ACM Transactions on Modeling and Computer Simulation. vol 7, no 1. January, 1997. pp. 7-41.

Trap-driven Memory Simulation with Tapeworm II[†]

RICHARD UHLIG, DAVID NAGLE, TREVOR MUDGE, and STUART SECHREST
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122

Trap-driven simulation is a new approach for analyzing the performance of memory-system components such as caches and translation-lookaside buffers (TLBs). Unlike the more traditional trace-driven approach to simulating memory systems, trap-driven simulation uses the hardware of a host machine to drive simulations with operating-system kernel traps instead of with address traces. As a workload runs, a trap-driven simulator dynamically modifies access to memory in such a way as to make memory traps correspond exactly to misses in a simulated cache structure. Because traps are handled inside the kernel of the host operating system, a trap-driven simulator can monitor all components of multi-task workloads including the operating system itself. Compared to trace-driven simulators, a trap-driven simulator causes relatively little slowdown to the host system because traps occur only in the infrequent case of simulated cache misses. Unfortunately, because they require special forms of hardware support to cause memory-access traps, trap-driven simulators are difficult to port, and they are not as flexible as trace-driven simulators in the types of memory configurations that they can model.

Several researchers have recently begun to use trap-driven techniques in their studies of memory-system design tradeoffs, but little is know about how the speed and accuracy of the technique varies with the type of simulations conducted, or about the nature of its drawbacks with respect to portability and flexibility. In this paper, we use a prototype trap-driven simulator, named Tapeworm II, to explore these issues. We expose both the strengths and the weaknesses of trap-driven simulation with respect to speed, accuracy, completeness, portability, flexibility, ease-of-use, and memory overhead. Although the results are drawn from a specific implementation of trap-driven simulation, we believe that many of our results from Tapeworm hold true for trap-driven simulation in general.

Keywords: memory system, cache, TLB, simulation, trace-driven simulation, trap-driven simulation

1. INTRODUCTION

Trace-driven simulation is one of the most popular methods for evaluating memory-system architectures consisting of caches and translation-lookaside buffers (TLBs) [Smith82, Holliday91]. With trace-driven simulation, a stream of memory references generated by some workload of interest is first collected from an existing host machine, and then passed to a simulator that emulates the behavior of some yet-to-be-built cache or TLB. At its core, a trace-driven simulator executes a loop similar to that shown on the left side of Fig. 1. The processing steps include obtaining the next address in the trace, searching for that address in a simulated cache, and then invoking a replacement policy in the event of a miss. The trace addresses can come from a file created by a trace-extraction tool, or they might be

[†]This article is an expanded version of a paper that originally appeared at the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), 1994. This work was supported by ARPA Contract #DAAH04-94-G-0327, by NSF Contract #CISE9121887, and by a National Science Foundation Graduate Fellowship.

Trace-driven Simulation

```
while (addr = next_addr(trace)){
   if (search(addr))
       hit++;
   else {
       miss++;
       replace(addr);
   }
}
```

Trap-driven Simulation

```
traps invoke trap_handler(addr):

trap_handler(addr){
   miss++;
   set_access(addr, fullAccess);
   replAddr = replace(addr);
   set_access(replAddr,
noAccess);
```

Fig. 1. Trace-driven versus Trap-driven Simulation Algorithms. The core execution loops of trace-driven and trap-driven simulators. This code omits many details of actual simulation, such as the treatment of writes and assigning penalties for different types of misses (e.g., in a critical-word-first cache). Simulators that evaluate multiple memory configurations in a single trace pass also have a more complex structure [Mattson70; Hill87; Thompson89; Sugumar93].

generated on-the-fly by an annotated workload [Agarwal86, Borg90, Eggers90, Holliday91, Smith91, Cmelik94]. The search procedure involves indexing a data structure that represents the cache and then, depending on the associativity of the cache, performing one or more comparisons to test for a hit. Though a simple operation, the search and test must be performed for every address in the trace.

Trace-driven simulation has worked well in the design of memory systems supporting single-task applications such as those found in the SPEC benchmark suite [SPEC91, SPEC93, Gee93]. However, a growing body of work is revealing that memory systems tuned to such workloads do not perform well on more complex, multi-task workloads that frequently invoke operating-system services [Agarwal88, Anderson91, Chen93, Cvetanovic94, Mogul91, Nagle93, Nagle94, Uhlig95, Ousterhout89]. Unfortunately, most trace-driven simulation tools are ill-suited to analyzing workloads of this type because they are often limited to single, user-level tasks [Holliday91, Cmelik94]. Trace-collection tools that *can* monitor multi-task and OS-kernel activity rely either on expensive hardware monitoring equipment, or require a cumbersome preprocessing step to statically annotate all executable files that a multi-task workload might use [Mogul91, Sites96]. The resulting annotated-executable files consume additional disk space as well as physical memory, and thus typically require a special host machine loaded with extra storage. Another problem common to trace-driven simulators is speed; trace-driven simulations are slow because every memory reference, whether or not it causes a change in simulated cache state, must be collected and processed.

One way to overcome some of the limitations of traditional trace-driven simulation is to directly drive a simulator with memory-access traps caused by a workload as it runs on a host machine [Reinhardt93, Nagle93, Talluri94]. A *trap-driven simulator* begins by restricting access to all memory locations in a workload's address space. These inaccessible locations represent regions of memory that are not currently resident in some pre-defined, simulated cache structure. As the workload executes, the first reference to each location causes a trap (corresponding to a simulated cache miss), which is directed to a *trap handler* (see the right side of Fig. 1). The handler counts the miss and then makes the required memory location accessible. This action effectively caches the memory location in the simulated cache structure because subsequent references to this location will proceed at full hardware speed without trapping. As the simulated cache fills, new cache lines will begin to conflict with lines already held in the cache (as occurs in an actual hardware cache). Therefore, in a final step, the handler emulates cache conflicts by restricting access to a displaced memory location, in accordance with some replacement policy.

This paper is about a trap-driven simulator, named *Tapeworm II*, that offers two principal advantages: (1) completeness and (2) speed. Tapeworm simulations are complete because traps originating from any user task, as well as those generated by OS kernel activity are captured. Tapeworm simulations are fast because its trap handlers are invoked only in the uncommon case of a simulated TLB or cache miss. Tapeworm offers other advantages as well: it requires no pre-processing of a workload before monitoring begins, and it adds little memory overhead.

Despite these advantages, trap-driven simulation does suffer some drawbacks. Although capable of simulating TLBs and caches with a range of sizes, associativities, and replacement policies, trap-driven simulation is generally less flexible than trace-driven approaches with respect to the simulation of other architectural structures, such as write buffers or instruction pipelines. Tapeworm, for example, is able to simulate a range of TLBs and instruction caches, but is unable to simulate data caches because it lacks the necessary support from the particular host machine on which it runs.[1] A second problem is portability; trap-driven simulation requires some mechanism for controlling access to memory on the host machine, a feature that may not be fully supported (especially at a fine-grained level) by the host hardware. Finally, a trap-driven simulator's presence in a system can introduce new forms of measurement bias and variability. Though not strictly a disadvantage, trap-driven simulations are more sensitive to inherent variations in memory performance in an actual running system, an effect that is generally not accounted for in trace-driven simulation studies.

The remainder of this paper examines the pros and cons of trap-driven simulation in greater detail. Section 3 gives a detailed description of the design and implementation of the Tapeworm II trap-driven simulator, which will serve in Section 4 as our prototype for examining the strengths and weaknesses of trap-driven simulation in general. We begin with a discussion of related work in the next section.

2. RELATED WORK

Trace-driven simulation has been used to evaluate memory systems for decades. In his 1982 survey of cache memories, A. J. Smith gives examples of trace-driven memory-system studies that date as far back as 1966 [Smith82]. Holliday has surveyed trace-driven simulation methods for both uniprocessor and multiprocessor memory-system design [Holliday91], Stunkel et al. have studied trace-driven simulation in the specific context of multiprocessor design [Stunkel91], and a more-recent survey of trace-driven tools and techniques can be found in [Uhlig96].

1. We explain in Section 4.4 the problems that we encountered simulating data caches in our prototype implementation of Tapeworm II. It should be noted, however, that there are no *inherent* limitations to trap-driven simulation that prevent data-cache simulation (as work on the Wisconsin Wind Tunnel has shown [Reinhardt93]).

Early work on trace-collection tools that capture complete system activity generally involved designing special monitoring hardware for an existing machine, or modifying its microcode [Clark83, Alexander85, Agarwal86]. Similar approaches have been adopted in some more recent tools [Flanagan92, Nagle92, Torrellas92], but researchers have noted that these hardware-based approaches are typically costly to implement and suffer from problems of portability. Recent work overcomes these limitations by extending software-based code-annotation techniques [Eggers90, Smith91, Srivastava94] to include multi-process and OS activity [Mogul91, Chen94, Sites96]. A promising new approach, used by SimOS, makes OS monitoring easier by running the kernel executable inside a user-level process that acts as a virtual hardware platform [Rosenblum95]. When combined with fast emulation techniques that use dynamic binary translation [Cmelik94], SimOS is able to drive memory-system simulators with address traces that include complete system activity [Witchel96].

A few simulators avoid memory-reference traces altogether and are driven, instead, by kernel traps. An early example of this approach is the first generation Tapeworm, which performs TLB simulation [Nagle93]. This system intercepts kernel traps to the software-managed TLB miss handlers of an R2000-based workstation to drive a TLB simulator. Because all user and kernel misses are intercepted, Tapeworm is able to fully consider multi-task and OS effects. Talluri describes a similar trap-driven TLB simulator that runs on SPARC-based workstations [Talluri94]. Another early trap-driven simulator, the Wisconsin Wind Tunnel (WWT) simulator causes traps by modifying the error-correcting code (ECC) check bits in a SPARC-based CM-5 [Reinhardt93]. Unlike Tapeworm, which performs only uni-processor simulations but includes multi-task and OS kernel references, WWT is designed to investigate multi-processor cache coherence algorithms but is limited to user-level activity of a single task.

Other work shares properties of both trace-driven and trap-driven simulation [Martonosi92, Cmelik94, Lebeck95]. Like traditional annotation-based trace collectors, these hybrid approaches annotate a program to invoke simulation handlers on every memory reference. They differ from standard annotation in their support for an optimization where a null handler is called on memory references known to be satisfied by a simulated cache or TLB.

This paper advances previous work in two significant ways. First, it describes the design of a second-generation Tapeworm which combines the OS-capable features of the original Tapeworm TLB simulator with a WWT-like mechanism for setting fine-grained memory traps. The resulting simulator is capable of both instruction-cache and TLB simulation and captures multi-task and OS kernel activity. Second, using Tapeworm II as a prototype, we investigate the pros and cons of trap-driven simulation in general. We cover, in particular, the *flexibility*, *speed*, *accuracy*, and *portability* of trap-driven simulation, issues that have not been well explored by early work on this new memory-simulation method. We use a benchmark suite (described in Section 4) to illustrate certain points in our comparisons with trace-driven simulation, but a detailed report of simulation *results* obtained with Tapeworm II is beyond the scope of this paper. Tapeworm and Tapeworm II have, however, been used in several case studies of interactions between operating-system structure and memory architecture [Nagle93, Nagle94, Uhlig95].

Table I. Hardware-dependent Tapeworm Primitives. This is an idealize interface. For reasons discussed below, the prototype implementation of Tapeworm II does not support the full functionality of this interface.

