Fast-Cache: A New Abstraction for Memory System Simulation

Alvin R. Lebeck
David A. Wood

Technical Report #1211

January 1994

Fast-Cache: A New Abstraction for Memory System Simulation

Alvin R. Lebeck
David A. Wood

Computer Sciences Department
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI. 53706
(608) 262-6617
alvy@cs.wisc.edu

November 4, 1993

Abstract

Trace-driven simulation has long been the dominate technique for evaluating memory system performance. However, the reference trace abstraction, upon which it is based, does not exploit the full potential of on-the-fly simulation systems, which tightly couple reference generation and simulation. In particular, an on-the-fly trace-driven system must simulate each reference even when the common case, e.g., a cache hit, requires no action. We have developed a new memory system simulation method that optimizes this common case, significantly reducing simulation time. Fast-Cache tightly integrates reference generation and simulation by providing the abstraction of tagged memory blocks: each reference invokes a user-specified function depending upon the reference type and memory block state. The simulator controls how references are processed by manipulating memory block states, specifying a special NULL function for no action cases. Fast-Cache implements this abstraction by using binary-rewriting to perform a table lookup before each memory reference. On a SPARCStation 10, Fast-Cache simulation times are two to three times faster than a conventional trace-driven simulator that calls a procedure on each memory reference; simulation times are only three to six times slower than the original, un-instrumented program. Fast-Cache also outperforms specialized hardware, the error correcting code (ECC) bits on a Thinking Machines CM-5, for all but one of our experiments. Fast-Cache slowdowns range from 3.1 to 6.7, whereas ECC slowdowns range from 1.3 to 46.

1. Introduction

Simulation is the most-widely-used method to evaluate memory system performance. Hardware designers use simulation to develop new memory system architectures, while programmers use it to improve the utilization of existing memory systems [6, 8, 17, 20]. However, current simulation techniques are discouragingly slow; simulation times are at least an order of magnitude slower than the execution time of the original program. Gee, et. al. [5], estimate that *17 months* of processing time were used to obtain miss ratios for the SPEC92 benchmarks [19].

Most memory system studies rely on *trace-driven simulation* [23], which consists of two components: reference generation and trace simulation. A reference generator produces a list of memory addresses—a *reference trace*—in program order. A trace simulator models the memory system, at an arbitrary level of detail, by processing each entry in the reference trace and simulating the actions of the hardware.

The reference trace abstraction has two main advantages. First, the trace acts as a simple interface to hide the the details of reference generation from the simulator itself, simplifying simulator design and providing a measure of portability. Second, the trace can be saved and reused for multiple simulations, guaranteeing reproducible results and amortizing reference generation overhead. However, software reference generation techniques have improved to the point that regenerating the trace is nearly as efficient as reading it from disk or tape [13]. On-the-fly simulation techniques—where the trace is simulated as soon as it's produced—have become popular because they

eliminate I/O overhead, context switches, and large storage requirements [1, 2, 4, 20].

However, these on-the-fly simulation systems continue to use the reference trace abstraction. Although simple, this abstraction requires the simulator to process each reference; this is unnecessary in most simulations because the common case requires no action. For example, to compute miss ratio, the most popular memory system metric, a simulator need only count misses since the total number of memory references can quickly be obtained by profiling tools [13]. In Tycho [7], the simulator used by Gee et. al., hits in the smallest (1 Kilo-byte) direct-mapped cache require no action. Therefore, sixty to seventy percent of the references invoke the simulator unnecessarily. Increasing the smallest cache size to 16 Kilo-bytes filters nearly 90% of the references.

This paper presents a new abstraction for memory system simulation—based on tagged memory blocks—specifically designed for on-the-fly simulation. In this new abstraction, reference generation and simulation are tightly coupled. On each memory reference, the reference generator invokes a user-specified function depending upon the reference type and memory block state. The simulator controls which function is invoked by manipulating the states. A predefined NULL function allows simulators to expedite the processing of no action cases.

To illustrate this new abstraction, consider a simulation to count cache misses. Memory blocks are either present in the cache, or absent; the simulator can represent these cases with the two states *valid* and *invalid*. References to blocks tagged *valid* invoke the NULL handler and continue immediately; references to *invalid* blocks invoke a user-written *miss* handler. The miss handler counts the miss, selects a victim using a standard cache data structure, and updates the state of both the replaced and referenced blocks. Because most references invoke the predefined NULL function, this simulation is much faster than one using the traditional trace abstraction.

A special case of the tagged block abstraction has been previously implemented using special hardware. The Wisconsin Wind Tunnel (WWT) uses the error correcting code (ECC) bits of a Thinking Machines CM-5 as valid bits [22]. References to valid blocks, e.g., cache hits, execute at hardware speed; references to invalid blocks, e.g., cache misses, trap to the simulator. By executing most references without software intervention, WWT potentially achieves significant performance gains over other simulation systems. However, it requires operating system and hardware support that is not readily available on most machines, and cannot easily be generalized to support more than two tag values.

Fast-Cache is a software system that implements a general version of the tagged memory block abstraction. It provides an efficient simulation framework by:

- highly-optimizing no action cases;
- rapidly invoking the necessary routine for action cases;
- isolating simulator writers from details of reference generation;
- providing simulator portability.

Fast-Cache achieves these goals by inserting 9 instructions before each memory reference to lookup a memory block's state and invoke the user-specified handler. The NULL handler executes two instructions, so no action cases require only 11 instructions. Fast-Cache uses binary rewriting to insert the necessary instrumentation and automatically incorporate user-written handlers. Fast-Cache provides a set of routines to manage the tagged memory block abstraction, thereby isolating simulator writers from the details of reference generation.

The tagged memory block abstraction supports complex memory system simulations. For example, Fast-Cache is currently being used to simulate a system that includes a translation look-aside buffer, a two-level cache hierarchy, and a memory-mapped network interface. Fast-Cache also facilitates simulations that require the notion of time (e.g., prefetch operations and write buffers) by providing accurate instruction cycle counts.

