Test Driving Your Next Cache

Simulating cache performance using real memory traces

BY MARK D. HILL

The U.S. Government reports the fuel efficiency of new automobiles with the warning: Your actual mileage may vary. Most computer manufacturers report that their machines will execute up to 20 MIPS; some say you will get 14 MIPS sustained performance. In any case, your actual "mileage" will vary a lot and will probably be less than what the manufacturer claims.

One major reason that a computer's speed varies so much, and in fact rarely exceeds half its reported peak, is the effect of cache memories. (For more details on cache concepts and organization, see "Memory Caching Schemes" in the June issue.) While it is nice to assume that all references are serviced in a cache without accessing main memory, real programs are not as kind as that. All programs cause the cache to miss sometimes. Variation in the frequency of misses is caused by differences in loop, array, and record sizes, procedure call patterns, the number of local variables, etc. Furthermore, small differences in how often the cache misses are magnified by how much longer it takes to get information from main memory on a miss, often a factor of five to twenty. Thus, if a computer's cache misses on two percent of the references from someone else's program and on five percent of your program's references, you will find the computer 15 to 60 percent slower than they will.

This article will describe a method, called trace-driven simulation, that you can use to evaluate the effectiveness of alternative caches on your workload. Using trace-driven simulation reduces the uncertainty of how a computer will perform with your programs, allowing you to buy your next computer based on your numbers, not the manufacturer's. Below I provide some background on cache design, describe trace-driven simulation, and explain how to obtain the cache simulator. I restrict the discussion to caches in uniprocessors, since multiprocessor cache design and simulation are not as well understood.

Measuring Cache Performance

The most commonly used measure of cache performance is the miss ratio, which represents the fraction of memory references for which the