Routine	Description
`tw_set_access(pa, size, state)`	Set the access state of the memory region containing `pa` (a physical address). The operation is performed on memory boundaries that align to `size` bytes, starting at (`pa` div `size`) and extending for exactly `size` bytes. All subsequent references to memory locations in this range are checked against the trap `state`, which may be one of three values: `noAccess`, `readAccess` or `fullAccess`. Memory accesses that violate the access rights on a given memory location should result in an OS kernel trap that passes control to Tapeworm with a call to `tw_trap()` (see below), while valid accesses should proceed at full hardware speed.
`tw_get_access(pa)`	Return the access state for the memory location at physical address `pa`.
`tw_trap(pa, va, type)`	This routine, the entry point to the Tapeworm trap handler, is invoked by the host hardware whenever an access violation occurs. The host hardware should provide the physical address (`pa`) and virtual address (`va`) of the violating memory reference, and the type of memory reference (`type`), which can be a `dataLoad`, a `dataStore` or an `instrFetch`.
`tw_get_counts(type)`	Returns a count of occurrences of a given `type` of event (e.g., `dataLoad`, `dataStore`, `instrFetch`, `instrExec`). Only references to pages added to the Tapeworm domain should be counted (see `tw_add_page()` and `tw_remove_page()` in Table II).

3. TAPEWORM II DESIGN AND IMPLEMENTATION

We outlined the essential ideas behind trap-driven simulation in the introduction. In this section we expand our description of trap-driven simulation by describing the Tapeworm II design in detail. In particular, we will cover how Tapeworm controls memory access, how it handles traps, and how it interacts with the host operating system.

3.1 Hardware-dependent Tapeworm Primitives

We begin by describing a collection of primitive functions that Tapeworm requires from the host hardware that it runs on. To enhance portability, Tapeworm collects these functions together into a single interface, shown in Table I. This interface is ideal, in the sense that if a given port of these primitives is fully supported, then Tapeworm will be able to simulate the full range of memory configurations that it was designed for. For some hosts, however, it many not be possible to support the full semantics of the interface without modifications to the host hardware (this was the case with our prototype implementation of these primitives on a DECstation 5000/200). A partial implementation of the primitives still proved useful, although the range of simulations we could perform was somewhat restricted (e.g., Tapeworm II supports TLB and I-cache simulation, but cannot simulate D-caches).

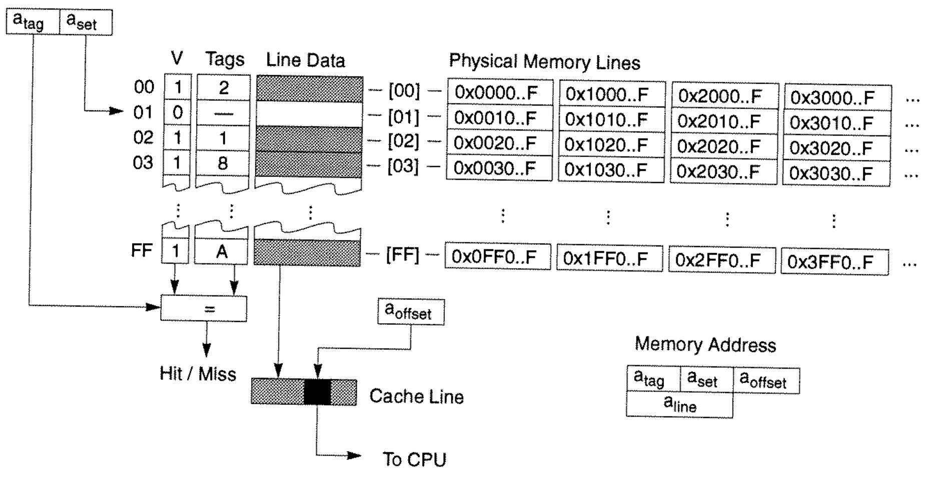
The first three routines, `tw_set_access()`, `tw_get_access()`, and `tw_trap()`, form the core of this interface; they enable Tapeworm to control the trapping mechanisms of the host hardware. To support the maximum flexibility in memory simulations, an implementation of `tw_set_access()` should support a wide a range of values in the `pa`, `size`, and `state` parameters. To enable multi-task and OS memory simulations, values of `pa` referring to any user or kernel memory location should be permitted. To enable both TLB and cache simulations, values of `size` ranging from as small as a cache line (4 or 8 words) to as large as a page (4 KB or 8 KB) should be supported. Finally, all three access (trap) states,[1] `noAccess`, `readAccess`, and `fullAccess`, should be supported to enable both I- and D-cache simulations.

Because most host hardware does not directly support fine-grained access control, implementing the full functionality of these primitives is difficult. We have experimented with three different ad-hoc approaches: (1) flipping page-valid bits in the OS page-table structure (to cause page-fault traps), (2) dynamically replacing/restoring instruction breakpoints in a workload's text segment (to cause program-debugging traps), and (3) modifying error-correcting code (ECC) bits in the host machine's main memory (to force memory-error traps). The latter method was first proposed and implemented by Reinhardt et al. [Reinhardt93]. Each of these approaches has its limitations. Although page-valid bits enable access control to both data and text (program) memory, they can only control access at the granularity of a page. Breakpoints enable finer-grained access control, but they only work on text memory and they require some mechanism for saving the original breakpointed instructions. Finally, ECC-bit modification is very platform-dependent, and not all machines support error-correcting memory. Despite their limitations, these techniques for controlling memory access enabled us to build a prototype trap-driven simulator without requiring the design of any additional hardware.

The final routine in the interface, `tw_get_counts()` is used to obtain event counts, which are combined with the base metrics obtained Tapeworm's trap handlers (i.e., miss counts), to compute a variety of other performance metrics. To support the computation of a range of performance metrics, this routine should report counts of memory load and store references, as well as instruction fetches and number of instructions executed. As with the access-control routines, we were forced to use ad-hoc methods (a logic analyzer connected to the host machine running the Tapeworm II prototype) to obtain the event counts necessary for implementing the `tw_get_counts()` routine.

These hardware-dependent primitives form an interface that is very similar to the memory-protection model supported by most microprocessor memory-management units. The important difference is that protection is provided at a finer granularity. Similar fine-grained access-control interfaces have been proposed for systems that implement distributed shared memory [Appel91; Reinhardt94]. This interface differs slightly in its orientation to trap-driven simulation, including the addition of the event-counting routine `tw_get_counts()`.

1. *Access states* refer to those set by a trap-driven simulator using `tw_set_access()`. These levels of access are always a lower than the page-level access rights granted to a workload by the host VM system.



a_{tag} a_{set}
V Tags Line Data
Physical Memory Lines
00 01 02 03 FF
[00] [01] [02] [03] [FF]
0x0000..F 0x1000..F 0x2000..F 0x3000..F
0x0010..F 0x1010..F 0x2010..F 0x3010..F
0x0020..F 0x1020..F 0x2020..F 0x3020..F
0x0030..F 0x1030..F 0x2030..F 0x3030..F
0x0FF0..F 0x1FF0..F 0x2FF0..F 0x3FF0..F
a_{offset}
Hit / Miss
Cache Line
To CPU
Memory Address
a_{tag} a_{set} a_{offset}
a_{line}

Fig. 2. Direct-mapped Cache Hardware. A 4-KB cache with 16-byte lines.

3.2 Tapeworm Trap Handling

With the hardware-dependent primitives defined, we can now present a more detailed description of the Tapeworm trap handlers for a simple simulation case. Fig. 2 shows the hardware for a simple direct-mapped cache. The address (a) that is used to access cache is divided into three parts. The middle part (a_{set}) is used to select a cache set, the high-order bits of the address (a_{tag}) are used to compare against the cache tags, and the low-order bits (a_{offset}) are used to select the appropriate instruction from the cache line. Notice that the combination of bits a_{tag} and a_{set} completely define the memory line. We therefore sometimes refer to the concatenation of these bits as a_{line}. The *memory equivalence class* of an address, denoted by [a], is:

$$[a] = \{\text{all addresses, } b\text{, such that } (a_{set} = b_{set})\} \qquad \text{(Eqn 1)}$$

The cache in Fig. 2 has 256 (FF in hex) memory-equivalence classes, the elements of which are shown as a row of memory locations to the right of each cache set. The concept of a memory equivalence class is important because it specifies precisely the subset of memory locations that a given cache structure can hold. A direct-mapped cache, for example, is constrained to hold at most one line from each memory-equivalence class at a time, while a set in an N-way set-associative cache may hold up to N lines from a given memory-equivalence class.

With these definitions in place, we can now explain how a trap-driven simulator models a direct-mapped cache. Fig. 3 shows that Tapeworm represents a direct-mapped cache with a simple data structure (`cache[]`) that holds the starting addresses of cached memory lines, and a variable (`misses`) that counts the number of references that miss the simulated cache during the run of a workload.

Tapeworm's main task is to continually update these data structures so that they mirror what the state of an hypothetical hardware cache would be, if it were running the same workload. Tapeworm accomplishes this by constraining access to the host memory in a way that causes a trap to occur whenever the workload makes a memory reference that would result in a change of simulated cache state.

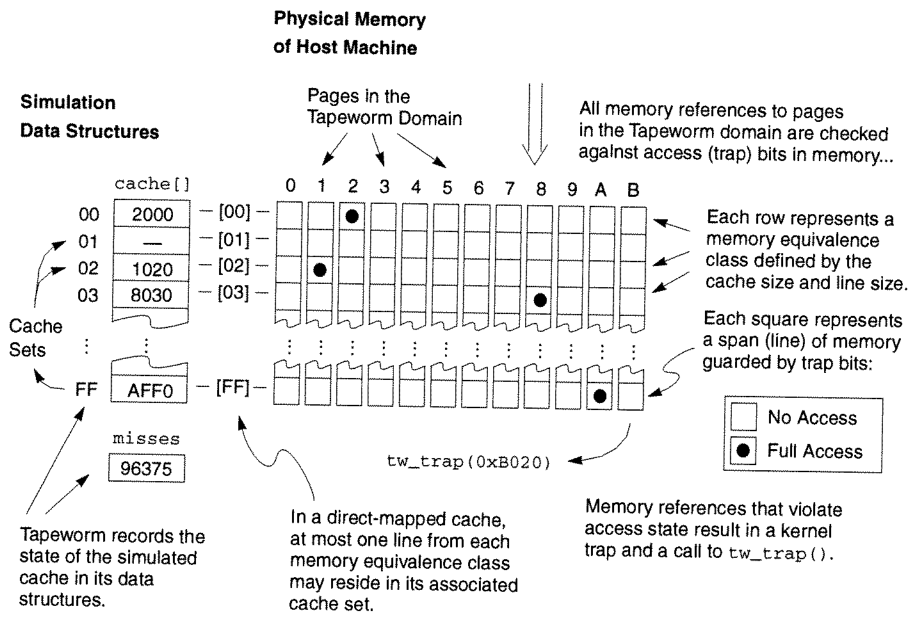

Physical Memory of Host Machine
Simulation Data Structures
Pages in the Tapeworm Domain
All memory references to pages in the Tapeworm domain are checked against access (trap) bits in memory...
cache[]
0 1 2 3 4 5 6 7 8 9 A B
00 2000 [00]
01 — [01]
02 1020 [02]
03 8030 [03]
Cache Sets
FF AFF0 [FF]
Each row represents a memory equivalence class defined by the cache size and line size.
Each square represents a span (line) of memory guarded by trap bits:
No Access
Full Access
misses
96375
tw_trap(0xB020)
Tapeworm records the state of the simulated cache in its data structures.
In a direct-mapped cache, at most one line from each memory equivalence class may reside in its associated cache set.
Memory references that violate access state result in a kernel trap and a call to tw_trap().

Fig. 3. Data Structures and Host Memory for Direct-mapped Cache Simulation. This figure shows the data structures and host physical memory required for trap-driven simulation of the direct-mapped cache shown in Fig. 2. In a direct-mapped cache, at most one line from each main-memory equivalence class may be held in the cache at any point in time. Tapeworm emulates this constraint by permitting access to at most one line in each main-memory equivalence class.

In the simplest case, references that hit a cache do not change its state, but references that miss a cache do change its state because they are followed by a line refill that overwrites one of the cache sets.