Although generality is an important feature of Fast-Cache, in this paper we focus on the performance improvements achieved by optimizing cache simulation for the common case—cache hits. Our results show that Fast-Cache reduces simulation time by a factor of two or three compared to a trace-driven simulation, where the

reference generator calls a procedure on every memory reference. Fast-Cache simulations run 3.7 to 5.5 times slower than the original program on a SPARCstation 10, while a trace-driven simulator runs 10 to 15 times slower. We also compare Fast-Cache to a hardware implementation of tagged memory blocks that uses the error correcting code (ECC) bits on a CM-5 [22]. Fast-Cache slowdowns range from 3.1 to 6.7, while ECC slowdowns range from 1.3 to over 45. For very low miss ratios, ECC outperforms Fast-Cache because of its low overhead miss detection. However, Fast-Cache performs significantly better than ECC for miss ratios greater than 2% because ECC incurs much higher overhead on a miss.

This paper is organized as follows. The next section describes Fast-Cache, and Section 3 compares its performance to a conventional trace-driven simulator on a SPARCstation 10. Section 4 compares Fast-Cache to a hardware implementation that uses the error correcting code bits on a CM-5 as memory block tags. In Section 5, we discuss some applications of Fast-Cache and possible extensions. Related work is discussed in Section 6, and we give our conclusions in Section 7.

2. Fast-Cache

In this section we describe Fast-Cache, a simulation system that implements the tagged memory block abstraction. By tightly coupling reference generation and simulation, Fast-Cache significantly improves performance by highly-optimizing cases that do not require simulator action. Performance also improves when action is required, since Fast-Cache directly invokes (user-specified) functions. Fast-Cache isolates simulator writers from the details of reference generation by providing a well defined interface for manipulating memory block tags.

Fast-Cache allocates a user-specified amount of state for each memory block (e.g., cache block or page) in a program's virtual address space. Each state can be assigned a unique *handler*, and different handlers can be

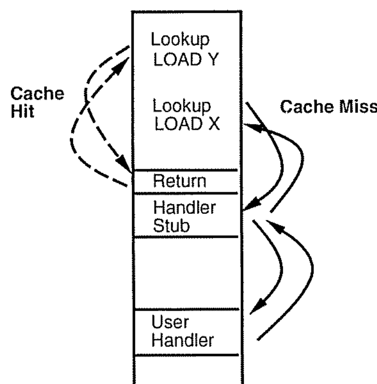

Figure 1: Fast-Cache Operation

This figure shows the execution of Fast-Cache for two cases: a cache hit and a cache miss. In this example the table lookup for the LOAD Y instruction indicates a cache hit and the handler stub simply returns. Conversely, a cache miss is detected for the LOAD X instruction; the handler stub saves processor state, and calls a user written handler.

specified for LOADS and STORES. Fast-Cache performs a table lookup before each memory reference to obtain the corresponding state, and invoke the appropriate user handler. The user handlers can be written in a high-level language, e.g., C, and are provided their own library routines and data segment, which prevents perturbation of simulation results.

Fast-Cache invokes user handlers through a handler stub that normally saves processor state. However, Fast-Cache can special case some functions and perform their entire action within the stub. Currently, Fast-Cache provides only one special function—the NULL handler—that optimizes cases that do not require any simulator action (see Figure 1). Fast-Cache can easily be extended to provide other predefined functions, such as incrementing an arbitrary counter, that avoid saving processor state. Although our current implementation optimizes the NULL handler by using a procedure call, and thus avoids saving the SPARC condition codes, on most machines (e.g., MIPS) we could optimize these functions with a compare and branch.

Although Fast-Cache may seem to allocate an extraordinary amount of state, a program's actual memory is limited by system resources (e.g., available swap space or maximum text, stack and, data sizes). Therefore, we do not have to allocate state for the entire 4 Giga-bytes in a 32-bit address space. Furthermore, since UNIX automatically zeros data pages on first reference, Fast-Cache need not explicitly initialize the block states and un-accessed state pages are never instantiated.

Fast-Cache inserts the table lookup instructions before each data reference[1] by rewriting existing executable files. Binary rewriting, also known as executable file rewriting [12], takes a program that is executable on an existing machine, adds instrumentation code and produces an executable that runs on the same machine. We discuss some of the advantages of binary rewriting in Section 3.3.

Under this new abstraction, the simulator is a set of handlers that control reference processing by manipulating memory block states with Fast-Cache provided routines. Fast-Cache imports the user handlers along with user startup and termination routines. Table 1 summarizes the interface between Fast-Cache and the simulator. In order to insert the appropriate instrumentation, Fast-Cache reads the following information from a configuration file (see Figure 2): the amount of state, the power-of-two block size, and the handler for each state.

The tagged memory block abstraction enables efficient simulation of a variety of memory systems. Simple simulations benefit primarily from the predefined NULL handler, whereas more complex simulations also benefit from Fast-Cache's direct invocation of simulator functions. For example, the property of inclusion [18] allows efficient simulation of multiple cache configurations in a single pass over the reference trace [7, 27]. However, with the trace abstraction, simulation time is much higher for caches that do not guarantee inclusion. The simulator must probe each cache individually to determine if the reference is a hit or a miss. Using the state of a memory block to encode which caches the block is in, Fast-Cache expedites this search by directly invoking a simulator function specialized to update the appropriate caches.

A simulation system should minimize perturbation of the application program's memory system behavior. Interleaving simulator data and application data changes data placement, and can significantly alter memory system behavior [11]. Fast-Cache avoids this problem by providing a separate data segment for the simulator and allowing users to specify alignment. Padding all additional text and data to this alignment guarantees that application data in the instrumented program maps to the same set as in the original, un-instrumented program. Fast-Cache also provides separate library routines for the user handlers, and thus prevents them from executing the instrumented version.

[1] Currently we do not instrument instruction fetches.

Fast-Cache Provided	`__fc_read_state(address);`	Returns the current state of the memory block that contains `address`.
	`__fc_write_state(address, state);`	Updates the state for the memory block containing `address`.
User Written	*user_handler*`(address);`	A user written function invoked by Fast-Cache's reference generator.
	`__fc_tgt_init();`	A user written routine called on simulation startup.
	`__fc_tgt_exit();`	A user written routine called on simulation exit.

