proposed. MRAM will have cost and density similar to that of DRAM, but will retain data even after power is removed. However, there is a drawback to MRAM: The amount of energy needed to write a bit (by switching the magnetic orientation of a small area of magnetic material) is higher than that needed to set a bit in DRAM. As a result, MRAM designers must trade off write rate and data

density: For a given density there is a maximum write rate beyond which the chip will melt. Propose uses for MRAM that would employ a dense, but slow to update memory. Are there applications for MRAM that would use a less dense but faster write memory?

5.29 [Discussion] <3, 4, 5> As we saw in Chapters 3 and 4, the time needed to execute a single instruction has fallen dramatically over the past few years, thanks to techniques such as pipelining, superscalar execution, and VLIW organization. However, memory access speeds have not kept up. As a result, memory access times may soon dominate program execution times (if they don't already). What is the impact of these changes on other computer science research areas such as algorithms, data structures, operating systems, and compilers? Find textbooks that suggest solutions to problems appropriate for older systems where processing time is the bottleneck, and suggest changes that might be more appropriate for newer systems in which the memory system is the bottleneck.