6th International Conference on Architectural Support for Programming Languages and Operating Systems. October, 1994. pp. 132-144.

Trap-driven Simulation with Tapeworm II

Richard Uhlig, David Nagle, Trevor Mudge & Stuart Sechrest

Department of Electrical Engineering and Computer Science
University of Michigan
e-mail: uhlig@eecs.umich.edu, bassoon@eecs.umich.edu

***Abstract**: Tapeworm II is a software-based simulation tool that evaluates the cache and TLB performance of multiple-task and operating system intensive workloads. Tapeworm resides in an OS kernel and causes a host machine's hardware to drive simulations with kernel traps instead of with address traces, as is conventionally done. This allows Tapeworm to quickly and accurately capture complete memory referencing behavior with a limited degradation in overall system performance. This paper compares trap-driven simulation, as implemented in Tapeworm, with the more common technique of trace-driven memory simulation with respect to speed, accuracy, portability and flexibility.*

***Results**: For reasonable miss ratios, Tapeworm simulations are significantly faster than traditional trace-driven simulations. Tapeworm typically slows a system down by less than an order of magnitude (10x) when cache miss ratios are under 10%, and slowdowns approach zero as miss ratios decrease. Tapeworm can employ set sampling techniques to further reduce slowdowns, but at the expense of higher measurement variance. Unlike trace-driven simulations, which typically produce identical results from run to run, trap-driven simulations exhibit greater sensitivity to inherent variations in memory system behavior on a real machine. Less than 5% of Tapeworm's code is machine-dependent, enhancing its portability to different machines provided that they support a few essential primitive operations. Although the trap-driven approach is flexible enough to simulate most TLB and cache configurations, other architectural structures, such as write buffers or instruction pipelines cannot be simulated with this approach. Tapeworm implementations currently exist for TLB and instruction cache simulation on MIPS-based DECstations and for TLB simulation on a 486-based Gateway PC.*

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

1 Introduction

Trace-driven simulation is probably the most popular method for evaluating memory system architectures consisting of caches and TLBs [Smith82, Holliday91]. This technique has worked well in the design of memory systems supporting single-task, user-intensive applications such as those found in the SPEC benchmark suite [Gee93, SPEC91]. However, there is a growing body of work showing that memory systems tuned to this type of workload do not perform as well on interactive and digital-media applications, or with distributed file systems and databases, all of which require frequent interaction with the operating system or other tasks [Agarwal88, Anderson91, Chen93a, Cvetanovic94, Mogul91, Nagle93, Nagle94, Uhlig94b, Ousterhout89]. Unfortunately, most trace-driven simulation tools are limited to single user-mode tasks and thus cannot capture a significant portion of the memory system activity of these applications. Those trace-driven simulators that are OS-capable tend to rely on expensive hardware monitoring equipment and are generally not very portable.

We have developed a software-based tool, called *Tapeworm II*, that attempts to overcome some of these limitations. Tapeworm simulations are driven not by traces, but by traps into the operating system kernel where Tapeworm resides. Each kernel trap corresponds to a simulated TLB or cache miss. This approach has three principal advantages: (1) Completeness, (2) Speed and (3) Portability. Tapeworm simulations are complete because traps can originate from any user task, or even the OS kernel itself. Tapeworm is fast because the simulator is invoked only in the uncommon case of TLB or cache misses. Finally, because Tapeworm is software-based, it can be ported to any system that provides support for certain key primitives.

Despite these advantages, this method does suffer from certain drawbacks. Although capable of simulating TLBs and caches with different sizes and associativities, 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's presence in a system can also introduces new forms of measurement bias. Though not strictly a disadvantage, trap-driven simulations are also more sensitive to inherent variations in memory system performance, an issue which is generally ignored in trace-driven simulation studies.

This paper presents a detailed description of the Tapeworm design in Section 3 and then uses this prototype to compare the strengths and weaknesses of trap-driven simulation against trace-driven simulation in Section 4. We begin with a discussion of related work in the next section.

2 Related Work

To capture multi-task and OS activity, memory architecture studies traditionally have relied on hardware instrumentation tech-

To appear in the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), October 5-7, 1994, San Jose, California

This work was supported by Defense Advanced Research Projects Agency under DARPA/ARO Contract Number DAAL03-90-C-0028, by a National Science Foundation CISE Research Instrumentation Grant No. CDA-9121887, by a Digital Equipment Corporation Grant, and by a National Science Foundation Graduate Fellowship.