Table 1: Fast-Cache Interface

```
/* Fast-Cache configuration file for a simple cache simulation */

state_bits 1            /* number of state bits per block */
lg2blocksize 5          /* log base 2 of the block size */
memory  65536           /* maximum amount of dynamically allocated memory */
align 65536             /* pad additional text and data to this size */
virtual_time  0         /* enable/disable instruction cycle counting */

LOADS
0       miss_handler    /* name of a user handler to call */
1       fc_return       /* predefined NULL handler */

STORES
0       miss_handler    /* name of a user handler to call */
1       fc_return       /* predefined NULL handler */
```

Figure 2: Fast-Cache Configuration File

Fast-Cache provides a framework for efficient memory system simulations by providing a tagged memory block abstraction. This abstraction allows Fast-Cache to highly-optimize cases that do not require simulator action, and improve performance when action is required. Fast-Cache also isolates simulator writers from the details of reference generation by providing a well defined interface for manipulating memory block states. In the next

section we evaluate the performance of Fast-Cache by comparing it to a conventional trace-driven simulator. Section 4 compares Fast-Cache, which implements memory block tags in software, to a hardware implementation of memory block tags.

3. A Comparison of Abstractions

We have currently implemented Fast-Cache for SPARC processors. In this section, we evaluate the performance of Fast-Cache on a SPARCstation 10 by comparing it to a trace-driven simulator that invokes a procedure for each memory reference. For simple miss counting, Fast-Cache simulations are two to three times faster than the procedure call implementation. Fast-Cache simulation times are 3.7 to 5.5 times slower than the original program. Calling a procedure on every memory reference runs 10 to 15 times slower. These results clearly show the advantage of optimizing for the common case, which the tagged memory block abstraction allows.

3.1. Methodology

Although there are many trace-driven simulators available, most are self-contained programs and execute independently of the reference generator. To make our comparison fair we modified Fast-Cache to implement our own single address space trace-driven simulator (PROC). Our implementation invokes a single simulator function before each memory reference; we insert four instructions to compute the effective address and jump to a handler stub, which is the same as the normal Fast-Cache stub. The stub saves processor state (including condition codes), calls the cache simulator, then restores the processor state. Although SPARC register windows allow saving most processor state with a single instruction, condition codes and some global registers must also be preserved across user handler invocations. We use a sequence of instructions to save and restore the condition codes [2], avoiding the overhead of system calls. Condition codes are seldom live [10], and we could eliminate saving and restoring them on some handler invocations; however, our current implementation does not perform the necessary analysis.

We use a metric called *slowdown* to evaluate simulation techniques. Slowdown is the simulation time divided by the execution time of the original, un-instrumented program. We use the average of two executions as measured with the UNIX `time` command. System time is included because the state bits may affect the virtual memory system.

Benchmark	Total Instructions	Data References	Reference Ratio	Miss Ratio
Compress	550,967,773	132,255,602	0.24	0.1257
Eqntott	1,196,745,802	219,396,712	0.18	0.0423
Gcc	122,217,042	30,150,226	0.25	0.0160
Tomcatv	1,350,644,358	481,774,508	0.35	0.0391
Xlisp	5,157,504,732	1,702,779,137	0.33	0.0016

Table 2: Benchmark Characteristics on a SPARCstation 10

The miss ratio is obtained for a 64 K-Byte cache with 32-byte blocks.

We present results for two simulations: All-Hits and Miss-Count. All-Hits, which models an ideal cache, indicates the overhead of detecting cache misses. Miss-Count, which counts misses in a 64 K-byte direct-mapped cache with 32-byte blocks, includes the overhead for processing cache misses. The PROC implementation of All-Hits calls a procedure that always performs a successful tag comparison. The Fast-Cache implementation of All-Hits uses the predefined NULL handler for all cache block states. For this comparison, Fast-Cache uses 8 state bits and we use an array for the cache data structure in PROC and the Fast-Cache miss handler. Section 3.3 discusses using fewer state bits in Fast-Cache.

We use 5 programs from the SPEC92 benchmark suite [19]: `compress`, `eqntott`, `gcc`, `xlisp`, and `tomcatv`.[2] All programs operate on the SPEC input files; for `gcc` we simulate only the `cc1` program operating on the single SPEC input file `1stmt.i`. All programs are compiled with `gcc`[25] version 2.4.5 at optimization level -O2. Program characteristics are shown in Table 2.

3.2. Results

The slowdowns for both Fast-Cache and PROC are presented in Figure 3. As shown, Fast-Cache significantly outperforms calling a procedure to simulate each memory reference. Fast-Cache simulation times are 3.7 to 5.5 times slower than the original program, while PROC ranges from 10 to 15 times slower. Fast-Cache also compares favorably with previously published slowdowns for uniprocessor simulations [2, 4, 16].

As the dark bars show, miss detection dominates. The overhead of invoking the simulator on every memory reference clearly dominates PROC simulation times. Fast-Cache significantly reduces the absolute overhead of miss detection, although it remains a large fraction of the simulation time. Intuitively, All-Hits simulation times should be a simple function of the *reference ratio*—the fraction of instructions that are memory references. Unfortunately, the combination of cache and pipeline effects prevents this. The SPARCstation 10 has a SuperSPARC processor, which is capable of issuing up to three instructions per cycle. Inserting a jump instruction before each memory reference can significantly affect the ability to issue multiple instructions per cycle.

PROC uses the same structure for miss detection as miss processing. By the time a miss is detected, the appropriate index into the cache data structure is available and the additional overhead to process the miss is negligible. Conversely, Fast-Cache uses a different structure for miss detection than for miss processing. The miss handler must still perform a lookup in the cache structure to obtain the replacement block and insert the new block. This makes Fast-Cache simulation time dependent on the miss ratio. For example, `xlisp` has the lowest miss ratio and exhibits the smallest overhead for miss processing, whereas `compress` has the highest miss ratio, and miss processing is a significant portion of simulation time.

3.3. Discussion

The results presented in this section clearly demonstrate the advantage of optimizing for cache hits. Fast-Cache uses the tagged memory block abstraction to optimize for the common case and significantly reduce simulation times. It also maintains the isolation of simulator writers from the details of reference generation. Although the data and code expansion caused by the memory tags and extensive in-lining could degrade memory system performance, we have not found this to be a problem.