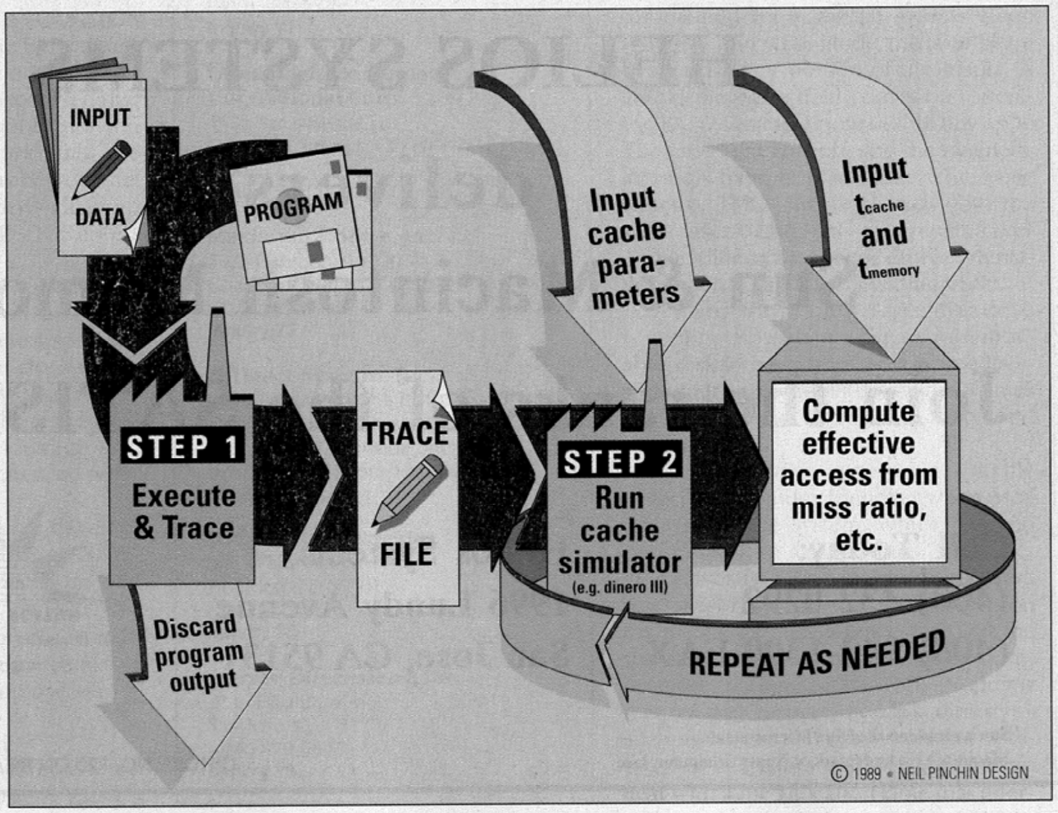

FIGURE 1. Trace-Driven Simulation

cache fails to have the information requested. Miss ratio is used because it is easy to define, interpret, compute, and perhaps most important, it is implementation-independent. This independence facilitates performance comparisons of caches that are implemented with different technologies and in different kinds of systems as well as caches that are not yet implemented.

The miss ratio metric, however, suffers from two disadvantages. First, since miss ratio comparisons contrast the number of misses, they can be misleading if the delay associated with a miss varies. For instance, increasing the block size (i.e., the amount of information brought in on a cache miss) often reduces the number of misses and hence the miss ratio, but often it also increases the number of cycles needed to load that information. The actual change in cache performance will depend on how much the number of misses decreases and how much the time to service a miss increases. The second disadvantage is that small changes in the miss ratio can lead to large changes in the average time taken to service memory references. Thus, one needs to be careful when ignoring small differences in miss ratio.

For this reason, I prefer to measure cache performance with effective access time, or the average time required to service a memory reference. Using effective access time has three advantages. First, relative changes in effective access translate almost directly into changes in how fast a computer runs. Reducing effective access time by 10 percent usually increases a computer's speed by 5 to 10 percent, depending on the frequency of memory references. Second, one can compare caches where the delay for a miss differs, as in the above example. Third, effective access time can be simply modeled and computed from the miss ratio and two implementation parameters:

$$t_{eff} = t_{cache} + m \times t_{memory}$$

where t_{cache}, m, and t_{memory} are cache access (hit) time, miss ratio, and average miss penalty (the delay beyond a cache access to access main memory). More sophisticated models of effective access time treat writes differently than reads and instruction fetches.

The principal disadvantage of effective access time is that implementation details must be examined and assumptions made to determine the values of t_{cache} and t_{memory}. Performance estimates with any implementation assumptions are less general, and those with incorrect assumptions are misleading. **Table 1** shows some sample miss ratios and effective access times. Note, for example, that with $t_{memory} = 10$ cycles, a 4-Kbyte single-cycle cache (i.e., no wait states) has an effective access time of 2.0 cycles, better than any two-cycle cache.

Key Cache Parameters

The cache design parameters that have the greatest effect on t_{cache}, m, and t_{memory} include cache size, whether data and instructions are cached together, block size, and associativity.

Cache size and whether data and instructions are cached together are most important. For highest performance, the cache should be the largest size that still permits no-wait-state access. In practice, cost or limited chip or board area can cause designers to choose smaller caches. Today, typical on-chip caches contain 4 Kbytes or less, while large static RAM chips permit much larger board-level caches (e.g., 64 Kbytes) than were heretofore optimal.

A cache designer must also choose between caching data and instructions in one cache or in separate caches. The principal advantage of separate caches is that they facilitate accessing an instruction and a data item simultaneously, while the main disadvantage is higher cost. Most caches used on today's microprocessors are separate, while most board-level caches are unified.

Block (line) size is the amount of data brought into a cache on a miss. Larger blocks usually yield lower miss ratios because of spatial locality (the tendency to reference near recent memory references), but they also increase t_{memory}, or memory

TABLE 1. This table shows how cache miss ratio and effective access time interact. The miss ratios given here are for direct-mapped unified caches (data and instructions together) that have 32-byte blocks and are driven by some user/system traces from a DEC VAX-11 and an IBM 370. Your miss ratios and effective access times may differ considerably from these.