Trace-driven

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

Trap-driven

```
kernel traps invoke tw_miss(address):

tw_miss(address){
    miss++;
    tw_clear_trap(address);
    displaced_address = tw_replace(address);
    tw_set_trap(displaced_address);
}
```

Figure 1: Trace-driven versus Trap-driven Simulation Algorithms

The core execution loops of trace-driven and trap-driven simulators. This code abstracts away 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). Single-pass simulators, using stack algorithms, also have a more complex structure [Mattson70, Sugumar93, Thompson89].

niques that either attach extra hardware to a running system, modify the existing hardware (or microcode) or are designed into the architecture [Agarwal86, Alexander85, Clark83, Cvetanovic94, Flanagan92, Nagle92, Torrellas92]. Unfortunately, these hardware approaches are costly to implement and are usually tied to a single machine.

To overcome these limitations, recent research has extended software-only instrumentation techniques to include multi-process and OS activity. Mogul and Borg describe a system where each task in a multi-task workload is instrumented to make entries in a system-wide trace buffer [Mogul91]. A modified operating system kernel interleaves the execution of the different user-level workload tasks according to usual scheduling policies and invokes a memory simulator whenever the trace buffer becomes full. Chen has further extended this technique to include annotation of the OS kernel itself, thus enabling complete accounting of all system activity [Chen93b].

A few simulators avoid traces entirely and are driven, instead, by kernel traps. These tools use privileged machine operations to cause the underlying host hardware to filter hits in a simulated memory structure, and only trap to the simulator on a miss. One example of this approach is the first generation of Tapeworm which performs TLB simulation [Nagle93, Uhlig94a]. This system intercepts kernel traps to the software-managed TLB miss handlers of an R2000-based DECstation to drive a TLB simulator. Because all user and kernel misses are intercepted, Tapeworm is able to fully consider multi-task and OS effects for different TLB configurations. Talluri describes a similar trap-driven TLB simulator that runs on SPARC-based workstations [Talluri94]. Another example, the Wisconsin Wind Tunnel (WWT) simulator, is also based on kernel traps that are set and cleared 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 kernel references, WWT is designed to investigate multi-processor cache coherence algorithms but does not capture OS activity.

Other work shares some of the properties of both trace-driven and trap-driven simulation [Cmelik94, Lebeck94, Martonosi92]. These hybrid approaches annotate a program to invoke simulation handlers on every memory reference. In these systems, simulations can be optimized by calling a null handler on memory locations known to be in 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 cache and TLB simulation and captures multi-task and OS kernel activity. Second, using Tapeworm as a prototype, we investigate the positive and negative aspects of trap-driven simulation in general.

This paper is primarily a study of the pros and cons of trap-driven simulation. For examples of actual studies of operating system and architecture interactions that have used Tapeworm, see [Nagle93, Nagle94, Uhlig94b].

3 Tapeworm Design

3.1 The Tapeworm Algorithm

The Tapeworm simulation algorithm is best explained by contrasting its essential features against those of a traditional trace-driven simulator. We shall use the term *cache* in the following discussion, although these methods apply equally well to TLB simulation.

At its core, trace-driven simulation executes a loop similar to that shown on the left side of Figure 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 generated "on the fly" by an annotated workload [Agarwal86, Borg90, Chen93b, Cmelik94, Eggers90, Holliday91, Hsu89, Larus90, Larus93, Magnusson93, MIPS88, Mogul91, Sites88, Smith91]. 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.

Tapeworm operates on a different principle. It is driven not by address traces, but by traps into the operating system kernel where Tapeworm resides. Tapeworm begins a simulation by setting traps on all memory locations in a workload's address space. Locations with traps set represent memory locations not currently resident in the simulated cache structure. As the workload executes, the first reference to each location causes a trap into the kernel, which is directed to the Tapeworm simulator. Because all such traps represent simulated cache misses, there is no need to search a data structure representing the simulated cache. Tapeworm simply counts the miss and then clears the trap on the required memory location. Clearing the trap effectively caches the memory location