Fig. 3 shows how a trap-driven simulator can detect changes in cache state. The figure represents memory as a collection of pages, which in this example are each divided into 16-byte regions on which access levels of *no access* or *full access*[1] can be set. Notice that the current setting of full access on the 16-byte regions starting at 0x2000, 0x1020, 0x8030 and 0xAFF0, correspond exactly to the type of accesses that would result in hits for the cache shown in Fig. 2. References that would miss the simulated cache, however, are marked as not accessible in the memory of the host machine and would cause a trap into the simulator if referenced. Notice that the simulator permits access to at most one line in each memory-equivalence class, in keeping with the constrained way that a direct-mapped cache can hold memory lines.

Fig. 3 shows the state of the simulator data structures and the host memory at one particular point in time during the run of the workload. We can see by inspection that the particular pattern of traps that have been set for this particular cache

1. Full access means the maximum access given to the page of the memory location by the host operating system. For text pages, full access is typically read-only, while for data pages it may be read-write access.

structure at this particular point in time will have the desired effect: the next reference to a memory location that is not in contained in the simulated cache will cause a trap. But what happens after the trap? That is, what actions must be taken by the trap handler to ensure that future references that change the state of the simulated cache will also cause traps? We need a more precise specification of the pattern of access rights on memory that are permitted throughout an entire workload run for a given cache configuration. To this end, we introduce the concept of *access constraints.*

We model the host physical memory system as a set of elements, P, that consists of the byte addresses of all memory locations in the host machine. The size of this set, denoted by |P|, is the total number of physical memory locations that are subject to the simulator's access controls. The subset $C \subset P$ represents the memory locations that may be accessed without causing a trap. We can now express the access constraints for the simulation of a direct-mapped cache as follows:

The *cache-size constraint*:

$$|C| \leq \text{cacheSize} \qquad \text{(Eqn 2)}$$

The *line-size constraint*:

$$\forall(a, b \in P)\{(a_{line} = b_{line}) \Rightarrow (a \in C \Leftrightarrow b \in C)\} \qquad \text{(Eqn 3)}$$

The *direct-mapping constraint*:

$$\forall(a \in P)\{|[a] \cap C| \leq \text{lineSize}\} \qquad \text{(Eqn 4)}$$

The *cache-size constraint* says that at most *cacheSize* memory locations in the domain of the trap-driven simulator can be accessed without causing a trap. The *line-size constraint* says that all the memory locations in the same memory line must all be accessible or not accessible as a group. Finally, the *direct-mapping constraint* says that at most one line from each memory equivalence class is accessible at a time. The set C is exactly the set of memory locations that can be accessed by the workload without causing a change of cache state and a corresponding kernel trap. The operation of the trap handler can now be simply stated as follows: *A trap handler maintains the validity of some set of access constraints during the run of a workload.*

We now give a detailed example of how a trap handler responds to an incoming trap in a way that satisfies the access constraints of Eqn 2 - Eqn 4. Fig. 4 shows the trap that occurs after a reference to location 0xB024. This trap corresponds to the cache miss that would occur in an actual cache like the one shown in Fig. 2. In an actual cache, the required line, starting at 0xB020, would be loaded from memory and inserted in cache set 02, displacing the line starting at 0x1020. The trap handler invoked by `tw_trap()` simulates this change in cache state by rearranging the access rights of the host physical memory in accordance with the access constraints and then records the new line in its cache data structures. These actions are depicted in Fig. 4, which shows the trap handler removing access to region (0x1020 to 0x102F), the displaced cache line, and permitting access to region (0xB020 to 0xB02F), which represents the newly accessed line. The trap handler updates the cache data structure, counts the miss, and when it returns to the running workload, the access state of the host memory will be in conformance with all three access constraints. That is, no more than 4 KB of physical memory addresses can be accessed without a kernel trap (Eqn 2, the cache-size constraint),

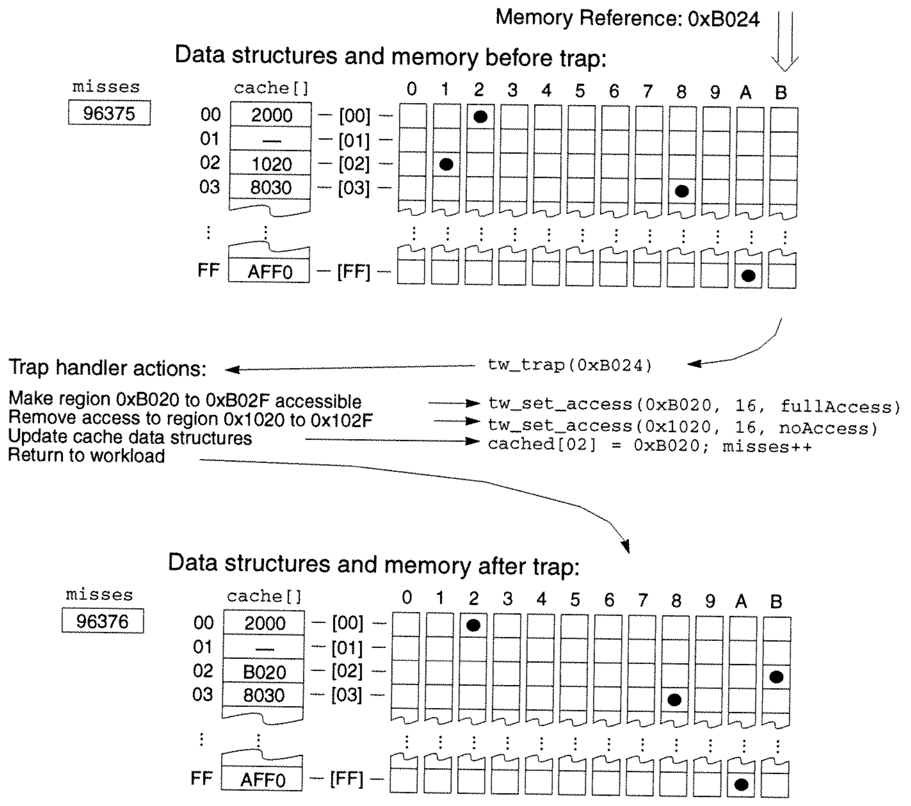

Memory Reference: 0xB024
Data structures and memory before trap:
misses
96375
cache[]
00 2000 [00]
01 — [01]
02 1020 [02]
03 8030 [03]
FF AFF0 [FF]
0 1 2 3 4 5 6 7 8 9 A B
Trap handler actions:
tw_trap(0xB024)
Make region 0xB020 to 0xB02F accessible
Remove access to region 0x1020 to 0x102F
Update cache data structures
Return to workload
tw_set_access(0xB020, 16, fullAccess)
tw_set_access(0x1020, 16, noAccess)
cached[02] = 0xB020; misses++
Data structures and memory after trap:
misses
96376
cache[]
00 2000 [00]
01 — [01]
02 B020 [02]
03 8030 [03]
FF AFF0 [FF]

Fig. 4. An Example Trap in Detail

addresses belonging to the same memory line all have the same access rights (Eqn 3, the line-size constraint), and at most one line from each memory equivalence class can be accessed (Eqn 4, the direct-mapping constraint).

Trap handlers can be written in a flexible way that supports a range of cache configurations. Fig. 5, for example, shows a trap handler that can simulate a variable cache size and a variable line size (specified by the parameters `cacheSets` and `lineSize`). Changing these trap-handler parameters changes, in effect, the access constraints that they enforce, and thus automatically re-partitions memory into a new set of equivalence classes. Note that the size of the simulated cache is not constrained by the cache(s) of the host machine; they may be larger or smaller than the actual caches of the host.

3.3 OS-dependent Tapeworm Primitives

At the beginning of a Tapeworm simulation, the simulated cache is empty, a condition that Tapeworm models by initially setting traps on all workload memory locations. To accomplish this, Tapeworm works with the OS virtual-memory (VM) system. When a task faults on the first access to one of its pages, the VM system registers the page with Tapeworm using `tw_add_page()` (see Table II), which restricts access to each memory location in the page. As the workload begins to access the new page, the first reference to each location causes a trap

```
int cache[cacheSets];
int misses;

tw_trap(pa) {
   tw_set_access(pa, lineSize, fullAccess);
   if (tw_get_access(pa) != noAccess))
       tw_set_access(cache[pa_set], lineSize, noAccess);
   cache[pa_set] = pa_line;
   misses++;
}
```

Fig. 5. A Trap Handler for Direct-mapped Cache Simulation.

into the kernel, which is directed to the Tapeworm trap handlers. A parallel routine, `tw_remove_page()`, is used by the VM system to remove pages from the Tapeworm domain when they are unmapped due to task termination or paging to secondary storage. `tw_remove_page()` clears all traps on a page and flushes the contents of the page from the simulated cache. This mimics the same actions performed by the VM system on the host machine's real cache.

If the VM system maps more than one virtual page to a given physical page, Tapeworm increments a reference count for that physical page, but does not otherwise change access rights to the page. This enables a new task to benefit from shared entries brought into the cache by another task, as would happen in a real system. Similarly, `tw_remove_page()` decrements the reference count, and flushes the page from the simulated cache when the reference count reaches zero.

Tapeworm supports cache simulation for workloads consisting of multiple tasks. To control which tasks are included in a given simulation, each is assigned two Tapeworm attributes (`simulate` and `inherit`), which are set by calling `tw_attributes()`, and are stored in an extended version of the OS task data structure. If `simulate` is zero (the default value), the task runs without any intervention from Tapeworm. When non-zero, `simulate` causes all current and future pages used by the task to be added to the Tapeworm domain via a `tw_add_page()` call. A second attribute, `inherit`, defines the initial value of `simulate` for all children of the task. After a task fork, a child task inherits the Tapeworm attributes of its parent as follows:

```
child.simulate <-- parent.inherit
child.inherit <-- parent.inherit
```

Different settings of the (`simulate`, `inherit`) pair are useful for common simulation situations. For example, if the attribute pair (`simulate=0`, `inherit=1`) is set on a shell task, then any workload that is started from this shell, and all of the workload's children will be registered with Tapeworm. The shell task itself, however, is excluded from the simulation. This inheritance mechanism simplifies the simulation of workloads with complex task fork trees, such as `sdet`, `kenbus` (see Table III), or a multi-stage optimizing compiler. Another common attribute pair, (`simulate=1`, `inherit=0`) is used when only the task itself, but not its children, are to be simulated. This combination is useful for registering kernel pages with Tapeworm.

Table II. OS-dependent Tapeworm Routines

Routine	Description
`tw_add_page(tid, p, v)`	Add a page to the Tapeworm domain. The page is added by restricting access to all of its physical memory locations starting at the page address `p`. The task ID `(tid)` and the virtual-to-physical page mapping defined by `(p,v)` are recorded by Tapeworm to enable forward and reverse address translations.
`tw_remove_page(tid, p, v)`	Remove the page define by `(tid, p, v)` from the Tapeworm domain. The page is removed by flushing it from the simulated cache and by clearing all traps on its memory locations.
`tw_attributes(tid, simulate, inherit)`	Set Tapeworm attributes for the task identified by `tid`. A `tid` of zero signifies the kernel. A non-zero value of `simulate` registers a task with Tapeworm. A non-zero value of `inherit` indicates the initial value of the `simulate` attribute for children of the task.

3.4 Design Summary

Embedding Tapeworm II inside a running system so that it can interact with the host trapping hardware and operating system is an essential characteristic of its design, one that distinguishes it from other trap-driven simulators. By running in kernel mode, the Tapeworm code can control access to all physical memory, and can thus capture the complete activity all user-level tasks in the system, as well as the OS kernel itself. By interacting with the VM code in the host OS, Tapeworm can start the monitoring process just at the moment that a workload begins to execute, avoiding the need to manually pre-process or annotate workload executable files. Finally, because Tapeworm understands task creation, forking, and termination, it can flexibly and dynamically control which tasks in a running system are monitored, and which are ignored.

4. TAPEWORM EVALUATION

We have implemented the Tapeworm II design for TLB and instruction-cache simulation in the Mach 3.0 operating system kernel running on a MIPS R3000-based DECstation 5000/200.[1] In the remaining sections of this paper, we will use this prototype to draw some general conclusions regarding the flexibility, speed, accuracy and portability of trap-driven simulation.