		Effective Access Time (cycles)		
Cache Size (bytes)	Miss Ratio (percent)	$t_{cache} = 1$ $t_{memory} = 10$	$t_{cache} = 1$ $t_{memory} = 20$	$t_{cache} = 2$ $t_{memory} = 10$
512	25	3.5	6.0	4.5
1K	19	2.9	4.8	3.9
2K	14	2.4	3.8	3.4
4K	10	2.0	3.0	3.0
8K	8	1.8	2.6	2.8
16K	6	1.6	2.2	2.6
32K	4	1.4	1.8	2.4
64K	3	1.3	1.6	2.3
128K	2	1.2	1.4	2.2

Obsolescence designed out

Now, a 386 motherboard that keeps up with progress. One board with over 200 options you choose —in seconds. One board you can upgrade in the future—up to 40Mz —in seconds.

We've designed a 386 motherboard that offers the best of what is available today with what is coming in the future. We've put the most advanced support functions available on the motherboard. Whether it's a general purpose computer, a workstation, or a file server, you choose the best combination of processor features for your needs. Your investment is secure for the coming years of progress. Features of the motherboard include:

- **6 AT slots, 1 PC slot**
- **2 serials, 1 parallel, 2 floppies**
- **G2 386 chipset & Quadtel BIOS**
- **LIM 4.0, EEMS, & shadow ROM**
- **Interleaved 0 wait state memory**
- **Standard baby AT board size**
- **Engineered and manufactured in U.S.A. using advanced surface mounted technologies**

The processors are on a separate board connected through a high speed private bus. Options are:

- **Intel 80386 at 20, 25, 33, or 40Mz (when available)**
- **both 80387 and Weitek 3167 coprocessor sockets standard**
- **Austek cache controller with set-associative or direct modes**
- **Cache RAM from 32Kb to 128Kb**
- **Up to 16Mb of SIMM memory**

This board set is sold alone or in configured FCC approved systems. Call us today for details and a quote. Quantity prices for VARS and distributors.

Business MicroSystems, Inc.
531 South Plymouth Court
Chicago, Illinois 60605
312.939.0062 FAX 312.939.0066
1.800.282.0286

NO. 139 ON READER SERVICE CARD

system cost. This is because transferring more data on a cache miss requires either more transfers or a wider path from main memory. Block sizes of 16 or 32 bytes are best for many caches; however, 8-byte blocks may be preferred for caches of 4 Kbytes or smaller, while 64-byte blocks may be optimal in caches larger than 64 Kbytes. For ease of implementation, almost all caches use aligned blocks, meaning that the address of the first item in the block divided by the block size has remainder zero.

Associativity is the final important cache parameter. This is the number of places in a cache where a block may reside. Larger associativities lower the miss ratio by providing the hardware with more flexibility regarding what to throw out of cache to make room for a new block. However, they increase cache cost and access time by requiring that more places be searched to determine if the reference is present. Blocks must be replaced from a cache, because a cache can hold far fewer blocks than exist in main memory. A cache that allows a block to be anywhere is called fully-associative; one that restricts a block to one place in the cache is direct-mapped; and one organized to search *n* places is *n*-way set-associative.

Intuition suggests that an *n*-way set-associative cache with large *n* is probably the best compromise between cost and performance. Unfortunately, intuition is wrong, much as the intuition that suggests that more complex instructions always lead to faster computers is wrong. Data show that direct-mapped, two-way and four-way set-associative caches are usually best. I recommend using two-way, unless direct-mapped allows a larger no-wait-state cache to be built (frequently true for large caches) or four-way costs little extra (rarely true).

Cache design parameters of secondary importance include how to make replacement decisions, whether or not to prefetch, and how to update memory on writes. Replacing the least-recently-used block usually gives the lowest miss ratio, but random replacement works better than expected, at lower cost.

Prefetching (loading data before it is referenced) is rarely worth its implementation complexity, except for small instruction caches.

Caches may update memory on every write (write-through, non-store-in) or only when a block is replaced (write-back, copy-back, store-in). A naive implementation of write-through requires a processor to stall until each write completes at main memory. Stall time can be significantly reduced by adding write buffers to hold pending writes and allowing the processor to continue. An implementation of write-back requires that each cache block remember whether it has been written and that main memory be updated when a so-called "dirty" block is replaced. Since replacement usually happens on cache misses, the update must occur before or after the new block is loaded, costing additional delay if done before or requiring a special buffer if done after. In a uniprocessor, write-through and write-back with proper buffering yield similar performance. Thus, implementation considerations dictate which is preferred. In multiprocessor caches (not discussed in this article), the choice of write policy is of primary importance, and a form of write-back is most often used because it results in less traffic to main memory.