Routine	Description
`tw_set_trap(pa, size)`	Set a memory trap starting at `pa` (physical address) and extending for `size` bytes. Subsequent use of memory locations in this region should trap to the kernel and pass control to Tapeworm.
`tw_clear_trap(pa, size)`	Clear all previously set memory traps starting at `pa` and extending for `size` bytes. Subsequent use of memory in this region by a workload may proceed uninterrupted.
`tw_register_page(tid, p, v)`	Register a page with Tapeworm. The page is added by setting traps on 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 virtually-indexed or physically-indexed cache simulations.
`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 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.
`tw_replace(tid, pa, va)`	Insert a missing memory location, defined by a `pa` (for a physically-indexed cache) or `va` (for a virtually-indexed cache) into a data structure for a simulated cache. If needed, the `tid` is used to form part of the cache (or TLB) tag. A displaced entry, selected on the basis of various simulation parameters such as cache size, line size or associativity, is returned by the call.

Table 1: Tapeworm Primitives in Detail

in the simulated cache structure, because subsequent references to this location do not trap and will proceed at full hardware speed. Tapeworm then sets a new trap on a different memory location, in accordance with some replacement policy, to emulate displacement in the simulated cache structure.

The Tapeworm design offers a number of advantages over trace-driven simulation. First, no annotation or rewriting of the workload is required because traps are dynamically set and cleared while the workload runs. Second, Tapeworm's kernel privileges allow it to easily set traps on any user-level task, as well as the kernel itself. This enables the study of multi-task and OS interactions. Third, Tapeworm uses the underlying hardware to filter out hits in the simulated cache structure. This can provide speed advantages over conventional trace-driven simulation (see Section 4.1).

3.2 Tapeworm Primitives

Although we have outlined the core Tapeworm algorithm, a number of important details, such as mechanisms for setting memory traps and policies for registering workload pages with Tapeworm, need to be clarified. Table 1 describes the key Tapeworm primitives.

The Tapeworm operations of setting and clearing traps are performed by `tw_set_trap()` and `tw_clear_trap()` which can be implemented on many machines by using various privileged operations intended for diagnostics or debugging (see Table 2). For example, a trap can be set by using a diagnostic mode to alter the parity bits for a given memory location [Reinhardt93]. Subsequent use of that memory location will result in a memory parity error trap. The trap can be cleared by restoring correct parity to the memory location. Other mechanisms, such as instruction and data breakpoints or page valid bits, work equally well if they are supported by the host hardware. `tw_set_trap()` and `tw_clear_trap()` accept a size parameter to support a range of simulation parameters, such as various page sizes for TLB simulation and various line sizes for cache

Privileged Operation	Description
Memory Parity or ECC Traps	Trap to the OS kernel after detecting a memory-parity error. Read and write operations enable software to change the parity bits associated with each memory location.
Instruction Breakpoint	Trap to the OS kernel if a breakpoint instruction is encountered.
Data Breakpoint	Trap to the OS kernel if a specific data memory location is read or written.
Variable Page Size	Support for different sized pages. Typical page sizes range from 128 byte to 1-Mbyte.
Instruction Counters	Count the total number of instructions executed by the processor.

Table 2: Privileged Operations Useful for Trap-driven Simulation

This table summarizes some of the privileged operations that are useful building blocks in a trap-driven memory simulator. These are common operations on many existing architectures (see Table 12).

simulation. The actual trapping mechanism selected for the underlying implementation of these routines depends on the size of trap required. For TLB simulation, where the granularity is large, page valid bits are most effective, particularly if the machine supports variable page sizes. Memory parity traps or breakpoints (perhaps set in clusters of more than one) are the best choice for cache simulation, where the required granularity is on the order of a cache line.

Using memory parity or ECC check bits to cause kernel traps can interfere with their intended purpose of detecting true memory errors. In practice, this is only a problem if memory errors are frequent and if it is not possible to distinguish true ECC errors from those caused by Tapeworm. In our experience, neither condition is true. While Tapeworm has been inactive on the system in which it runs, we have only logged one true single-bit ECC error during nearly a year of operation. Even when Tapeworm is active, it correctly detects true memory errors with high probability.[1]

As noted previously, workloads require no previous modification for Tapeworm simulation because traps are set and cleared dynamically as tasks run. For this to work, Tapeworm requires assistance from the OS virtual memory (VM) system. When a task faults on the first access to a page, the VM system registers the page with Tapeworm using a `tw_register_page()` call. After the page is marked valid by the VM system, `tw_register_page()` sets traps on all memory locations in the page so that any future references to parts of the page will invoke the Tapeworm cache miss handler.[2] A parallel routine, called `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 the page and flushes it from the simulated cache. This mimics the same action 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, it must still register the mapping with Tapeworm by using `tw_register_page()`. In this situation, Tapeworm increments a reference count for that physical page, but does not set any new memory traps. 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 only flushes the page from the simulated cache when the reference count reaches zero.