To validate the accuracy of Tapeworm results we use a hardware-monitoring system, called Monster, based on a DAS 9200 logic analyzer [Nagle92]. This system allows us to unobtrusively count total instructions and stall cycles. For comparisons with trace-driven simulation, we use the Cache2000 memory simulator [MIPS88] driven by Pixie-generated traces [Smith91]. Note that Pixie generates only user-level address traces for a single task, which limits to some extent our ability to compare results with Tapeworm.

1. Ports of Tapeworm also exist for the DECstation 3100 and for x86-based PCs.

Table III. Workload Summary. Benchmarks were compiled with the Ultrix MIPS C compiler version 2.1 (level 2 optimization).

Workload	Description
`xlisp`	Lisp interpreter written in C. Configured to solve the 8-queens problem. A SPEC92 benchmark.
`espresso`	Boolean function minimization. A SPEC92 benchmark.
`eqntott`	Translates logical representation of boolean equation to a truth table. A SPEC92 benchmark.
`mpeg_play`	`mpeg_play` V2.0 from the Berkeley Plateau Research Group. Displays 610 frames from a compressed video file [Patel92].
`jpeg_play`	The `xloadimage` program written by Jim Frost. Displays four JPEG images.
`ousterhout`	John Ousterhout's benchmark suite from [Ousterhout89].
`sdet`	A multiprocess, system performance benchmark which includes programs that test CPU performance, OS performance and I/O performance. From the SPEC SDM benchmark suite.
`kenbus`	Simulates user activity in a research-oriented, software development environment. From the SPEC SDM benchmark suite.

Throughout this evaluation, we use the workloads summarized in Table III and Table IV. With the exception of the SPEC92 benchmarks `xlisp`, `espresso` and `eqntott`, the common characteristic of each of these workloads is that they consist of multiple tasks and/or spend a significant fraction of their time executing OS services.

4.1 Flexibility

In the previous section, we showed how to count only misses in direct-mapped cache simulations. To be useful, trap-driven simulation methods must, of course, be able to determine the performance of a much broader range of memory structures (defined by cache size, line size, associativity, replacement policy, indexing policy, etc.) in terms of a variety of other metrics (such as miss ratios, misses per instruction, etc.). Trap-driven simulation methods can, in fact, be used to simulate these other cases, as well as other more general monitoring and simulation optimizations.

In showing how other memory configurations can be simulated, it is helpful to recall the concept of access constraints, introduced in Section 3.2. Simulating other aspects of caching structures — associativity, for example — can be accomplished by defining a new set of access constraints, and then implementing a trap handler that enforces the new constraints. We can replace the direct-mapping constraint of Eqn 4 with a new constraint that enables multiple lines in a given memory-equivalence class to occupy a given cache set:

The *set-associativity constraint*:

$$\forall (a \in P)\{|[a] \cap C| \leq (\text{cacheAssoc} \cdot \text{lineSize})\} \qquad \text{(Eqn 5)}$$

The trap handler that enforces this constraint would set cacheAssoc = 2 to allow 2 memory lines to occupy the same cache set, thus implementing a 2-way set-associative cache.

Adding associativity requires a new policy decision to be made: which line should be replaced when a given cache set becomes full? For some common replacement policies, such as Random or first-in-first-out (FIFO), no changes to

Table IV. Workload Characteristics. The Monster monitoring system was used to obtain instruction counts and the fraction of time spent in different tasks. All experiments were performed on a Mach 3.0 kernel (version mk77) with a user-level BSD UNIX server (version uk38) and the DECstation X display server (version 7, release 5). *Run Time* is the total elapsed time in seconds. *User Task Count* is the total number of tasks created (not including the X or BSD server) during the execution of the workload.

Workload	Instr (10^6)	Run Time (secs)	Kernel	BSD Server	X Server	User Tasks	User Task Count
`xlisp`	1,412	67.52	7.3%	7.1%	0.0%	85.6%	1
`espresso`	534	26.80	2.9%	1.9%	0.0%	95.1%	1
`eqntott`	1,306	60.98	1.5%	1.2%	0.0%	97.2%	1
`mpeg_play`	1,423	95.53	24.1%	27.3%	4.0%	44.6%	1
`jpeg_play`	1,793	89.70	9.1%	9.4%	2.6%	78.8%	1
`ousterhout`	567	37.89	48.0%	31.4%	0.0%	20.6%	15
`sdet`	823	43.70	43.7%	35.5%	0.0%	20.8%	281
`kenbus`	176	23.13	48.9%	29.1%	0.0%	22.0%	238

the access constraints are required. For others, such as least-recently-used (LRU) or not-most-recently-used (NMRU), a somewhat more restrictive set of access constraints are required to obtain information about access order.

We have used the access-constraint method to define and implement Tapeworm trap handlers that support a flexible range of cache sizes, line sizes, associativities, replacement policies, and indexing policies (virtual or physical), as well as multi-level caches. We have also implemented set (congruence-class) sampling as defined by Puzak and Kessler [Puzak85; Kessler91], as well as the single-pass, multi-configuration stack algorithms first proposed by Mattson et al. [Mattson70]. We compute other performance metrics given miss counts from the trap handlers and data from the `tw_get_counts()` call. For example, we compute misses per instruction by dividing miss counts with the value returned from a call to `tw_get_counts(instFetch)`. More detailed descriptions of the access constraints and trap handlers required to implement these and other simulation configurations and performance metrics can be found in [Uhlig95b].

Although we have achieved a high degree of flexibility in trap-driven simulations with our implementation of Tapeworm II, there are some cases that cause difficulties. In particular, Tapeworm has trouble simulating memory structures that require an accurate accounting of time, such as write buffers. Although our prototype of Tapeworm II does not support data-cache simulation, this is not an inherent limitation of trap-driven simulation as work by Reinhardt et al. has shown [Reinhardt93].

4.2 Speed

We compare the speed of simulators using a *slowdown* metric, which we define as the ratio of simulation overhead to the run time of an un-instrumented workload. Depending on the simulator, we compute slowdown as follows:

Cache Size	Miss Ratio
1K	0.118
2K	0.097
4K	0.064
8K	0.023
16K	0.017
32K	0.002
64K	0.002
128K	0.000
256K	0.000
512K	0.000
1024K	0.000

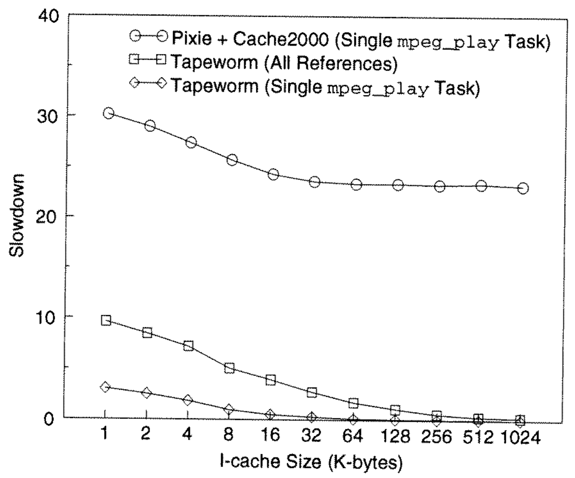


Fig. 6. Comparison of Trace-driven and Tapeworm Slowdowns. Tapeworm slowdowns compared with slowdowns of a Cache2000 simulation driven by Pixie-generated instruction address traces. The simulation is of `mpeg_play` for different sizes of direct-mapped instruction caches with 4-word lines (4 bytes/word). Two different Tapeworm simulations are shown: one with user-only references from just the `mpeg_play` task and another with references from all workload components, including the kernel and user-level servers (BSD and X). The Pixie + Cache2000 combination can measure only a single-task workload. In all cases, slowdowns were computed relative to the total wall-clock run time for all workload components.
The Pixie + Cache2000 simulations were performed under Ultrix 4.1 on a DECstation 5000/133. The Tapeworm simulation were performed under Mach 3.0 on a DECstation 5000/200. Slowdowns in each case were computed relative to the respective host machine to make them comparable.

Slowdown = (Tapeworm Overhead) / (Normal Workload Run Time) (Eqn 6)
Slowdown = (Cache2000 Overhead) / (Normal Workload Run Time) (Eqn 7)

where *Overhead* is the time added to a workload run by Tapeworm or Cache2000. In the case of Cache2000 simulations, this overhead includes the time to generate addresses from a pixie-annotated workload. *Normal Workload Run Time* is for an unmodified workload running on a host machine.

Fig. 6 plots Tapeworm and Cache2000 slowdowns against cache size for the `mpeg_play` workload.[1] Because the Pixie and Cache2000 combination can measure only a single-task workload, Tapeworm was configured to set traps only on memory locations in the `mpeg_play` task to enable a fair comparison.[2] For both simulators, slowdowns decrease as cache sizes increase. Cache2000 slowdowns are approximately 30 for the smallest caches and decrease to just under 25 for the largest caches, while Tapeworm slowdowns start at about 3-4 for small caches and decrease to 0 as cache size is increased. To understand this behavior, consider the following expression for the overhead of the Cache2000 simulations:

1. The other workloads in our suite exhibit similarly-shaped slowdown curves, although their position against the y-axis (i.e., their absolute slowdowns) vary. The `mpeg_play` slowdowns are among the highest in our workload suite.
2. The plot also shows Tapeworm slowdowns when all workload components are monitored. The resulting slowdowns are about 2 to 2.5 times greater.

Table V. Tapeworm Miss Handling Time. This table shows the instructions required to handle different components of a Tapeworm trap for the simulation of direct-mapped caches with 4-word line sizes. A 25-MHz DECstation 5000/200 required 299 cycles to execute the 188 instructions in the handler.

Task	Instructions
Kernel Entry and Exit	59
Obtain Faulting Address	33
Direct-mapped Cache Simulation	45
Set Trap	46
Clear Trap	5
Total	188

$$\text{Cache2000 Overhead} = (\text{Miss}_{\text{count}})\,(\text{Miss}_{\text{time}}) + (\text{Hit}_{\text{count}})\,(\text{Hit}_{\text{time}}) \qquad \text{(Eqn 8)}$$

where $\text{Miss}_{\text{count}}$ and Hit_{time} represents the number of simulated cache misses and hits, while $\text{Miss}_{\text{time}}$ and Hit_{time} are the average amount of time required to process simulated cache hits and misses, respectively. These processing times are different because a simulated cache hit requires only an address generation and search operation (about 60 cycles in Cache2000), but a simulated cache miss also requires data structures to be updated with the missing cache line (about 260 cycles in Cache2000). This explains why the Cache2000 slowdowns decrease with increasing cache size; larger caches exhibit more hits than misses, and hits require less time to process. In contrast, Tapeworm adds overhead only when executing its trap handler:

$$\text{Tapeworm Overhead} = (\text{Trap}_{\text{count}})\,(\text{Trap}_{\text{time}}) \qquad \text{(Eqn 9)}$$

where $\text{Trap}_{\text{count}}$ is the number of Tapeworm traps and $\text{Trap}_{\text{time}}$ is the average time to process a single trap. The Tapeworm trap handler, can displace workload instructions from the host I-cache, thus increasing the number of workload I-cache misses. We include the cost of host I-cache pollution as part of the average time to handle a Tapeworm trap. Pollution of the host D-cache is also included as part of the average trap-handling time, but this effect is minor.

The original implementation of the Tapeworm miss handler was written entirely in C and required over 2,000 cycles per miss to execute, similar to the 2,500 cycles required for the same operation in the Wisconsin Wind Tunnel Simulator [Lebeck94]. This cost was so high in comparison with the trace-driven hit and miss times that Tapeworm slowdowns were comparable to Cache2000 slowdowns when simulating small cache structures that frequently trapped.

To improve performance, the handler was optimized by re-writing it entirely in assembly code and by bypassing the usual kernel entry and exit code. The new code uses no execution stack and saves fewer registers, requiring approximately 300 cycles to handle simulated misses in direct-mapped caches with 4-word line sizes (see Table V for the components of this time).