LISTING 1. Pseudo-C code after register allocation.

```
reg_addr_a = &a;
reg_sum = 0;
for (reg_i = 20; reg_i >= 0; reg_i = reg_i - 4) {
      reg_sum = reg_sum + *(reg_addr_a + reg_i);
}
sum = reg_sum;
```

LISTING 2. Assembly language produced for a load-store architecture.

```
104 move 0x200, reg1            ;; assume 0x200 is address of array a
108 move 0, reg2                ;; zero sum
10c move 0x14, reg3             ;; initialize i

110 load reg1+reg3, reg4        ;; begin loop; load a[i] in a register
114 add reg2, reg4, reg2        ;; add to sum
118 sub reg3, 4, reg3           ;; reduce i
11c branch>=0 0x110             ;; end loop

120 store reg3, 0x300           ;; assume 0x300 is address of sum
```

Test Driving with Simulation

Trace-driven simulation, a two-step process, assesses expected cache performance by evaluating how your programs interact with the caches you design or purchase. In step one, you gather and save data on how your programs reference memory. This process is called tracing, and the resulting data is called a trace. In step two, you feed one or more of your traces into a program that mimics a cache in order to compute a miss ratio for your trace. Finally, you combine that miss ratio with assumptions for t_{cache} and t_{memory} to compute the effective access time for caches of interest. Step two can be repeated many times to assess alternative cache performance on your trace. **Figure 1** illustrates the process of trace-driven simulation.

Step One: Gathering Traces

I will illustrate trace-driven simulation with the following fragment of C code, which, when executed, will make 35 memory references. Useful simulations, however, require traces that are millions of references long.

```
sum = 0;
for (i=0; i<6; i++) {
   sum = sum + a[i];
}
```

An optimizing compiler could do register allocation and alter the loop index variable to get the pseudo-C code shown in **Listing 1** (where `reg` stands for a register, each element of the array *a* is 4 bytes in size, and byte addressing is used). Finally, the compiler could produce the assembly language shown in **Listing 2**, for a load-store architecture.

To trace this code, you must execute it and record the type of reference (read, write, or instruction fetch) and the address for each memory reference in program execution order. The execution of the above program yields the trace shown in **Listing 3**, where 0 stands for read, 1 for write, and 2 for instruction fetch, and addresses are in hexadecimal.

Saving this trace completes step one for this example. I have, however, glossed over two items of considerable practical complexity. The first item is how to execute the program and gather the trace. Three methods are used: software, microcode, and hardware; none is clearly best. In the software method, you write a simulator of the hardware to be traced that will both execute programs and write a trace file. The advantages of this approach are that little hardware expertise is required, the target machine need not exist, and the trace file can be immedi-

LISTING 3. Trace file used as input to the cache simulator.

```
2 104
2 108
2 10c
2 110          ;; begin loop iteration 1
0 214
2 114
2 118
2 11c
2 110          ;; begin loop iteration 2
0 210
2 114
2 118
2 11c
2 110          ;; begin loop iteration 3
0 20c
2 114
2 118
2 11c
2 110          ;; begin loop iteration 4
0 208
2 114
2 118
2 11c
2 110          ;; begin loop iteration 5
0 204
2 114
2 118
2 11c
2 110          ;; begin loop iteration 6
0 200
2 114
2 118
2 11c
2 120
1 300
```

ately piped into a cache simulator without disturbing experimental results. The main disadvantages are that the approach can be very slow and it is hard to trace anything but single-user programs (i.e., no operating system or multiprogramming effects).

In the microcode method, you modify a machine's microcode so that it writes a trace file as a side effect of execution. The advantages of this approach are that programs being traced run much faster than with the software method and multiprogramming effects and non-real-time parts of the operating system can be traced. The principal disadvantages are that it requires a machine to exist and have extra space in writable control store. You also need to have considerable hardware expertise. This approach has been used with great success on the VAX 8200. In the final method, you attach a hardware monitor to record addresses off a computer's bus as it runs at full speed. This approach allows all things to be traced, but requires special hardware and expertise.