Both Fast-Cache and PROC cause code expansion proportional to the static number of memory references, however, Fast-Cache inserts twice as many instructions as PROC. Fast-Cache inserts 9 instructions to perform the table lookup for a byte of state, and 15 instructions if memory block tags are less than 8 bits. PROC only requires 4

[2] We used a C-language version of `tomcatv` that was restructured to improve its cache behavior [8].

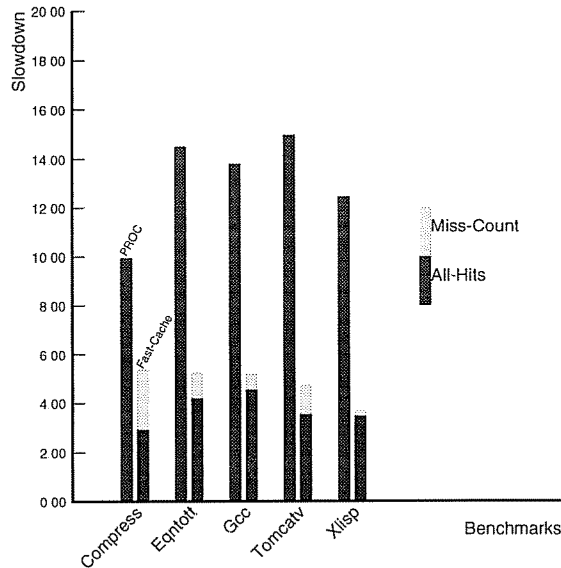

Figure 3: Fast-Cache Slowdowns

Fast-Cache outperforms the procedure call implementation (PROC) by a factor of 2 to 3. The dark bars clearly show that PROC is dominated by miss detection, and miss ratio has little influence on simulation time. Conversely, optimizing for cache hits makes Fast-Cache simulation time dependent on miss ratio.

instructions to generate the reference and invoke the simulator. Although we do not have quantitative results on the impact of code expansion, our results indicate that it has limited impact on performance.

However, we did find the number of instructions *executed* on each memory reference to significantly affect simulation times. Fast-Cache simulations that require bit extraction execute from 10% to 30% slower (e.g., a slowdown of 6 rather than 5) than those that take advantage of SPARC byte operations. However, they are still nearly twice as fast as PROC. This implies that in-lining the entire simulation will not improve performance much over tagged memory blocks, unless it significantly reduces the number of instructions for miss detection. For simple miss counting in a direct-mapped cache, we found that tagged memory blocks perform comparable to complete in-lining. However, in-lining the entire simulation has other disadvantages: complex simulations would require a large number of instructions to determine the appropriate action, it is difficult to isolate simulator writers from the details of reference generation, and code expansion could quickly become a problem.

We found very little variation between Fast-Cache simulators that use 1, 2, and 4 bits of state. This indicates that data expansion has negligible impact on simulation time. Fast-Cache's data expansion is proportional to the amount of memory a program may access and depends on the memory block size, whereas PROC allocates memory proportional to the size of the simulated cache. Although Fast-Cache requires much more memory, the memory block tags exhibit even better locality than data in the application program, and therefore do not degrade the performance of the SuperSPARC memory system.

Although our implementation of Fast-Cache uses binary rewriting on SPARC processors, the tagged memory block abstraction is independent of instrumentation methodology. We chose binary rewriting because it is easy to include library routines,[3] and it avoids the need to re-compile the instrumented program. The alternative techniques of source code annotation and assembly instruction instrumentation require source code, which may not be available. They also require re-compilation of the instrumented program, which is often time consuming. Source code annotation may alter register allocation, producing a significantly different reference trace. Binary rewriting and assembly instruction instrumentation preserve the memory reference pattern of the original application program since they occur after register allocation. For simulations that require accurate timing (e.g., analysis of bus traffic) assembly code instrumentation must correctly account for synthetic instructions that are expanded into multiple machine instructions by the assembler. Binary rewriting avoids this problem by instrumenting machine instructions. However, binary rewriting requires complex analysis to correctly insert instrumentation because it occurs after the program is linked. Recent advances in this area have produced systems that separate the details of binary rewriting from the insertion of instrumentation [14].

In some scenarios, it is necessary to include the instrumentation time in slowdown computations. We do not include binary rewrite time in our slowdown computation for two reasons: rewrite time is usually much lower than execution time or compilation time (see Table 4), and we are focussing on simulation techniques not instrumentation methodology. Other overheads of binary rewriting, e.g, indirect jumps and system calls, are included in simulation times, but are negligible.

The results presented in this section clearly demonstrate the advantage of optimizing for cache hits. Fast-Cache allows the simulator to control which references it processes by providing a tagged memory block abstraction. Using software implemented tags, Fast-Cache achieves speedups of two to three over a conventional trace-driven simulator, while maintaining the isolation of simulator writers from the details of reference generation.

4. A Comparison of Implementations

Although Fast-Cache uses software techniques to provide memory block tags, it is possible to implement them in hardware. The Wisconsin Wind Tunnel does this by using the the error correcting code (ECC) bits on a

Benchmark	Rewrite Time (sec)	Execution Time (sec)	Compilation Time (sec)
Compress	6.25	24.30	7.47
Eqntott	6.58	26.80	16.94
Gcc	31.53	3.80	400.23
Tomcatv	5.79	54.05	5.49
Xlisp	9.16	222.80	29.40

Table 4: Fast-Cache Binary Rewrite Time

[3] Currently we do not rewrite dynamically linked executables.

Thinking Machines CM-5 as valid bits, which eliminates software overhead for cache hits. In this section, we provide a quantitative comparison between Fast-Cache and a simulator that uses ECC bits. Our results show that Fast-Cache performs better or equal to ECC in all but one simulation, where the miss ratio is less than 0.1%. Fast-Cache simulation times are 3 to 6.7 times slower than the original program, while ECC executes 1.4 to 46 times slower. Although ECC optimizes cache hits, it incurs significant overhead for miss processing. Fast-Cache remains stable across simulations because it provides a better balance between miss detection and miss processing.