The expression for Tapeworm overhead explains the shape of the Tapeworm slowdown curves shown in Fig. 6. Small caches frequently miss, resulting in a change of cache state and a Tapeworm trap. The resulting overall slowdowns for a

1-KB cache are about 3 to 4. As the number of misses decreases for larger caches, the number of traps also decrease to negligible amounts, and slowdowns approach zero for caches as small as 8-KB to 16-KB.

Large fractions of the time in the Tapeworm trap handler could be further reduced with the help of better host hardware support. The 59 instructions required by the kernel entry and exit consist mostly of instructions that save and restore registers and that redirect a trap from the general-exception vector to the Tapeworm trap handler. This cost could be reduced if the host hardware supported a dedicated vector for access-fault traps. The 33 instructions required to obtain a faulting address and the 46 instructions required to set an access trap are due mostly to an awkward interface to the ECC diagnostic logic on the DECstation 5000/200, and could be reduced substantially with a cleaner design. An additional benefit of a cleaner design is that it would reduce the number of working registers required by the trap handler, thus further reducing the cost of kernel entry and exit.

We have introduced a simple model to explain Cache2000 and Tapeworm slowdowns. In the following sections, we use this model to explain Tapeworm slowdowns in greater detail over a broader range of simulated cache and TLB configurations. Because the following sections do not include comparisons with Cache2000, Tapeworm slowdowns for the remainder of this chapter include all system activity (the `mpeg_play` task, the Mach 3.0 kernel and the user-level BSD and X servers).

4.2.1 Line Size and Slowdown. The slowdowns show in Fig. 6 are for the simulation of a direct-mapped cache with a 4-word line size. Simulating larger line sizes increases the amount of time in the handler because access must be changed on larger clusters of memory. On the other hand, increasing the line size decreases the number of cache misses because larger lines better exploit the temporal and spatial locality in memory-reference streams. These two opposing effects are shown in the upper-left graph of Fig. 7, which shows that each doubling of the line size reduces the number of cache misses by 30% to 45%, with diminishing reductions in misses as the line size increases. On the other hand, each doubling of line size increases the miss-handling time by 25% to 80%, with larger relative increases in time as line size increases. Changing memory access on a cluster of 4 words requires about 100 cycles. For small line sizes, this is a relatively small component of the miss-handling time, which is dominated by the kernel entry and exit code. However, as line sizes grow large, the fraction of miss-handling time spent changing memory access begins to dominate, and each doubling of the line size nearly doubles the time to handle a miss.

Recall that the overall Tapeworm overhead is the product of the number of traps and the time required to handle each trap. For direct-mapped caches, traps occur if and only if a reference misses the simulated cache. The resulting slowdowns are shown in middle-left graph of Fig. 7, which shows that initially, increasing the line size reduces overall simulation slowdowns because the number of misses is substantially reduced, but the relative increase in miss handling times is relatively small. However, for the largest line sizes, slowdowns begin to increase because

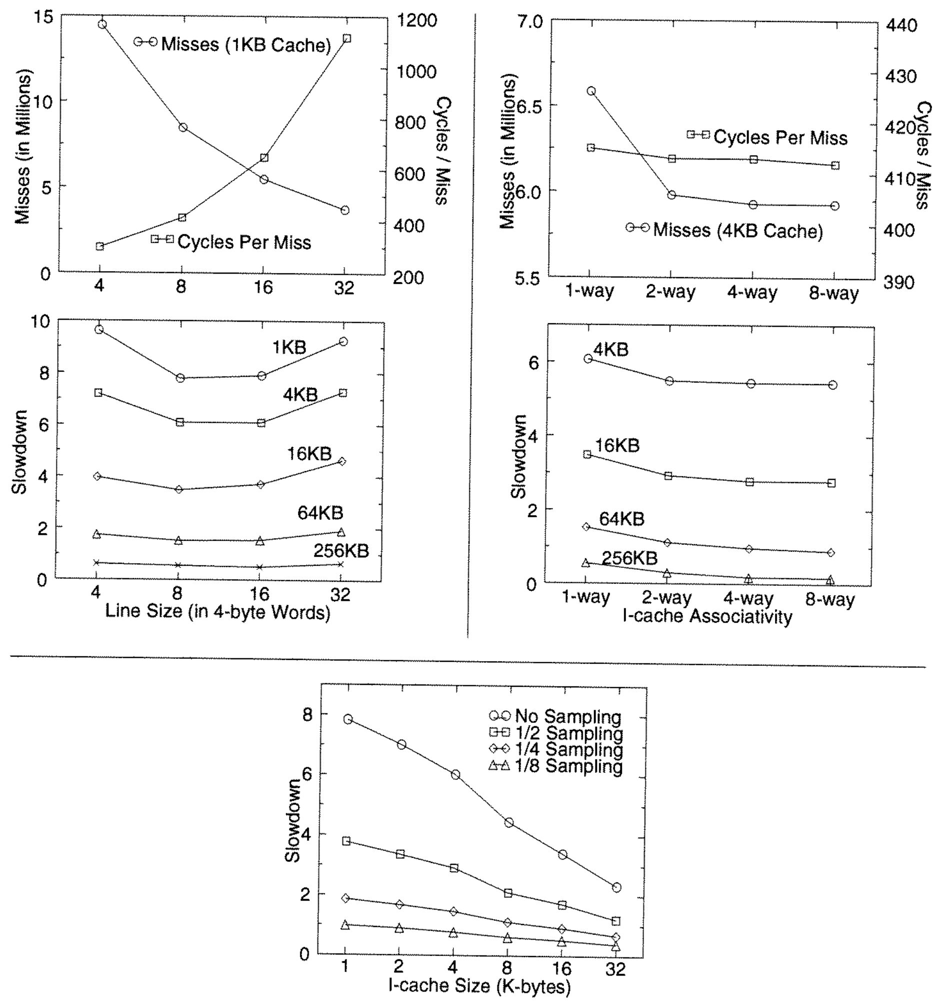


Fig. 7. Tapeworm Slowdowns for Different Simulation Configurations. The figures to the left show Tapeworm slowdowns for direct-mapped caches with varying line sizes, while those to the right show different degrees of associativity (with a line size fixed at 8 words). The top plots superimpose miss counts and cycles per miss on the same graph, while the middle plots show the combined effect of these terms on overall slowdown. The bottom, center plot shows slowdowns for different sampling ratios when simulating small, direct-mapped I-caches with a line size of 8 words

the relative reduction in misses begins to diminish, while the cost of handling a miss increases geometrically. The "U-shape" of these Tapeworm slowdowns versus line size is very similar to those of the performance of actual hardware caches that exhibit cache pollution due to large lines [Przybylski90].

4.2.2 Associativity and Slowdown. For a very simple replacement policy, such as Random, the simulation of cache associativity does not appreciably change trap-handling times in the Tapeworm II prototype. However, caches with higher degrees of associativity typically exhibit fewer cache misses, resulting in overall decreases in simulation slowdowns.These effects are show in the top and middle graphs on the right side of Fig. 7. Trap-handling times and number of traps (misses) for caches ranging in associativity from 1-way (direct-mapped) to 8-way are shown. The product of these two terms show that trap-driven slowdowns decrease with increasing simulated associativity. Because the greatest reductions in miss counts come from 2-way, set-associativity, overall slowdowns do not decrease substantially for associativities of 4-way or greater.

4.2.3 Set Sampling and Slowdown. Many researchers have shown that it is possible to obtain good estimates of overall performance by sampling only a subset of all references made by a workload [Puzak85, Laha88, Kessler91, Wood91]. Trap-driven simulation supports very efficient implementations of sampling because memory locations that are not in a time or set sample never cause any traps. The bottom plot of Fig. 7 illustrates the benefits of set sampling as implemented by Tapeworm. Notice that sampling 1/N-th of the cache sets reduces slowdowns by a full factor of N. This same reduction in simulation time is difficult to achieve with trace-based tools that use code annotation techniques. In an implementation of time sampling in MemSpy, for example, Martonosi shows that a sampling ratio of 10% results in a speedup of only 2 due to the base overhead of static code annotations that slow a running workload both when sampling is enabled and disabled [Martonosi93].

When 1/8-th set sampling is used, Tapeworm overheads for even the smallest 1-KB I-caches result in less than a doubling of workload run times. Larger caches (> 32-KB) add less than 20% to 30% to run times. Slowdowns that are this low make it possible to monitor cache performance while the host workstation is in actual use, opening up new possibilities for real-time memory-system analysis.

Although set sampling improves simulation speeds, it also increases the amount of measurement variance. We examine this effect in greater detail in later sections on simulation accuracy.

4.3 Completeness and Accuracy

Measurements of performance delivered by a memory-system simulator are typically subject to two basic types of error: variance and bias. *Variance* refers to differences in measured performance over multiple trial runs of the same workload on the same memory-system configuration, while *Bias* refers to consistent, systematic over- or under-estimates of true performance during multiple experimental trials. Memory-system simulators are subject to many sources of measurement variation and bias, some of which are due to natural effects occurring in real systems, while others are induced by the method of instrumentation and simulation itself. An ideal memory-system simulator is sensitive to the real, naturally-occurring effects, but avoids the induced, artificial sources of measurement error. In this section, we will show that trap-driven simulation is not inherently any more or less accurate than trace-driven simulation, but it is more sensitive to certain real-system effects that can cause true variability in performance.

Table VI. Variation in Measured Memory System Performance. These measurements include 16 trials apiece, were taken using 1/8 set sampling and consider all activity including the kernel and servers. The simulations are of 16 K-byte, 4-word line, direct-mapped, physically-indexed caches. x is the mean number of misses, and s is the standard deviation of the trial set. Numbers in parenthesis are the percent of the mean value for s and *Range*, and the percent difference from the mean value for *Minimum* and *Maximum*.

Workload	$\bar{x}$ ($\times 10^6$)	s ($\times 10^6$)	Minimum ($\times 10^6$)	Maximum ($\times 10^6$)	Range ($\times 10^6$)
eqntott	4.42	2.53 (57%)	3.25 (26%)	13.13 (197%)	9.88 (223%)
espresso	4.91	2.93 (60%)	3.45 (30%)	13.72 (180%)	10.28 (209%)
jpeg_play	18.58	1.34 (7%)	16.26 (13%)	21.96 (18%)	5.71 (31%)
kenbus	20.89	5.30 (25%)	17.10 (18%)	36.37 (74%)	19.27 (92%)
mpeg_play	58.48	7.01 (12%)	47.34 (19%)	68.95 (18%)	21.61 (37%)
ousterhout	31.50	2.61 (8%)	27.09 (14%)	35.03 (11%)	7.94 (25%)
sdet	41.28	8.77 (21%)	32.58 (21%)	63.48 (54%)	30.90 (75%)
xlisp	41.55	31.78 (76%)	15.16 (64%)	104.48 (151%)	89.32 (215%)

4.3.1 Sources of Measurement Variation. With trace-driven simulations, the same trace from a given workload is typically used repeatedly to obtain performance measurements for different memory configurations. As a result, trace-driven simulations exhibit no variance if the simulation for a given memory configuration is repeated. The precise sequence of traps that drive a Tapeworm simulation, however, are impossible to reproduce from run to run because of dynamic-system effects. For example, the distributions of physical page frames allocated to a task are different from run to run, which changes the sequence of traps to the simulator. This is precisely the same effect that causes performance variations in actual, physically-indexed caches [Kessler92, Sites88]. Measurement variance can also be caused by Tapeworm itself when it employs set sampling; cache-miss estimates vary depending on the number and selection of cache sets that are included in a given sample.

Table VI shows the combined effect of page allocation and set sampling on the measured performance of our workload suite. The table summarizes measurements from 16 trial simulation runs of a 16 K-byte, physically-indexed cache when sampling 1/8th of the cache sets. Standard deviations of the different measurement trials are rather large, ranging from about 10% to as high as 70% of the mean values. In some cases, minimum and maximum values differ from the mean by as much as a factor of two.