The second problem encountered gathering traces is that you must gather traces that are both representative of your workload and sufficiently long to exercise caches of interest. If you fail to get good traces and feed garbage into a cache simulator, the simulator will faithfully mimic cache behavior and give you garbage miss ratios. The following rules are useful for gathering good traces:

- Select a dozen or more programs important to you (usually these include compilers and an operating system).
- Run the programs on important input data (e.g., the compiler compiling the operating system).
- Make sure traces are long enough that each cache block will see numerous misses (often requiring millions of addresses, implying tens of millions of bytes of trace data).

Step Two: Running a Cache Simulator

The easy part of trace-driven simulation is running a cache simulator to explore the cache design space. I wrote such a simulator in C for Unix and named it dinero (Spanish for cash). The current version is dineroIII.

Running dineroIII is much simpler than gathering traces. After you specify cache parameters with command-line arguments and feed a trace into its standard input, dineroIII writes the miss ratio and related numbers to standard output. Consider the following command line:

```
dineroIII -i32768 -d32768 -b16
  -a1 < trace1
```

where `trace1` is a file containing the extremely short (35-reference) trace described above. This command line tells dineroIII to simulate a 32-Kbyte instruction cache and a separate 32-Kbyte data cache with 16-byte (four-word) blocks and direct-mapped placement and to use the defaults for other parameters (e.g., write-back write policy).

FIGURE 2. This sample simulation shows results of a short run on the dineroIII cache simulator. The first five lines give information about the version of dineroIII used and the cache parameters selected (e.g., "blocksize=16" means that cache block size is 16 bytes). The two lines labeled "Demand Fetches" give the number and fraction of memory references that are of a given type (e.g., 28 of 35 references are instruction fetches). "Demand Misses" give the number of misses and miss ratio for each type of reference (e.g., 2 of 6 data reads missed). Finally, the last metrics give total (normalized) numbers for words brought into the cache (normalized by the number of memory references), the words written back to memory (normalized by the number of writes), and the total traffic into and out of the cache (also normalized by the number of memory references).

```
-Dinero III by Mark D. Hill.
-Version 3.1, Released 2/25/86.

CMDLINE: dineroIII -i32768 -d32768 -b16 -a1
CACHE (in bytes): blocksize=16, sub-blocksize=0, Dsize=32768, Isize=32768.
POLICIES: assoc=1-way, replacement=1, fetch=d(1,0), write=c, allocate=w.

Metrics                   Access Type:
 (totals,fraction)        Total    Instrn    Data     Read     Write    Misc
-----------------------------------------------------------------------------
Demand Fetches                35       28        7        6        1        0
                          1.0000   0.8000   0.2000   0.1714   0.0286   0.0000

Demand Misses                  6        3        3        2        1        0
                          0.1714   0.1071   0.4286   0.3333   1.0000   0.0000

Words From Memory             24
 (/Demand Fetches)        0.6857

Words Copied-Back              4
 (/Demand Writes)         4.0000

Total Traffic (words)         28
 (/Demand Fetches)        0.8000
```

Figure 2 shows the results of this simulation run. The two caches experience six misses, three on instruction blocks containing hexadecimal addresses 100-10f, 110-11f, 120-12f, and three on data blocks 200-20f, 210-21f, and 300-30f. The resulting miss ratio is 0.1714 (6/35). The trace is not long enough for any blocks to be replaced.

I can now experiment with varying the block size by running:

```
dineroIII -i32768 -d32768 -b32
  -a1 < trace1
```

Figure 3 shows these results, where four misses now occur on blocks 100-11f, 120-13f, 200-21f, and 300-31f.