The Wisconsin Wind Tunnel uses ECC to provide hardware valid bits for simulated cache blocks [22]. Misses in the simulated cache also cause misses in the hardware cache, which in turn accesses physical memory marked with bad ECC. Only those cache blocks present in the simulated cache have valid ECC. References to invalid cache blocks generate a hardware trap that is redirected to the simulator (see Figure 4). The simulator chooses a block to replace, then copies the data to an internal buffer, and writes bad ECC to the physical memory location of the replaced block. The simulator writes valid ECC for the referenced block, then copies the referenced data from its internal buffer, and updates the tags of the simulated cache.

This technique has no software overhead for cache hits, but the miss processing overhead is very high, over 2500 cycles. This overhead can be divided into three components: trap, data movement, and ECC update. First, because the application program and the simulator run in separate address spaces, the trap overhead includes a round-trip between the application and simulator. Second, the error correcting code bits are not a true valid bit: changing the ECC bits requires overwriting the data. Therefore data that is not in the simulated cache must be kept in an internal simulator buffer. Finally, updating ECC is a privileged operation that must be performed in supervisor mode; its performance is dominated by the overhead of trapping to the operating system.

To isolate the individual overheads of trapping and data movement, we incrementally add them to a base Fast-Cache implementation. In the first extension (Fast-Cache+Trap), we modify the miss handler to trap to a separate process to update the state bits, see Figure 5. This trap overhead is slightly higher than ECC faults incur because it includes the Fast-Cache overhead to save/restore processor state. The next implementation, Fast-Cache+Trap+DM, adds the overhead of data movement by copying data to and from an internal buffer. Fast-Cache+Trap+DM is very similar to the ECC implementation, but has higher miss detection overhead and does not

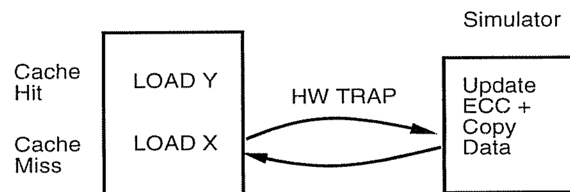

Figure 4: Miss Detection with ECC

This figure shows the execution of ECC for two cases: a cache hit and a cache miss. In this example the reference to Y is a cache hit and proceeds at hardware speed, while the reference to X is detected as a cache miss. The physical memory location of X has bad ECC and generates a hardware trap when referenced. The simulator updates the ECC, and copies the data from its internal buffer.

perform the system calls required to update ECC. All of the Fast-Cache implementations use 1 bit of state per cache block.

4.1. Methodology

Our evaluation environment is a significantly modified version of the Wisconsin Wind Tunnel running on a Thinking Machines CM-5. Since we are not investigating parallel processing, we retain only the code necessary for uniprocessor cache simulation (we were, however, still able to exploit parallelism by executing a different simulation on each node of the CM-5). Simulation times are presented as cycle counts, using the CM-5 cycle counters. We exclude I/O since it goes through the shared control processor. We use three of the benchmarks presented in Section 3: `compress`, `eqntott`, and `tomcatv`. The programs are modified to use file I/O so they execute correctly under the new environment (the CM-5 processing nodes do *not* run UNIX). `compress` and `eqntott` operate on the SPEC input and `tomcatv` executes 20 iterations on the SPEC input matrices.

We simulate four cache configurations: starting from a 16 K-byte direct-mapped cache with 32 byte blocks, we double the block size and quadruple the cache size up to a 1 M-byte cache with 256 byte blocks. Increasing both the cache size and block size reduces the miss ratio for all three programs (see Table 3), allowing us to examine the dependence of simulation time on miss ratio. The current ECC implementation does not simulate stack references or `LOADS` from the text segment. Our Fast-Cache implementations check for these references in the miss handler and mark their blocks valid for the remainder of the simulation[4].

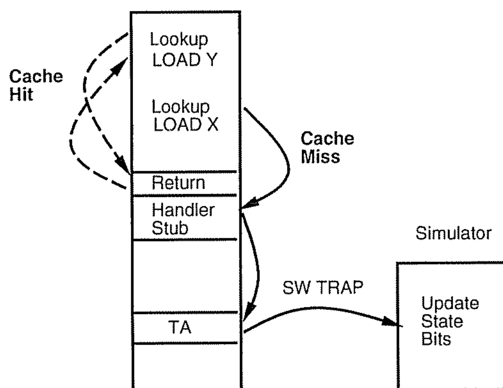

Figure 5: Fast-Cache With Trap

This figure shows the execution of Fast-Cache+Trap for two cases: a cache hit and a cache miss. In this example the table lookup for the LOAD Y instruction indicates a cache hit and the handler stub simply returns. Conversely, a cache miss is detected for the LOAD X instruction. The handler stub saves processor state and calls a user written handler that executes a software trap to a separate process, using the SPARC trap always (TA) instruction, where the state bits are updated.

[4] The miss ratios for these benchmarks do not change significantly when stack and text references are included.

Benchmark	16 K-Byte 32 Byte	64 K-Byte 64 Byte	256 K-Byte 128 Byte	1 M 256 Byte
Compress	0.1582	0.1195	0.0434	0.0007
Eqntott	0.0539	0.0332	0.0159	0.0045
Tomcatv	0.0735	0.0198	0.0098	0.0048

Table 3: Benchmark Miss Ratios

4.2. Results

Figure 6 shows the slowdowns of the four simulators presented above, where slowdown is the number of CM-5 cycles required for simulation divided by the number of cycles executed by the original program. We also show the slowdown for a Fast-Cache All-Hits simulation. The x-axis in Figure 6 is misses per cycle (MPC), obtained by dividing the number of cache misses by the cycle count of the original program. We chose MPC instead of miss ratio or misses per instruction because it allows comparison of results across benchmarks. Miss ratio is inadequate since it does not account for the memory reference rate of a program. For example, `tomcatv` has 35% references while `eqntott` has less than 20%. Ideally, we would plot memory reference slowdown versus miss ratio, but this requires factoring out cycles not attributed to memory references, which we can not do.