To isolate the measurement variation caused by *set sampling*, we removed page-allocation effects by simulating a virtually-indexed, rather than a physically-indexed cache. The memory references applied to a virtually-indexed cache from run to run of the same workload are unaffected by virtual-to-physical page allocation. After removing variation due to page allocation, new trials were performed with and without set sampling. The results are shown in Fig. 8 for `espresso`. Results without sampling show zero variance over multiple trials of the experi-

Size (KB)	Misses (x 10^6)		
	x	s	%
With Sampling			
4	1.72	0.13	8%
8	0.89	0.09	10%
16	0.38	0.02	6%
32	0.13	0.01	11%
64	0.02	0.00	3%
128	0.02	0.00	5%
Without Sampling			
4	1.80	0.00	0%
8	0.93	0.00	0%
16	0.41	0.00	0%
32	0.14	0.00	0%
64	0.03	0.00	0%
128	0.02	0.00	1%

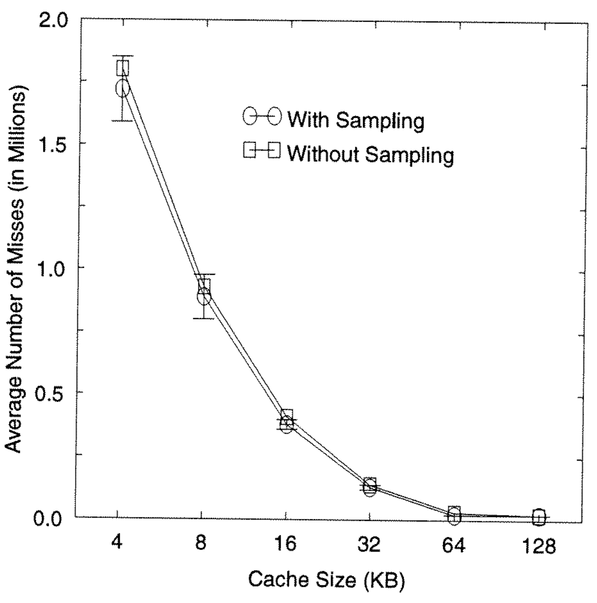


Fig. 8. Variation due to Set Sampling. This table isolates measurement variance due to set sampling Tapeworm removed all other sources of variation by considering only activity from the espresso process (no kernel or servers) and by simulating virtually-indexed caches (4-word line, direct-mapped). The two sets of data points are for measurements with and without sampling and consist ol 16 trials each. The error bars on the plot represent one standard deviation.

ment. Notice that results without sampling consistently predict slightly higher miss counts than those with sampling. This measurement bias, discussed more completely in the next section, is due to an increased time dilation effect from the higher slowdown of the non-sampled experiments.

Fig. 9 shows how *page allocation*, working in isolation, can vary cache performance. We removed sampling variation and then simulated the same workload (mpeg_play in this example) in both a physically-indexed and a virtually-indexed cache. Simulations of the virtually-indexed cache exhibited zero variation because the sequence of references to the cache is independent of the distribution of physical page frames assigned by the OS from run to run. This is essentially the assumption made by most trace-driven cache simulators. Note that the 4 K-byte, physically-indexed cache simulation results do not vary. This is because the page size on this machine is 4 K-bytes; any page allocation will appear the same because all pages overlap in caches that are 4 K-bytes or smaller.

With the physically-indexed cache, the greatest degree of variation (as a percentage of the mean) appears at a cache size of 32 K-bytes, which is roughly the size of the program text used by mpeg_play. This observation is consistent with Kessler's probabilistic model of cache conflicts [Kessler91]. Kessler's model predicts that with random page allocation, the probability of cache conflicts peaks when the size of the cache roughly equals the address space size of the workload,

Size	Misses (x 10^6)		
(KB)	x	s	%
Physically Indexed			
4	37.81	0.09	0%
8	22.38	5.89	26%
16	12.07	4.84	40%
32	9.01	5.62	62%
64	5.83	5.96	10%
128	2.92	4.60	15%
Virtually Indexed			
4	37.75	0.00	0%
8	14.03	0.00	0%
16	10.20	0.00	0%
32	1.90	0.00	0%
64	1.38	0.00	0%
128	0.28	0.00	1%

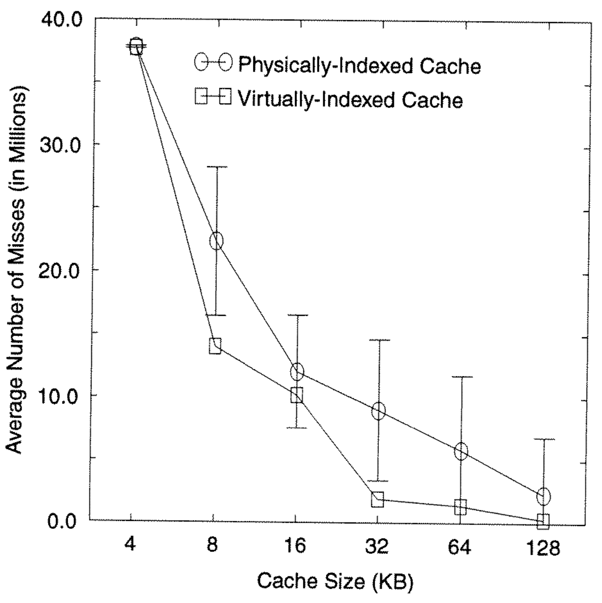


Fig. 9. Variation Due to Page Allocation. This table shows how page allocation alone can vary cache performance. Tapeworm removed all other sources of variation by considering only activity from the `mpeg_play` process (no kernel or servers), and by not sampling. The two sets of data points are for a physically- and virtually-indexed cache (4-word line, direct-mapped). Each data point is the average of 4 trials. The error bars on the plot represent one standard deviation.

and decreases for larger and smaller caches. Fig. 10 illustrates this effect more clearly for other workloads and over a wider range of cache sizes and associativities. The plot shows that increased cache associativity reduces performance variation. This happens for two reasons. First, increased associativity increases the size of cache required for page allocation to have any affect at all.[1] Second, associativity reduces cache conflict misses, which are the type of cache misses that are affected by page-allocation decisions [Kessler91].

Variation due to page allocation is comparable to, if not larger than, that of set sampling. This suggests that the error introduced by sampling is a reasonable trade for increased speed when simulating physically-indexed caches. Of course, the combined effect of both sources of variance is greater than either in isolation, which may require a larger number of trials to be performed to increase the level of confidence in the mean value. Even when few experimental trials are conducted, sampling can be an effective method for quickly approximating the cache requirements of a workload when some experimental error is considered to be acceptable.

1. Increased associativity increases cache size, but does not increase the number of cache sets. Therefore, an 8-KB, 2-way set-associative cache is indexed in the same way as a 4-KB, direct-mapped cache.

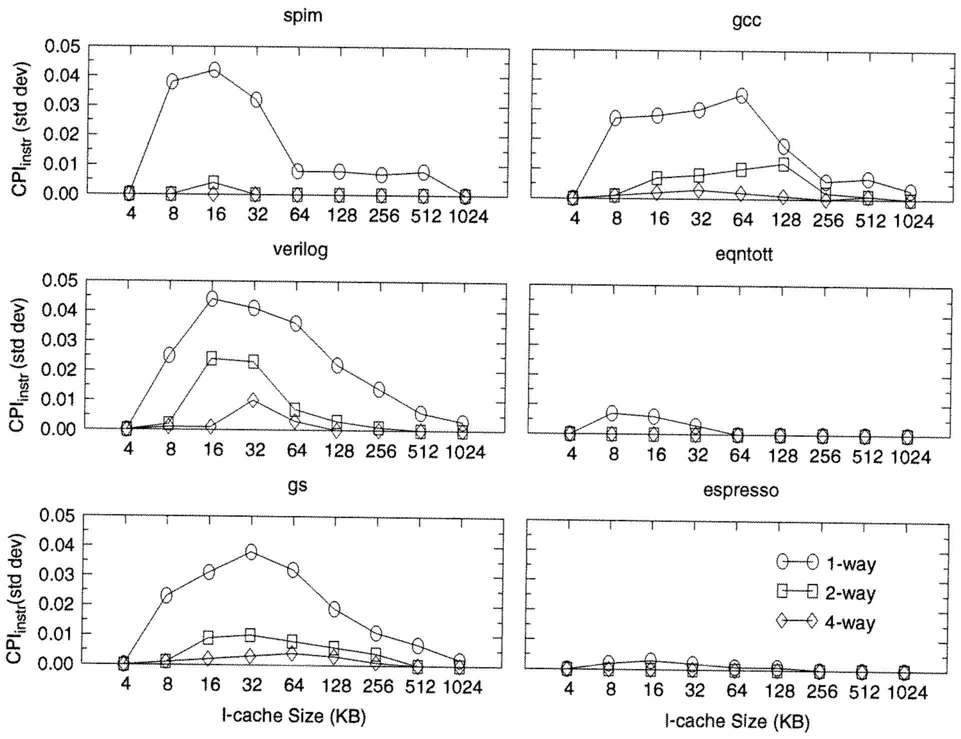


Fig. 10. Variability in I-cache Performance versus Size and Associativity. These plots show variability in performance over multiple runs of the same workload in a physically-indexed I-cache Performance varies because the allocation of virtual pages to physical cache page frames is different from run to run. Variability is reported on the y-axis in terms of one standard deviation of CPI_{instr}, the I-cache contribution to CPI.

In addition to page allocation, we have observed other sources of memory-system performance variation due to OS effects, such as substantial increases in TLB misses due to kernel and server memory fragmentation in a long-running system.

In summary, trap-driven simulation results, as produced by the Tapeworm prototype, are subject to both artificial sources of variation (set sampling), as well as natural sources of variation (page-allocation effects and memory fragmentation). Because Tapeworm is part of an actual, running system, is also sensitive to other system effects, such as link order in a system that implements dynamic linking, or randomness in the order of task scheduling.

Tapeworm's sensitivity to *natural* sources of performance variation, which may necessitate multiple experimental trials, is not a liability. Performance variations due to page allocation and memory fragmentation are real system effects that should be understood and taken into account when making design decisions. If necessary, however, Tapeworm simulations can be configured to remove these effects and produce measurements with less variation, like those from traditional trace-driven simulators. An example of this is shown in Table VII.

Table VII. Measurement Variation Removed. These measurement were made as in Table VI, but with variation due to sampling and page allocation removed. This was accomplished by configuring Tapeworm for simulation of virtually-indexed caches without set sampling.

Workload	x (x 10^6)	s (x 10^6)		Minimum (x 10^6)		Maximum (x 10^6)		Range (x 10^6)	
eqntott	4.19	0.10	(2%)	4.11	(2%)	4.26	(2%)	0.15	(4%)
espresso	4.26	0.06	(1%)	4.21	(1%)	4.30	(1%)	0.09	(2%)
jpeg_play	20.60	0.06	(0%)	20.56	(0%)	20.64	(0%)	0.08	(0%)
kenbus	22.03	0.05	(0%)	21.99	(0%)	22.06	(0%)	0.07	(0%)
mpeg_play	53.16	0.06	(0%)	53.12	(0%)	53.20	(0%)	0.08	(0%)
ousterhout	34.69	1.22	(4%)	33.83	(2%)	35.55	(2%)	1.72	(5%)
sdet	41.23	0.00	(0%)	41.22	(0%)	41.23	(0%)	0.00	(0%)
xlisp	21.67	0.19	(1%)	21.53	(1%)	21.80	(1%)	0.27	(1%)

4.3.2 Sources of Measurement Bias. With sufficient experimental trials, the variance errors of a workload can be quantified and analyzed. In the absence of other sources of error, the resulting mean value will provide a good estimate of true system performance. In this section we examine more serious forms of measurement error that systematically over- or under-estimates true system performance. Sources of measurement bias are hard to correct for because they are more difficult for the simulator to account for and remove. Nevertheless, we will use certain Tapeworm features to isolate and identify the magnitude of sources of measurement bias, whenever possible.