FIGURE 3. This figure shows results similar to those of Figure 2. The only input change is that a block size of 32 bytes (`-b32`) is used in place of 16 bytes. This change does not affect the number of fetches but does affect the miss ratio and traffic numbers.

```
-Dinero III by Mark D. Hill.
-Version 3.1, Released 2/25/86.

CMDLINE: dineroIII -i32768 -d32768 -b32 -a1
CACHE (in bytes): blocksize=32, sub-blocksize=0, Dsize=32768, Isize=32768.
POLICIES: assoc=1-way, replacement=1, fetch=d(1,0), write=c, allocate=w.

Metrics                   Access Type:
 (totals,fraction)        Total    Instrn    Data     Read     Write    Misc
-----------------------------------------------------------------------------
Demand Fetches                35       28        7        6        1        0
                          1.0000   0.8000   0.2000   0.1714   0.0286   0.0000

Demand Misses                  4        2        2        1        1        0
                          0.1143   0.0714   0.2857   0.1667   1.0000   0.0000

Words From Memory             32
 (/Demand Fetches)        0.9143

Words Copied-Back              8
 (/Demand Writes)         8.0000

Total Traffic (words)         40
 (/Demand Fetches)        1.1429
```

Of course, no conclusions regarding the best block size should be drawn from a trace of 35 references. I'll now consider a more realistic trace containing 0.5 million instructions from the SPUR Lisp compiler, compiling part of itself. (SPUR is a research project at U.C. Berkeley that built a multiprocessor workstation using coherent caches and custom RISC microprocessors.) **Figure 4** shows that with this trace, doubling block size from 16 bytes to 32 bytes reduces the overall miss ratio from 0.0498 to 0.0356.

Which block size is preferred? Assume that cache misses with 16-byte blocks take 10 cycles beyond a single-cycle cache access to service. In this case, effective access time is 1.50 cycles (1 + 0.0498 × 10). If 32-byte blocks can also be transferred from memory in 10 cycles, then effective access time with them is 1.36 cycles. On the other hand, if doubling block size doubles the miss handling delay, making it 20 cycles, effective access time is a poor 1.71 cycles. Therefore, neglecting cost, 32-byte blocks are superior in the first case and inferior in the second.

You may also wonder at what miss penalty do the two block sizes yield the same effective access time. To answer this you need only solve:

$$1 + (0.0498) \times (10) = 1 + (0.0356) \times t_{memory}$$

to find that using 32-byte blocks yields a lower effective access time if their miss penalty is less than 14 cycles.

Similar analysis can be done for other cache parameters. A reliable analysis should use a dozen or more traces and include cache cost considerations.

I've focused here on caches in a uniprocessor, where minimizing effective access time and cache cost are most important. In the future, however, many caches will be attached to each processor in a multiprocessor machine. Multiprocessor caches differ from uniprocessor caches in at least two ways. First, multiprocessor caches may be more concerned with reducing interconnection network traffic than minimizing effective access time. Second, in many cases, they must guarantee that cached information does not become out of date. This problem, called cache coherency, arises when one processor writes information that another has cached. Multiprocessor cache design is currently a topic of much research interest. Thus, our understanding of what is best is likely to evolve in the coming years. Interested readers should consult the studies by Archibald and Baer, and Goodman (references are listed below).