Fast-Cache slowdowns range from 3.1 to 6.7, and ECC ranges from 1.3 to 46. Two observations can be made from the data presented in Figure 6. First, Fast-Cache performs much better than ECC for most of our simulations. For low MPC, ECC performs very well, but as MPC increases, miss processing overhead dominates and the slowdown increases almost linearly. Fast-Cache exhibits much less dependence on MPC because it has much faster miss processing. Simulation time becomes more dependent on MPC as we add additional overhead for miss processing. The slight non-linearity exhibited by `compress` and `eqntott` is caused by cache effects on the CM-5.

The second observation is that simulation time is affected more by trap overhead than data movement. This is significant for ECC, because it usually requires four traps to process a single cache miss: one for the ECC fault, a second to mark the new block valid, a third to mark the replaced block invalid, and a fourth to trap back to the application program. The trap overhead is higher than it might otherwise be for several reasons: WWT's structure of separate address spaces, SPARC register windows, and a bug in the CM-5's cache controller. Nonetheless, for hardware valid bits to be better than Fast-Cache for even moderate-sized caches would require user-mode accessible valid bits [3] and fast user-level trap mechanisms [9]. However, it is unlikely that these features will be widely available in the near future.

5. Fast-Cache Applications and Extensions

The previous sections focus on counting misses for a specific cache configuration. However, Fast-Cache can be used for more complex simulations than simple miss counting. In this section we present some applications of Fast-Cache that take advantage of its generality. We also present some extensions that can easily be incorporated into our current implementation of Fast-Cache.

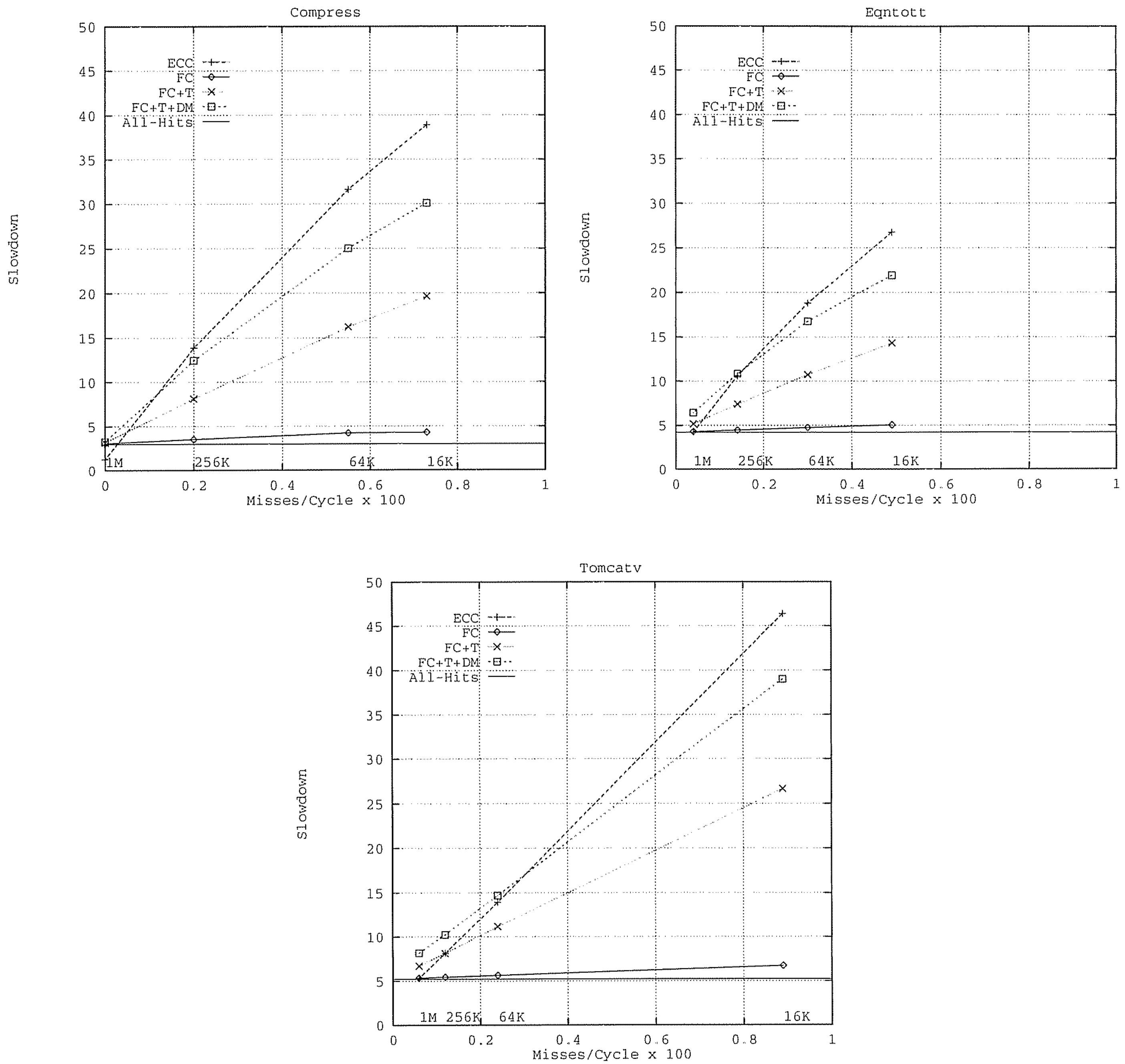

Figure 6: Fast-Cache vs ECC

This graph shows slowdown versus miss per cycle (MPC). Fast-Cache clearly outperforms ECC for high MPC, and is competitive at low miss rates. ECC performs better than Fast-Cache in only one case, where the MPC is less than 0.001, Although, MPC ranges from 0 to 0.01 in our graphs, it is not limited to this range.

5.1. Complex Simulations

The generality of the tagged memory block abstraction allows efficient simulation of memory systems much more complex than single-level caches. For example, Fast-Cache is currently being used to simulate a system with a TLB, two levels of cache, and a memory-mapped network interface. This simulation uses four state bits, one for each level of the memory system. The TLB and cache hierarchy handlers are similar to the handler described in Section 2. The network interface handler is much more complicated, modeling a user-level messaging system like in the CM-5 [15].