If a simulation method completely omits memory references made by certain portions of a workload, the accuracy of the resulting simulations will clearly be affected. The most common form of omission is to restrict memory references to a single task. This occurs, for example, when the Cache2000 simulator is driven by Pixie-collected traces. We illustrate the importance of including all workload components (user, server and kernel)[1] by using Tapeworm to measure their individual contributions to the total number of I-cache misses.

Table VIII shows I-cache miss counts and miss ratios for each of our workloads in a 4 K-byte cache. The table shows the number of misses from the kernel, the BSD and X servers, and the user tasks when each is allowed to run in a dedicated cache.[2] The *All Activity* column gives results when each of these workload components share a single cache. Due to cache interference among the individual workload components, the sum of the individual miss columns is less than the *All Activity* column.

1. By *user task*, we mean any of several tasks that are children of the shell from which the workload was initiated. We collect tasks together in our simulations with the Tapeworm inheritance attribute. A *server task* is the X display server or the BSD server, which exist prior to the initiation of a workload. We refer to the *server tasks* and the *kernel* as the *system components* of the workload.

2. The cache is shared by multiple user tasks in the case of `kenbus`, `sdet` and `ousterhout`.

Table VIII. Miss Count and Miss Ratio Contributions for Different Workload Components. This table gives the number of misses (in millions) and the miss ratios (in parentheses) for different workload components. The data were collected by running separate trials in which each workload was run in a dedicated direct-mapped cache of 4 K-bytes, with a 4-word line. Whenever possible (e.g., for the single-task workloads), *From Traces* gives the miss ratios predicted by a trace-driven simulation using Pixie+Cache2000. *All Activity* gives total miss counts when all workload components share the same cache. Note that because of cache interference effects, the values in this column are greater than the sum of the individual components. This difference is shown in the last column, entitled *Interference.* All miss ratios are relative to the total number of instructions in the workload, not just the instructions in a given workload component. Hence, the miss ratios from each individual component, plus interference, all sum to the total miss ratio given under*All Activity.*

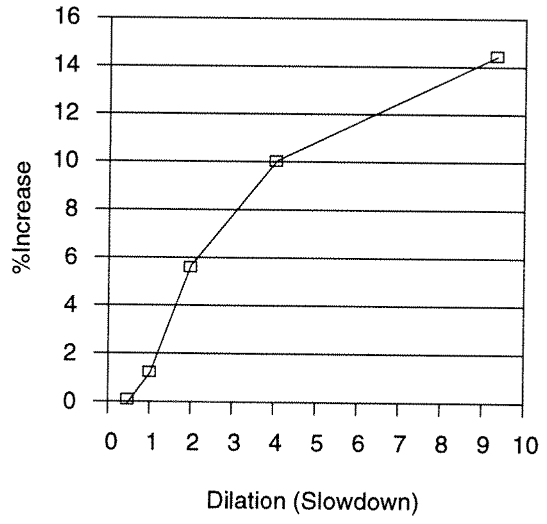
Workload	From Traces	User Tasks	Servers	Kernel	All Activity	Interfer-ence
eqntott	0.1 (.000)	0.1 (.000)	2.5 (.002)	2.4 (.002)	8.4 (.007)	3.4 (.003)
espresso	1.6 (.003)	1.8 (.003)	2.3 (.004)	2.0 (.004)	9.5 (.018)	3.5 (.007)
jpeg_play	3.0 (.002)	3.1 (.002)	14.6 (.008)	9.2 (.005)	36.3 (.020)	9.4 (.005)
kenbus	-----	7.5 (.043)	11.9 (.068)	12.8 (.073)	45.7 (.260)	13.5 (.077)
mpeg_play	37.6 (.027)	37.9 (.027)	33.9 (.024)	19.3 (.014)	112.5 (.079)	21.4 (.015)
ousterhout	-----	1.9 (.003)	18.6 (.033)	21.7 (.038)	61.4 (.108)	19.1 (.034)
sdet	-----	20.1 (.024)	25.2 (.031)	18.1 (.022)	104.6 (.127)	41.3 (.050)
xlisp	85.8 (.061)	90.0 (.064)	6.3 (.004)	3.0 (.002)	135.8 (.096)	36.6 (.026)

Note, first, that the SPEC92 benchmarks `eqntott` and `espresso` exhibit very low miss counts overall. This is consistent with previous observations that many of the SPEC92 benchmarks require only small I-caches to run well [Gee93]. The servers and kernel contribute the majority of total misses, but even with their contribution, the total number of misses is negligible. Other workloads, such as `mpeg_play`, `jpeg_play`, `sdet` and `ousterhout` exhibit the same predominance of server and kernel misses, but with much higher overall miss ratios. In `ousterhout`, for example, the total miss ratio is over 10%, mostly due to the system components and interference effects. A simulator that considers only the user-task component of `ousterhout` would incorrectly estimate the I-cache miss ratio to be less than 1%. The only workload in our suite with a greater fraction of misses coming from a user task is `xlisp` which performs much better in a cache that is only slightly larger.

The amount of memory used by Tapeworm is small in comparison with many trace-driven tools. In particular, Tapeworm does not cause a program to increase in size due to code annotation, nor does it require large regions of host memory to be reserved for trace buffers. As a result, Tapeworm does not suffer from measurement bias due to memory dilation, a problem often encountered by trace-driven simulation tools [Chen94]. Small amounts of host memory are, however, required for the Tapeworm code and data structures. About 256 K-bytes of physical memory are allocated to Tapeworm at boot time. This removes 64 pages from the free memory pool, resulting in a possible increase in paging activity. This effect could be offset by adding a small amount of additional memory to the host machine.

Table IX. Error Due to Time Dilation. Increases in cache misses due to time dilation were measured for the `mpeg_play` workload including all system activity (kernel and servers), running in a physically-addressed 4 K-byte, direct-mapped I-cache with 4-word lines. Time dilation was varied by changing the degree of sampling.

Dilation (slow-down)	Misses (x 10^6)	Increase %
0.43	90.56	0.0%
0.96	91.54	1.2%
2.08	95.70	5.7%
4.42	99.66	10.1%
9.29	103.57	14.4%



Because Tapeworm slows execution of a system, it is subject to the same form of *time dilation* errors present in memory traces. One effect of time dilation is that it causes more clock interrupts to occur during the run of a workload, leading to increased cache conflict misses. Table IX shows the magnitude of error induced by time dilation. Notice that error grows most steeply from slowdowns of 0 to 2, and then levels off for larger slowdowns. Most Tapeworm slowdowns are under 4 where bias tends to be under 10%. Because the amount of slowdown varies from workload to workload, time dilation cannot be removed by a simple adjustment to the clock interrupt frequency as is done in [Borg90, Chen94]. The most effective way to remove measurement bias due to time dilation is to use set sampling to reduce simulation slowdowns. Although multiple experimental trials may be required, the resulting mean value will be free of time dilation bias.

Until now we have described forms of measurement bias shared by both trace-driven and trap-driven simulators. One source of bias that is specific to trap-driven simulation is due to the masking of certain Tapeworm memory traps. In the DECstation 5000/200, single-bit ECC errors raise a hardware interrupt line to cause a kernel trap. If interrupts are disabled, a kernel trap cannot occur, resulting in a reduction of simulated cache misses seen by Tapeworm. Because only the kernel runs with interrupts masked, this limitation affects only kernel references. Unfortunately, we have no way to quantify this effect, but only a very small fraction of kernel code (< 1%) is affected. Special code around these regions helps Tapeworm to account for their cache effects, and better host-hardware support for controlling memory access (see Section 3.1) would avoid this problem altogether.

4.3.3 Accuracy Summary. With respect to artificial sources of measurement variation and bias, trap-driven simulation is subject to many of the same sources of error as trace-driven simulation. In particular, variation due to set sampling, and bias due to time dilation, and memory dilation are forms of error that both methods must contend with. The magnitude of these errors, however, is sometimes less with trap-driven simulation (e.g., with memory dilation), and trap-driven simulators are often able to employ certain techniques to minimize the effect of other sources of error (e.g., using set sampling to reduce slowdowns and hence error due to time dilation).

4.4 Portability

Our implementation of the Tapeworm hardware-dependent primitives exposed one of the main weaknesses of trap-driven simulation: portability. As noted in Section 3.1, we have explored three different methods for implementing the access-control primitives in Tapeworm.

The first method, modifying page-valid bits, worked well. The only difficulty was distinguishing between invalid-page traps caused by Tapeworm, and true page faults (due to a page not being memory resident). This problem was solved by adding an extra bit to page-table entries to indicate the true resident status of each page.

Implementing access control with ECC bits in Tapeworm II was far more difficult. First, this trap was routed to a generic exception vector and had to be identified from among many other sources of traps, interrupts and exceptions. A clumsy interface to the memory-control ASIC required a dozen load, shift, add and mask instructions to piece together the memory address of an ECC error (i.e., the `pa` value of the `tw_trap()` call). Once this value was obtained, the corresponding virtual address (i.e., the `va` value) had to be found by searching an inverted page table. These code sequences, along with the complex sequence of interactions with a memory-controller ASIC for re-coding ECC bits and flushing cache entries,[1] required several working registers. This, in turn, required the saving and restoring additional workload registers before and after each trap, further increasing the trap-handling time.

A more serious problem was caused by writes to memory locations marked non-accessible by re-coded ECC bits. These writes caused new (valid) ECC bits to be recomputed and stored to memory without checking the old (invalid) ECC bit values. This behavior effectively changed a memory location's access state from no-access to full-access without notification to Tapeworm. Fortunately, the ECC method could still be used on read-only text pages, but this limited simulations to I-caches with two access states, no-access and read-only. As noted by Reinhardt, it is possible to avoid this problem on a machine with an *allocate-on-write* policy[2] by flushing memory locations from the cache when setting their state to no-access [Reinhardt93]. In such a system, a write to the un-cached location causes the data to first be read (allocated) from main memory into the cache before the write completes. The ECC bits of this allocate operation will be checked in the same way as any other read to main memory, thus forcing a trap to occur. Although this solution enables D-cache simulation, it still only supports two access states: no-access and full-access.

1. On the DECstation 5000/200, ECC is only checked on cache-line refills after a cache miss.
2. The DECstation 5000/200 uses a write-though policy with no allocate-on-write.

Other problems with ECC caused difficulties when porting Tapeworm II to other machines. For example, our port of Tapeworm from a DECstation 5000/200 to a DECstation 5000/240 was hindered due to differences between the way that DMA is implemented on the two machines. Another minor limitation is that ECC bits are checked on 4-word cache-line refills, effectively limiting the simulation of cache line sizes to multiples of 4 words.

We have recently implemented a third method for controlling fine-grained memory access: dynamically swapping breakpoint instructions in place of original instructions. Although its applicability is limited to text pages only, this method substantially simplifies trap handling because instruction breakpoints are far easier to set and clear, and the breakpoint traps report the faulting instruction address in an easily accessible hardware register. Our implementation of this method requires about one third as many machine cycles as the ECC method, but adds more memory overhead to store breakpointed instructions.[1] Because our implementation of the breakpoint method is on a different architecture (an Alpha-based workstation) as part of a different trap-based monitoring tool, it is difficult to directly compare these results with those obtained by the ECC method. We therefore report results from only the ECC method in this paper, although they do not represent the fastest trap-driven simulations that we have measured.

Despite these various implementation problems, we were able to implement enough of the Tapeworm primitive operations to construct a usable trap-driven simulator prototype. Although limited to TLBs and I-caches, this prototype enabled us to evaluate the feasibility of trap-driven simulation without resorting to hardware modifications. We believe that the speed and accuracy obtained using this prototype are promising enough to justify special hardware support for trap-driven simulation.