FIGURE 4. This figure shows results from two simulations with a 0.5 million-address trace. All performance differences are caused by using 16-byte blocks in the first simulation and 32-byte blocks in the second.

```
-Dinero III by Mark D. Hill.
-Version 3.1, Released 2/25/86.

CMDLINE: dineroIII -i32768 -d32768 -b16 -a1
CACHE (in bytes): blocksize=16, sub-blocksize=0, Dsize=32768, Isize=32768.
POLICIES: assoc=1-way, replacement=l, fetch=d(1,0), write=c, allocate=w.

Metrics                   Access Type:
 (totals,fraction)        Total    Instrn    Data     Read     Write    Misc
-------------------------------------------------------------------------------
Demand Fetches            640146   500000   140146   114679    25467        0
                          1.0000   0.7811   0.2189   0.1791   0.0398   0.0000

Demand Misses              31884    19850    12034     8295     3739        0
                          0.0498   0.0397   0.0859   0.0723   0.1468   0.0000

Words From Memory         127536
 (/Demand Fetches)        0.1992

Words Copied-Back          17704
 (/Demand Writes)         0.6952

Total Traffic (words)     145240
 (/Demand Fetches)        0.2269
```

```
-Dinero III by Mark D. Hill.
-Version 3.1, Released 2/25/86.

CMDLINE: dineroIII -i32768 -d32768 -b32 -a1
CACHE (in bytes): blocksize=32, sub-blocksize=0, Dsize=32768, Isize=32768.
POLICIES: assoc=1-way, replacement=l, fetch=d(1,0), write=c, allocate=w.

Metrics                   Access Type:
 (totals,fraction)        Total    Instrn    Data     Read     Write    Misc
-------------------------------------------------------------------------------
Demand Fetches            640146   500000   140146   114679    25467        0
                          1.0000   0.7811   0.2189   0.1791   0.0398   0.0000

Demand Misses              22805    13261     9544     7498     2046        0
                          0.0356   0.0265   0.0681   0.0654   0.0803   0.0000

Words From Memory         182440
 (/Demand Fetches)        0.2850

Words Copied-Back          21080
 (/Demand Writes)         0.8277

Total Traffic (words)     203520
 (/Demand Fetches)        0.3179
```

Obtaining the Cache Simulator

I wrote dineroIII in C under Unix to do my research but am willing to make the source code available to other researchers and designers. The software will be available from *MIPS'* online listing service (see the editorial, page 4). I ask, but do not require, individuals who wish to acquire dineroIII to make an unrestricted donation of $30 (U.S.) to my University of Wisconsin grants. I ask companies to give $300. This money will aid continued research. To do this, write a check payable to Computer Sciences Fund —University of Wisconsin Foundation and mail it to Prof. Mark D. Hill, Computer Sciences Department, 1210 West Dayton Street, University of Wisconsin, Madison, WI 53706.

If you choose to acquire dineroIII, feel free to modify it and give it to researchers in your company. I ask, however, that you: (1) indicate to others any modifications you make, (2) do not distribute it to researchers beyond your company, (3) do not sell it, and (4) acknowledge dineroIII and me in papers that use it to a significant extent. I make no representations about the suitability of this software for any purpose. It is provided "as is," without expressed or implied warranty.

Additional Information

In this article I have discussed important aspects of cache design and have introduced a trace-driven simulation technique for evaluating caches. Readers interested in studying cache size, block size, and associativity in more detail may consult the studies by Przybylski, Horowitz, and Hennessy, M. D. Hill, and A. J. Smith's article, "Line Size Choice for CPU Caches," respectively. Those interested in more on trace-driven simulation should see Smith's study, "Cache Evaluation and the Impact of Workload Choice." ■

References

Archibald, J. and Baer, J."Cache Coherence Protocols: Evaluation Using Multiprocessor Simulation Model." *ACM Trans. on Computer Systems*, November 1986, pp. 273-298.

Goodman, J.R. June 1983. "Using Cache Memory to Reduce Processor-Memory Traffic." *Proc. Tenth International Symposium on Computer Architecture*, pp. 124-131. Stockholm, Sweden.

Hill, M.D."A Case for Direct-Mapped Caches." *IEEE Computer*, December 1988, pp. 25-40.

Przybylski, S., Horowitz, M., and Hennessy, J. June 1988. "Performance Tradeoffs in Cache Design." *15th Annual International Symposium on Computer Architecture*, pp. 290-298. Honolulu, Hawaii.

Smith, A. J. June 1985. "Cache Evaluation and the Impact of Workload Choice." *Proc. Twelfth International Symposium on Computer Architecture*. pp. 64-73.

———"LineSize Choice for CPU Caches." *IEEE Trans. on Computers*, September 1987, pp. 1063-1075.

Mark D. Hill is Assistant Professor of Computer Sciences at the University of Wisconsin at Madison and a recipient of the 1989 Presidential Young Investigator Award.