A minor modification to the `tw_register_page()` and `tw_remove_page()` primitives enables Tapeworm to support a form of cache set sampling to further enhance its speed. When implemented in a trace-driven simulator, set sampling uses a filtered trace containing exactly the addresses that map to a certain subset of cache sets [Kessler91, Puzak85]. The misses on these locations are used to form estimators for the total number of cache misses. Because less trace is used, set sampling can reduce trace-driven simulation times, but there is pre-processing overhead to construct a trace sample. Rather than filter addresses in software to obtain a sample, Tapeworm exploits its trapping framework to make the host hardware perform this function at a much lower cost. This is accomplished by modifying `tw_register_page()` to only set traps on memory locations that map to specific cache sets for a given sample. Memory locations that are not part of the sample never cause miss traps and are effectively filtered from the simulation with no overhead. The result is that Tapeworm slowdowns decrease in direct proportion to the degree of sampling. An additional benefit of this method is that different samples can be obtained simply by changing the pattern of traps on registered Tapeworm pages. With trace-driven simulation, the full trace must be re-processed to obtain a new set sample.

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 stored in an extended version of the OS task data structure. Attributes are set by calling `tw_attributes()` with a task identifier specifying which task to assign the new attributes to. A zero task identifier is used to indicate the OS kernel itself.

If `simulate` is zero (the default value), then the task runs in the system without any intervention from Tapeworm. If non-zero, `simulate` indicates that all current and future pages touched by the task must be registered with Tapeworm via a `tw_register_page()` call. A second attribute, `inherit`, defines the initial value of `simulate` for any children of the task. In other words, 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 greatly simplifies the simulation of workloads with complex task fork trees, such as `sdet`, `kenbus`, or a multi-stage optimizing compiler (see Table 4). 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.

The final primitive of Table 1, `tw_replace()`, is a direct analogue to the replacement routine of a trace-driven simulator. It maintains a data structure representing a simulated cache by inserting new entries and selecting others to be displaced according to some pre-defined simulation parameters. Because it is implemented entirely in software, simulation configurations are not restricted in any way by the TLB or cache structure of the underlying host hardware. For example, simulated caches may be either smaller or larger in size than the caches of the underlying host machine. A larger simulated cache simply sets fewer traps on a workload's memory locations. Similar adjustments can be used by `tw_replace()` to simulate different line sizes and associativities, as well as more complex cache structures including split, unified or multi-level caches. Additionally, because `tw_replace()` has access to the actual virtual-to-physical page mappings established by the VM system, it can simulate either virtual or physical cache indexing.

3.3 Design Summary

Because Tapeworm has kernel privileges and works in close cooperation with the VM system, it can include pages from any task, as well as the kernel itself. The inheritance of Tapeworm attributes during a task fork greatly simplifies the problem of capturing all activity from a complex multi-task workload. By allowing different combinations of tasks to have their cache effects simulated or not, Tapeworm attributes enable experiments that

1. Our implementation of Tapeworm on a DECstation 5000/200 makes use of a single-error correcting, double-error detecting ECC code. A trap is set by flipping a specific ECC check bit among the 7 total check bits assigned to each 32 bits of data. If Tapeworm detects a single-bit error in any of the other 38 check or data bit positions, or if it detects a double-bit error, it knows that a true error has occurred.
2. In the case of TLB simulation, where a page valid bits may be used by Tapeworm to set traps, an extra bit is maintained in software to indicate the true state of the page (resident in physical memory or not).

measure and isolate task interference effects. Finally, by fully optimizing the common case of cache hits and through its use of set sampling, Tapeworm is very fast.

4 Tapeworm Implementation and Experiences

We have implemented Tapeworm for TLB and instruction cache simulation in the Mach 3.0 operating system kernel running on a MIPS R3000-based DECstation 5000/200. The hardware mechanisms we selected to set and clear traps are page valid bits and ECC check bits. In this section, we use this implementation to study the effectiveness of the Tapeworm design in meeting our three main goals of simulation completeness, speed and portability. We also discuss difficulties encountered during the implementation, and suggest inexpensive hardware support that could increase Tapeworm's flexibility and further enhance its speed.