Fast-Cache provides accurate instruction cycle counts and can be used for timing dependent simulations, such as prefetching and write buffers. User handlers access the current value of the cycle counter through a global variable.

Fast-Cache can be used to simulate set-associative caches as well. A particular Fast-Cache implementation depends on the policy for replacing a block within a set. Random replacement can use an implementation similar to the direct-mapped cache, calling a handler only when a block is not resident in the cache. A Fast-Cache implementation of least recently used (LRU) replacement can optimize references to the most recently used (MRU) block since the LRU state does not change. References to MRU blocks would invoke the NULL handler, while all other references invoke the simulator. This is similar to Puzak's trace filtering for set-associative caches [21]; the property of inclusion [18] indicates the number of references optimized is equal to the number of cache hits in a direct-mapped cache with the same number of sets as the set-associative cache. A further optimization distinguishes misses from hits to non-MRU blocks by using more than two states per cache block. An example configuration file is shown Figure 7.

The performance of simulators that evaluate multiple cache configurations can also be improved by using Fast-Cache. For example, a Fast-Cache implementation that uses a binomial tree[27] to simulate multiple cache configurations would optimize references that hit at the root of the binomial tree. The binomial tree reduces the search time when determining what action to take. A Fast-Cache simulation can eliminate this search by using the memory block state to encode its position in the tree, and directly invoking a simulator function that is optimized to update the tree appropriately. As mentioned in Section 2, directly invoking simulator functions based on the state of a memory block also allows efficient simulation of multiple cache configurations that do not guarantee inclusion.

```
STATE   Handler           Comment

0       miss_handler    /* called for blocks not in the cache */
1       non_mru_hit     /* cache hits to non-mru blocks */
2       fc_return       /* cache hits to mru blocks */
3       fc_return       /* unused */
```

Figure 7: State Specification for Set-Associative Cache with LRU Replacement

5.2. Cache Profiling

Fast-Cache can be extended to provide the data necessary to produce a cache profile [8, 17]. A cache profile identifies code sections and data structures that exhibit poor cache behavior. A very simple profile displays the number of cache misses for each memory reference. Fast-Cache can be extended to provide the program counter for each memory reference by adding an additional parameter to the user handler invocations. More sophisticated cache profiles can be created by using different handlers. For example, CPROF [8] classifies cache misses as compulsory, capacity, conflict, or anti-conflict. Fast-Cache's generality allows invocation of a separate handler for each miss type.

5.3. Instruction Fetch

Fast-Cache currently does not support instruction fetch simulation. However, it can easily be extended to use a combination of table lookup and static analysis[29] to efficiently simulate instruction fetches. Fast-Cache knows the program counter for each instruction when it is rewriting the executable. For split instruction and data caches, at the beginning of each basic block; Fast-Cache can perform a table lookup for instructions that occupy unique cache blocks. For unified caches, exact simulation requires checking at a finer grain, however the added accuracy is probably not worth the extra overhead.

6. Related Work

Fast-Cache builds on the observation that in most simulations, the common case requires no action. This observation has been used in several earlier efforts to improve memory system simulation. MemSpy [16] optimizes for cache hits by saving only the registers necessary to determine if a reference is a hit or a miss; hits branch around the remaining register saves and miss processing. Although we cannot directly compare MemSpy's performance to Fast-Cache, there is an important disadvantage of this approach. MemSpy tightly integrates reference generation and simulation, but does so in an ad hoc way. The miss detection code must be written in assembler, so the appropriate registers may be saved, and must be modified for each different memory system. By presenting the tagged memory block abstraction, Fast-Cache hides these low-level details from simulator writers, allowing the entire simulator to be written in C.

Other work has focussed on reducing the size of reference traces by filtering out references that would hit. Smith[24] proposed doing this by deleting references to the n most recently used blocks. The subsequent trace can be used to obtain correct miss counts for fully associative memories that use LRU replacement with more than n blocks. Puzak [21] extended this work to set-associative memories by filtering references to a direct-mapped cache. However, these techniques are of little or no use with on-the-fly simulators.

Simulators that evaluate several cache configurations in a single pass over the reference trace [7, 18, 27, 28] use the property of inclusion to limit the search for caches that contain a given block. The portion of the simulator modeling larger caches only requires action when the block is not contained in all smaller caches. Fast-Cache can improve the performance of these simulators by filtering references that hit in all caches. Multiple cache configurations that do not guarantee inclusion can also be simulated with Fast-Cache. The tagged memory block abstraction enables Fast-Cache to eliminate this search by directly invoking a simulator function to update the appropriate caches. Again, references that hit in all simulated caches would invoke the NULL handler.

Several simulation systems reduce simulation time by executing the reference generator and the simulator in the same address space [1, 2, 4, 20, 26]. Some of these systems call a procedure before each memory reference [4, 20], incurring the overhead of saving and restoring processor state for each memory reference executed. Other systems write the trace to a shared buffer and invoke the simulator when the buffer is full [1] or at basic block boundaries [26]. These systems amortize the overhead of saving and restore processor state over several memory references.

Most of the above systems instrument assembly code. Shade[2] provides an alternative technique to instrumentation by interpreting each instruction. For tracing, Shade writes addresses to an internal buffer and calls a trace analyzer when the buffer is full. Shade also allows users to gain control at any point in the simulation, by specifying instructions that should invoke the simulator.

7. Conclusion

The performance of conventional simulation systems is limited by the simple interface—the reference trace abstraction—between the reference generator and the simulator. This paper presents Fast-Cache, a software system that provides a tagged memory block abstraction—a new method for memory system simulation. Fast-Cache provides an efficient simulation framework by: optimizing cases that do not require simulator action, rapidly invoking specific simulator functions when action is required, isolating simulator writers from the details of reference generation, and providing simulator portability. Fast-Cache associates state with each memory block, and directly invokes simulator functions based on this state. The simulator manipulates memory block states to control which references it processes. Fast-Cache provides a special NULL function to expedite the processing of references that do not require simulator action.