386 User's Log

Getting workstation capabilities from a 386 PC

BY WILLIAM L. RINKO-GAY

My typical working environment is an AST 20-MHz 386-based PC linked to our 3Com LAN with an Etherlink II card. An 80-Mbyte ESDI drive, 4 Mbytes of RAM, and a VGA graphics adapter and monitor round out my hardware configuration. In this series of articles I'll be telling you what I find as I dig into the capabilities of 386 machines. I'll share the problems I run into as well as the solutions. By way of background, I studied electrical engineering in college, and since then I've been working on minicomputers, engineering workstations, and microcomputers in a variety of environments. At *MIPS* I evaluate computers, software development systems, and benchmarking methodologies. I still write a lot of utilities when I need them.

Working on an Engineering Workstation

If you've used an engineering workstation, you're familiar with the differences between workstations and PCs. I used to work on an Apollo network. My workstation was the DN4000, which had about 4 Mbytes of real memory, more virtual memory than I ever used, and a very high resolution monochrome display that was run by the Display Manager (DM). DM sits between the screen and Aegis, Apollo's operating system. (Apollo layered Unix on top of Aegis, and this is what I used.) With DM you can create windows, resize them, and move them around.

The interesting thing about these windows is that they are full size. If you're used to Microsoft Windows or DESQview, you know that to display DOS in a small window is to give up several columns or rows. With Apollo's DM, the window you create for a text-based application can be a full-sized screen. You don't have to scroll horizontally or vertically. The text wraps at the right of the window regardless of the width you set and, unlike text displayed in Windows, doesn't extend beyond the window. If you create a window that's 50 columns wide and 10 lines deep, the application uses only 50 columns and 10 lines. This is a nice feature for applications like editors and spreadsheets, especially when you're displaying many of them at once. You can see each window without overlap if you want.

In addition to good display management, the Apollo network (like many Unix networks) gives you complete access to hardware installed on other systems on the network. In my job I worked with coprocessor cards, but I never had one of the cards on my workstation. When I wanted to test a program with one of the cards, I just accessed a system that had a card installed. This feature also came in handy if I needed a math coprocessor or modem or whatever. Sharing hardware wasn't entirely trouble-free. Occasionally the loudspeaker would announce, "Whoever just shut down the Saturn node is in big trouble," but we lived with these minor problems.

Last, the virtual memory the workstation network gave me was tremendous. I never ran out of memory. Processes could have all they wanted, regardless of what else was running. Of course, using more memory than was physically available slowed things down a bit, but I was never told that memory was not available.

The user interface, network capabilities, and the virtual memory are great to have when you're performing multiple tasks at the same time. 386 PCs can run multiple tasks, but those sophisticated features make a huge difference.

Tapping into the Power of the 386

Compared to the Apollo, the computer I use now seems low powered. I'm not talking about MFLOPS, but capabilities for the end user. The AST machine may benchmark faster than the Apollo, but its network environment and user interface are much less powerful.

I need multitasking and windowing environments. For a lot of what I do, context switching works okay, but multitasking is what I require. I print a lot of graphs, and bit-mapped printing takes a lot of computing power. I like to throw that in the background when I can. Compiling large programs is the same. I also do things like copying files in the background, a habit I got into while using the workstation. I now keep it up with the help of DOS-compatible multitasking operating systems.

When I'm working on an article, I usually have four directly related things going at once. I'm graphing benchmark results I've gathered so I can quickly see the overall picture. I'm running a spreadsheet that lets me make group calculations, like average percent differences in AIM benchmark results (the results are usually spread out all over my drive, and DOS is the easiest way to get them). I use a calculator for quick, single-point calculations, and I write in a word processor. I also use DOS to run programs I've written that put benchmark data in a format I can import elsewhere. In addition, I must have a business card filer (I constantly have to make phone calls in the middle of my work) and my calendar available at any time. In the background is my network server, which I use to access a printer and, occasionally, share files.

Windows/386

My first attempt to make all my requirements work together was Microsoft Windows/386, a fully multitasking environment and graphical user interface (GUI). It runs DOS programs in windows or with the full screen. When used in windows, programs display only as much on screen as you've allowed room for. If you don't make the windows large enough, text extends beyond them and you have to use horizontal and vertical scroll bars.

Compared to earlier versions of Windows, Windows/386 is slightly different. First, it supports multitasking. You get access to multitasking either through the Program Information File (PIF), which