To validate the accuracy of Tapeworm results and to assist in the computation of metrics such as miss ratios, 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 comparison with trace-driven simulation, we use the Cache2000 memory simulator [MIPS88] driven by Pixie-generated traces [Smith91]. Note that Pixie only generates user-level address traces for a single task. The widespread use of Pixie makes it representative of trace-driven simulation environments.

Table 3 and Table 4 summarize the workloads used in this study. 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.

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.

Table 3: Workload Summary

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

4.1 Speed

The original implementation of the Tapeworm miss handler was written entirely in C and required over 2,000 cycles to execute. This is similar to cycle counts of 2,500 for a similar operation in the Wisconsin Wind Tunnel Simulator [Lebeck94]. We optimized the handler by re-writing it entirely in assembly code and by bypassing the usual kernel entry and exit code. The new code requires no execution stack and saves a minimal number of registers.

With careful coding, the final optimized handler requires only 250 cycles to handle simulated misses in direct-mapped caches with 4-word line sizes (see Table 5 for the components of this time). Higher degrees of associativity slightly increase the time in `tw_replace()`, while longer cache lines increase the cost of `tw_set_trap()` and `tw_clear_trap()`. Simulating different cache sizes has little effect on these times. Although the 250 cycle cost of a kernel trap is greater than the average cost to gen-

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

Table 4: Workload and Operating System Summary

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.

Cache Size	Miss Ratio	Cache 2000 Slowdowns	Tapeworm Slowdowns
1K	0.118	30.2	6.27
2K	0.097	28.8	5.16
4K	0.064	27.0	3.84
8K	0.023	24.2	1.20
16K	0.017	23.5	0.87
32K	0.002	22.4	0.11
64K	0.002	22.3	0.10
128K	0.000	22.0	0.01
256K	0.000	22.1	0.00
512K	0.000	22.1	0.00
1024K	0.000	22.3	0.00

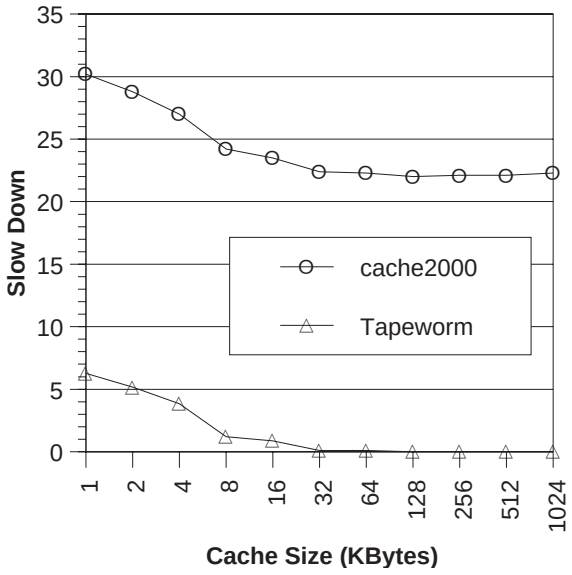


Figure 2: Comparison of Trace-driven and Tapeworm Slowdowns

Tapeworm slowdowns compared with 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). Because the Pixie/Cache2000 combination can only measure a single-task workload, Tapeworm attributes were set to measure activity only from the `mpeg_play` task and to exclude X display server, BSD UNIX server and kernel references. However, slowdowns in both cases were computed using the total wall-clock run time for the workload which includes time in the X and BSD servers.

erate and process a trace address (about 40 to 60 cycles per address for Cache2000 and Pixie), Tapeworm traps only occur on misses, while a trace-driven simulator must consider all addresses whether they hit or miss. This suggests a rough break-even ratio of 4 hits to 1 miss[1] before Tapeworm becomes slower than Cache2000 (see bottom of Table 5). Because only the most poorly performing caches exhibit miss ratios of 0.20 or higher, Tapeworm typically outperforms trace-driven simulation by Cache2000. Moreover, in contrast to trace-driven simulation, Tapeworm works better the larger the cache (and thus the smaller the miss ratio).