The most useful form of support would be better fine-grained access control, such as that provided by the Tera [Alverson90]. Such support would be useful for other applications as well, such as debugging and distributed-shared memory [Appel91]. Recent work by Reinhardt et al. shows that fine-grained access control hardware can be implemented, at relatively low cost in design time, as a plug-in board that monitors bus transactions to the host processor [Reinhard96]. More streamlined hardware mechanisms for handling traps would provide another boost to trap-driven simulation performance. Such support could include trap vector addresses dedicated to memory-access traps, shadow or scratch registers for the trap-handling code, and easier access to data such as the virtual and physical addresses causing an access trap. Newer implementations of several microprocessor families provide better support of this type [Huck93; Sun94; Digital92]. Finally, on-chip performance counters, also provided by many newer microprocessors, would help to implement the `tw_get_counts()` routine in a more convenient way.

1. The ECC implementation does not require any extra storage because although our method for causing a trap corrupts data (i.e., flipping a single bit), the original value can be recovered using the SECDED (single-error correcting, double error detecting) code employed by the memory controller.

5. SUMMARY AND FUTURE WORK

Using Tapeworm II as a prototype, we have shown that cache and TLB simulations driven by kernel traps can greatly simplify the problem of evaluating cache and TLB performance under workloads including multiple tasks and operating system loads; Tapeworm requires no pre-processing of a workload to be measured, and adds little memory overhead. Moreover, our measurements show that trap-driven simulations can be performed with a relatively small system slowdowns, compared with trace-driven simulation. Tapeworm slowdowns start at 10 in the worst case (with a 1-KB cache), and approach 0 for larger or more associative caches. Tapeworm can efficiently employ sampling techniques to further reduce slowdowns in direct proportion to the sampling ratio, but at the expense of higher measurement variance.

Unlike trace-driven simulators, which always obtain the same simulation result with a given trace, trap-driven simulators are sensitive to dynamic-system effects, such as page allocation and memory fragmentation, which cause variations in performance from run to run. This is a positive feature of trap-driven simulation because it provides better insight into the true behavior of real machines. One of the most useful features of the Tapeworm II prototype is its ability to monitor all system activity (with the exception of small regions of un-interruptible kernel code). As a result, it is not subject to bias due to omission of workload components. Our measurements showed that this form of error was among the most significant.

The main weaknesses of trap-driven simulation are its portability and flexibility. It remains an open question whether trap-driven simulation will be able to make continued advances in these regards. The outcome will depend, in large part, on the willingness of computer architects to make minor modifications in future designs to better support trap-driven simulation. Even with such support, trap-driven simulation is not suited to certain forms of architectural simulation, such as instruction-pipeline simulation, or simulations that require detailed, cycle-by-cycle accounting of time.

Many other applications would also benefit from fine-grained access control. Program debugging, garbage collection, persistent storage, and distributed shared memory could all be made faster and more efficient [Appel91; Reinhardt94; Schoinas94]. These applications, and the promise of very fast trap-driven memory simulation, suggest that architects should give more serious consideration to supporting fine-grained access control and fast trapping support in future processors and computer systems.

6. ACKNOWLEDGEMENTS

We thank Joel Emer and Bill Grundmann for essential information on the DECstation 5000/200 and its memory-controller ASIC. Many thanks also to Alessandro Forin for his help with Mach 3.0 and its trap handlers. Chih-Chieh Lee implemented the 486 Tapeworm port.

7. REFERENCES

[Agarwal86] Agarwal, A., Sites, R. L. and Horowitz, M. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th International Symposium on Computer Architecture*, Tokyo, Japan, IEEE, 119-127, 1986.

[Agarwal88] Agarwal, A., Hennessy, J. and Horowitz, M. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems* **6** (4): 393-431, 1988.

[Alexander85] Alexander, C. A., Keshlear, W. M. and Briggs, F. Translation buffer performance in a UNIX environment. *Computer Architecture News* **13** (5): 2-14, 1985.

[Alverson90] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. and Smith, B. The tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, 1-6, 1990.

[Anderson91] Anderson, T. E., Levy, H. M., Bershad, B. N. and Lazowska, E. D. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, ACM, 108-119, 1991.

[Appel91] Appel, A. and Li, K. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, ACM, 96-107, 1991.

[Borg90] Borg, A., Kessler, R. and Wall, D. Generation and analysis of very long address traces. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, IEEE, 1990.

[Chen93] Chen, B. and Bershad, B. The impact of operating system structure on memory system performance. In *Proceedings of the 14th Symposium on Operating System Principles*, 1993.

[Chen94] Chen, B., Wall, D. and Borg, A. Software methods for system address tracing: implementation and validation. *Technical Report, Carnegie-Mellon University, DEC Western Research Lab, DEC Network Systems Laboratory.* 1994.

[Clark83] Clark, D. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems* **1**: 24-37, 1983.

[Cmelik94] Cmelik, B. and Keppel, D. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, ACM, 128-137, 1994.

[Cvetanovic94] Cvetanovic, Z. and Bhandarkar, D. Characterization of Alpha AXP performance using TP and SPEC Workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Ill., IEEE, 1994.

[Digital92] Digital. *Alpha Architecture Handbook.* USA, Digital Equipment Corporation, 1992.

[Eggers90] Eggers, S., Keppel, D., Koldinger, E. and Levy, H. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, 37-47, 1990.

[Flanagan92] Flanagan, K., Grimsrud, K., Archibald, J. and Nelson, B. BACH: BYU address collection hardware. *Brigham Young University Technical Report TR-A150-92.1.* 1992.

[Gee93] Gee, J., Hill, M., Pnevmatikatos, D. and Smith, A. J. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro* (August): 17-27, 1993.

[Hill87] Hill, M. *Aspects of cache memory and instruction buffer performance.* Ph.D. dissertation, The University of California at Berkeley. 1987.

[Holliday91] Holliday, M. Techniques for cache and memory simulation using address reference traces. *International journal in computer simulation* **1**: 129-151, 1991.

[Huck93] Huck, J. and Hays, J. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, IEEE, 39-50, 1993.

[Kessler91] Kessler, R. *Analysis of multi-megabyte secondary CPU cache memories.* Ph.D. dissertation, University of Wisconsin-Madison. 1991.

[Kessler92] Kessler, R. and Hill, M. Page placement algorithms for large real-indexed caches. *ACM Transaction on Computer Systems* **10** (4): 338-359, 1992.

[Laha88] Laha, S., Patel, J. and Iyer, R. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers* **37** (11): 1325-1336, 1988.

[Lebeck94] Lebeck, A. and Wood, D. Fast-Cache: A new abstraction for memory-system simulation. *University of Wisconsin - Madison Technical Report 1211*, 1994.

[Lebeck95] Lebeck, A. and Wood, D. Active Memory: A new abstraction for memory-system simulation. In *Proceedings of the 1995 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, May, 1995.

[Martonosi92] Martonosi, M., Gupta, A. and Anderson, T. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, ACM, 1992.

[Martonosi93] Martonosi, M., Gupta, A. and Anderson, T. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the 1993 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Santa Clara, California, ACM, 248-259, 1993.

[Mattson70] Mattson, R. L., Gecsei, J., Slutz, D. R. and Traiger, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal* **9** (2): 78-117, 1970.

[MIPS88] MIPS. *RISCompiler Languages Programmer's Guide.* MIPS, 1988.

[Mogul91] Mogul, J. C. and Borg, A. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, ACM, 75-84, 1991.

[Nagle92] Nagle, D., Uhlig, R. and Mudge, T. Monster: A tool for analyzing the interaction between operating systems and computer architectures. *University of Michigan Technical Report CSE-TR-147-92*. 1992.

[Nagle93] Nagle, D., Uhlig, R., Stanley, T., Sechrest, S., Mudge, T. and Brown, R. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, IEEE, 27-38, 1993.

[Nagle94] Nagle, D., Uhlig, R., Mudge, T. and Sechrest, S. Optimal Allocation of On-chip Memory for Multiple-API Operating Systems. In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, 1994.

[Ousterhout89] Ousterhout, J. Why aren't operating systems getting faster as fast as hardware. *WRL Technical Note* (TN-11): 1989.

[Patel92] Patel, K., Smith, B. C. and Rowe, L. A. Performance of a Software MPEG Video Decoder. *Technical Report, University of California, Berkeley.* 1992.

[Przybylski90] Przybylski, S. The performance impact of block sizes and fetching strategies. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Seattle, WA, IEEE, 160-169, 1990.

[Puzak85] Puzak, T. *Analysis of cache replacement algorithms.* Ph.D. dissertation, University of Massachusetts. 1985.

[Reinhardt93] Reinhardt, S., Hill, M., Larus, J., Lebeck, A., Lewis, J. and Wood, D. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, ACM, 48-60, 1993.

[Reinhardt94] Reinhardt, S., Larus, J., and Wood, D. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April, 1994.

[Reinhardt96] Reinhardt, S., Pfile, R., and Wood, D. Decoupled hardware support for distributed shared memory. To appear in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[Rosenblum95] Rosenblum, M., Herrod, S., Witchel, E., and Gupta, A. Complete computer simulation: the SimOS approach, In *IEEE Parallel and Distributed Technology*, Fall 1995.

[Schoinas94] Schoinas, I., Falsafi, B., Lebeck, A., Reinhardt, S., Larus, J., Wood, D. *Fine-grain access control for distributed shared memory.* In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, ACM Press, 1994.

[Sites88] Sites, R. L. and Agarwal, A. Multiprocessor cache analysis with ATUM. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, IEEE, 186-195, 1988.

[Sites96] Sites, R., Perl, S., PatchWrx - A Dynamic Execution Tracing Tool. *Digital Equipment Corp. Systems Research Center Technical Report.* http://www.research.digital.com/SRC/personal/Dick_Sites/patchwrx/PatchWrx.html

[Smith82] Smith, A. J. Cache memories. *Computing Surveys* **14** (3): 473-530, 1982.

[Smith91] Smith, M. D. Tracing with pixie. *Technical Report, Stanford University*, Stanford, CA. 1991.

[SPEC91] SPEC. The SPEC Benchmark Suite. *SPEC Newsletter.* **3:** 3-4, 1991.

[SPEC93] SPEC. SPEC: A five year retrospective. *The SPEC Newsletter* **5** (4): 1-4, 1993.

[Srivastava94] Srivastava, A. and Eustace, A. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation,* 196-205, June 1994.

[Stunkel91] Stunkel, C., Janssens, B. and Fuchs, W. K. Collecting address traces from parallel computers. In *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, Hawaii, 373-383, 1991.

[Sugumar93] Sugumar, R. *Multi-configuration simulation algorithms for the evaluation of computer designs.* Ph.D. dissertation, University of Michigan. 1993.

[Sun94] Sun Microsystems, Nested traps in UltraSPARC, *http://www.sun.com/stb/Processors/UltraSPARC/WhitePapers/NestedTraps/NestedTraps.html.* September, 1994.

[Talluri94] Talluri, M. and Hill, M. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, ACM, 1994.

[Thompson89] Thompson, J. and Smith, A. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Transactions on Computer Systems* **7** (1): 78-116, 1989.

[Torrellas92] Torrellas, J., Gupta, A. and Hennessy, J. Characterizing the caching and synchronization performance of multiprocessor operating system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, ADM, 162-174, 1992.

[Uhlig95] Uhlig, R., Nagle, D., Mudge, T. Sechrest, S., and Emer, J. Instruction Fetching: Coping with Code Bloat. To Appear In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June, 1995.

[Uhlig95b] Uhlig, R. *Trap-driven Memory Simulation.* Ph.D. Disseratation, University of Michigan, August 1995.

[Uhlig96] Uhlig, R. and Mudge, T. *Trace-driven Memory Simulation: A Survey.* Submitted for publication to ACM Computing Surveys, Fall 1996.

[Witchel96] Witchel, E. and Rosenblum, M. Embra: fast and flexible machine simulation, In *Proceedings of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* Philadelphia, May, 1996.

[Wood91] Wood, D., Hill, M. and Kessler, R. A model for estimating trace-sampled miss ratios. In *Proceedings of the 1991 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 79-89, 1991.