Fast-Cache can be used for complex memory system simulations, however in this paper we concentrate on the performance of simple cache simulation. Fast-Cache simulation times are 3.7 to 5.5 slower than the original, uninstrumented program on a SPARCstation 10, while a traditional trace-driven simulator that calls a procedure for each memory reference is 10 to 15 times slower. We also show that Fast-Cache's software implemented memory block tags outperforms a hardware implementation of memory block tags (ECC bits on a CM-5) because it provides a better balance between miss detection and miss processing. Fast-Cache slowdowns range from 3.1 to 6.7, while ECC slowdowns range from 1.3 to 46. Our results show that Fast-Cache performs significantly better than ECC for simulated miss ratios greater than 2%. ECC outperforms Fast-Cache in only one of our experiments, where the miss ratio was less than 0.1%.

As the impact of memory hierarchy performance on total system performance increases, hardware and software developers will increasingly rely on simulation to evaluate new ideas. Fast-Cache provides the mechanisms necessary for efficient memory system simulation by using the tagged memory block abstraction to optimize for the common case.

Acknowledgements

We would like to thank James Larus for providing access to QPT. James Larus, Steve Reinhardt, Mitali Lebeck, Shubhendu Mukherjee, Madhusudhan Talluri, and Brad Richards provided comments and suggestions on early drafts of this paper. This work is supported in part by National Science Foundation Presidential Young Investigator awards CCR-9157366, and MIPS-8957278, National Science Foundation grants CDA-8920777, CCR-9101035, and MIP-9225097, donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, and Xerox Corporation, and the Graduate School of the University of Wisconsin.

References

1. Borg, A., Kessler, R. E., and Wall, D. P., "Generation and Analysis of Very Long Address Traces," *Proceedings 17th Annual International Symposium on Computer Architecture*, pp. 270-279 (May 1990).
2. Cmelik, R. F. and Keppel, D., "Shade: A Fast Instruction-Set Simulator for Execution Profiling," Technical Report UWCSE 93-06-06, Univeristy of Washington (June 1993).
3. Dally, W. J. and Willis, D. S., "Universal Mechanisms for Concurrency," *In Proceedings PARLE89*, (June 1989).

4. Davis, H., Goldschmidt, S. R., and Hennessy, J., "Multiprocessor Simulation and Tracing using Tango," *Proceedings of ICPP* **II** pp. 99-107 (August 1991).

5. Gee, Jeffrey D., Hill, Mark D., Pnevmatikatos, Dionisios N., and Smith, Alan Jay, "Cache Performance of the SPEC92 Benchmark Suite," *IEEE Micro* **13**(4) pp. 17-27 (August 1993).

6. Grunwald, D., Zorn, B., and Henderson, R., "Improving the Cache Locality of Memory Allocation," *Proceedings PLDI*, pp. 177-186 (1993).

7. Hill, M. D. and , A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers* **38**(12) pp. 1612-1630 (December 1989).

8. Hill, Mark D., Larus, James R., Lebeck, Alvin R., Talluri, Madhusudhan, and Wood, David A., "Wisconsin Architectural Research Tool Set," *Computer Architecture News* **21**(4) pp. 8-10 (August 1993).

9. Johnson, D., "Trap Architectures for Lisp Systems," *Proceedings of ACM Conference on Lisp and Functional Programming*, pp. 419-429 (June 1990).

10. Kessler, P. B., "Fast Breakpoints: Design and Implementation," *In Proceedings of PLDI '90*, pp. 78-84 (Jun 1990).

11. Kessler, R. E. and Hill, Mark D., "Page Placement Algorithms for Large Real-Index Caches," *ACM Trans. on Computer Systems* **10**(4) pp. 338-359 (November 1992).

12. Larus, J. R. and Ball, T., "Rewriting Executable Files to Measure Program Behavior," *to appear in Software Practice & Experience*, ().

13. Larus, J. R., "Efficient Program Tracing," IEEE Computer (May 1993).

14. Larus, J. R. and Schnarr, E., "EEL: A Library for Editing Executable Files," *To be submitted for publication.*, ().

15. Leiserson, et al., Charles E., "The Network Architecture of the Connection Machine CM-5," *Proc. ACM Symposium on Parallel Algorithms and Architectures*, (July 1992).

16. Martonosi, M., Gupta, A., and Anderson, T., "Effectiveness of Trace Sampling for Performance Debugging Tools," *Performance Evaluation Review* **21**(1) pp. 248-259 (May 1993).

17. Martonosi, M., Gupta, A., and Anderson, T., "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Performance Evaluation Review* **20**(1) pp. 1-12 (June 1992).

18. Mattson, R. L., Gecsei, J., Schultz, D. R., and Traiger, I. L., "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal* **9**(2) pp. 78-117 (1970).

19. Newsletter, SPEC, December 1991.

20. Porterfield, A., "Software Methods for Improvement of Cache Performance on Supercomputer Applications," Ph. D. Thesis, Dept of Computer Science, Rice University, (1989).

21. Puzak, T. R., "Analysis of Cache Replacement Algorithms," Ph. D. Thesis, Dept. of Electrical and Computer Engineering, University of Massachusetts (February 1985).

22. Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A., "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (May 1993).

23. Smith, A. J., "Cache Memories," *ACM Computing Surveys* **14**(3) pp. 473-530 (Sept. 1982).

24. Smith, A. J., "Two Methods for Efficient Analysis of Memory Address Trace Data," *IEEE Transactions on Software Engineering* **3**(12)(January 1977).

25. Stallman, R., *The GNU Project Optimizing C Compiler,* Free Software Foundation INC. ().

26. Stunkel, C. B. and Fuchs, W. K., "TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation," *Proc. SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 70-78 (May 1989).

27. Sugumar, R. A. and Abraham, S. G., "Efficient Simulation of Multiple Cache Configurations using Binomial Trees," *Technical Report CSE-TR-111-91*, (1991).

28. Wang, W. and Baer, J., "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis," *Proc. SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 27-36 (May 1990).

29. Whalley, David B., "Fast Instruction Cache Performance Evaluation Using Compiler-Time Analysis," *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Syst ems*, pp. 13-22 (May 1992).