Routine Name	Instructions
kernel trap and return	53
`tw_cache_miss()`	23
`tw_replace()`	20
`tw_set_trap()`	35
`tw_clear_trap()`	6

Cycles per miss in Tapeworm	246
Cycles per address in Cache2000	53

Table 5: Tapeworm Miss Handling Time

This table shows the total number of cycles required to handle a Tapeworm cache miss, along with the components of the handler in numbers of instructions. These times are for simulation of direct-mapped caches with 4-word line sizes. For comparison, we also show the average number of cycles per address (hit or miss) for a Cache2000 simulation. Note that this average includes the time to generate addresses on-the-fly by a Pixie-annotated workload.

1. This break-even point is only an approximation. Because the amount of processing differs for hits and misses, the average number of cycles per address in Cache2000 varies, depending on the ratio of hits to misses.

The best measure of overall simulation speed is the actual wall-clock time required to perform a simulation. We are interested in comparisons with other on-the-fly simulation techniques able to measure lengthy computations. We therefore define *Slowdown* as the ratio of simulation and trap (or trace generation) overhead to the run time of an uninstrumented workload:

$$Slowdown = \frac{Tapeworm\ Overhead}{Normal\ Workload\ Run\ Time}$$

$$Slowdown = \frac{Pixie + Cache2000\ Overhead}{Normal\ Workload\ Run\ Time}$$

Overhead is the time added to a workload run either by Tapeworm or Pixie and Cache2000. *Normal Workload Run Time* is for an unmodified run on the host machine, a DECstation 5000/200.

Figure 2 shows how Tapeworm slowdowns vary with cache size for the `mpeg_play` workload in comparison with Cache2000. With both simulators, slowdowns decrease as cache size increases, but for different reasons. Tapeworm slowdowns decrease to nearly zero with larger caches because the number of traps to the Tapeworm miss handler approaches zero. Cache2000 slowdowns decrease because slightly less manipulation of its data structures is required for hits (search only) than for misses (search and replace). However, Cache2000 slowdowns never fall below about 20, even for the largest caches. For the smallest size cache

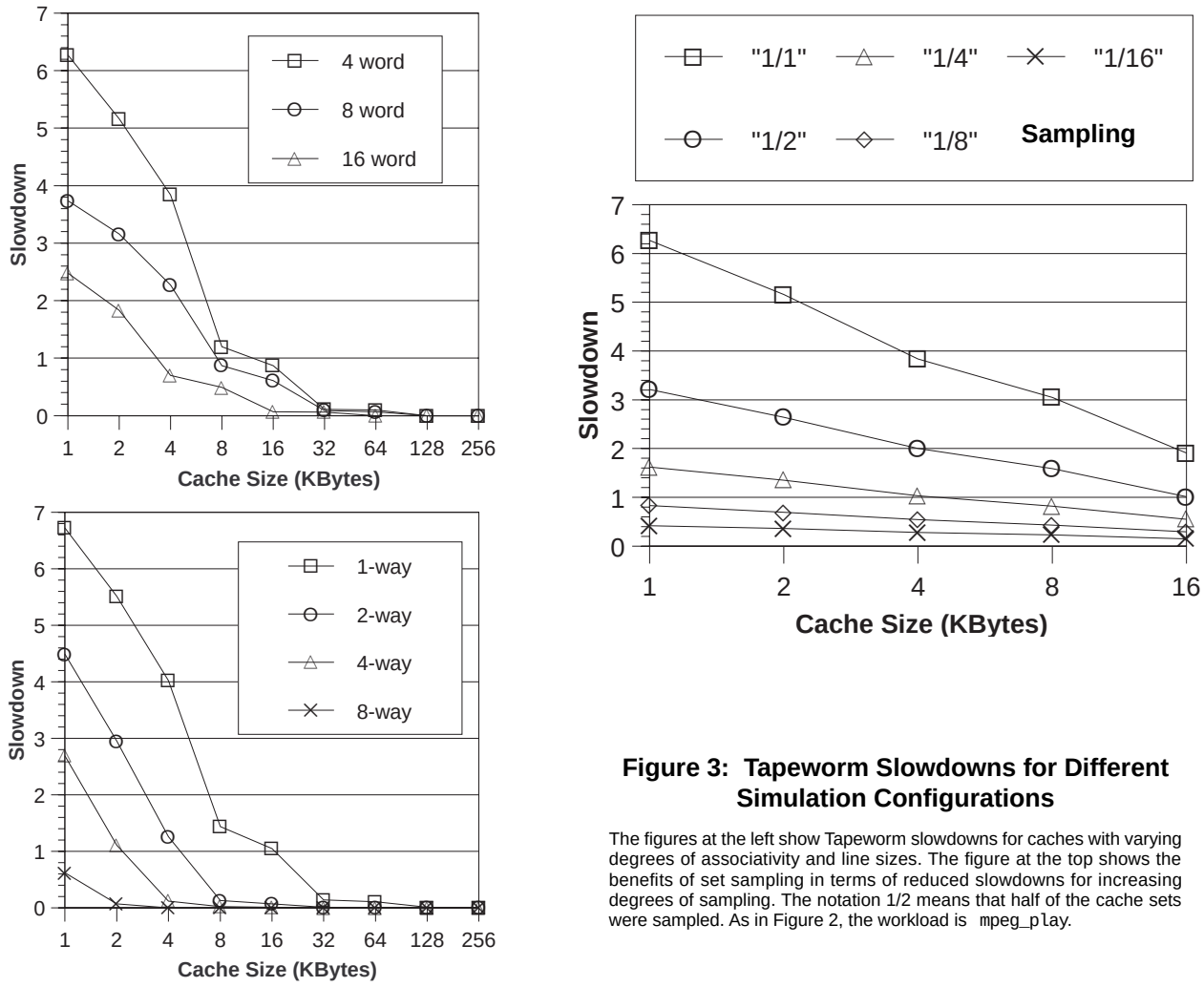


Figure 3: Tapeworm Slowdowns for Different Simulation Configurations

The figures at the left show Tapeworm slowdowns for caches with varying degrees of associativity and line sizes. The figure at the top shows the benefits of set sampling in terms of reduced slowdowns for increasing degrees of sampling. The notation 1/2 means that half of the cache sets were sampled. As in Figure 2, the workload is `mpeg_play`.

(1 K-byte with a miss ratio of 0.118), Tapeworm still out-performs Cache2000 by a factor of about 3.

Although simulations of caches with higher associativities and larger cache lines increase Tapeworm miss handling time slightly, these structures typically experience fewer misses overall, and thus actually lead to faster simulation. Figure 3 shows Tapeworm slowdowns over a broader range of cache configurations.

All slowdowns reported thus far are for simulations without any sampling. Figure 3 illustrates the speed benefits of set sampling. We give results only for the smallest cache sizes, noting that Tapeworm slowdowns for larger caches are sufficiently small to avoid the need for sampling altogether. Notice that slowdowns decrease in direct proportion to the fraction of sets sampled. However, sampling does increase measurement variance. We examine this effect, in addition to other sources of variation in performance measurements in the next section.

4.2 Completeness and Accuracy

In this section, we examine issues of simulation accuracy. We begin by illustrating the importance of including multi-task and kernel references by comparing the relative contributions of different workload components (user, server or kernel)[1] to overall I-cache miss counts. By isolating the components in this way, we are also able to partially validate our results by comparing the user component of single-user-task workloads with Pixie-driven Cache2000 simulations. Next, we study problems of measurement variation due both to OS effects and set sampling, by using various Tapeworm features to isolate, measure and effectively remove the individual contributions of these effects to overall measurement variance. Finally, we conclude with some comments on measurement bias due to Tapeworm's presence in a running system.

Miss Contributions of Workload Components

Table 6 shows typical 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 themselves 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 interfer-

1. By *user task*, we mean any of several tasks that are children of the shell from which the workload was initiated. We lump all of these tasks together in our simulations by using 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`.

Workload	From Traces	User Tasks	Servers	Kernel	All Activity	Interference
eqntott	0.06 (0.000)	0.07 (0.000)	2.52 (0.002)	2.44 (0.002)	8.44 (0.007)	3.41 (0.003)
espresso	1.60 (0.003)	1.80 (0.003)	2.28 (0.004)	1.96 (0.004)	9.53 (0.018)	3.49 (0.007)
jpeg_play	2.98 (0.002)	3.14 (0.002)	14.58 (0.008)	9.21 (0.005)	36.28 (0.020)	9.35 (0.005)
kenbus	———	7.50 (0.043)	11.89 (0.068)	12.78 (0.073)	45.70 (0.260)	13.53 (0.077)
mpeg_play	37.63 (0.027)	37.91 (0.027)	33.92 (0.024)	19.27 (0.014)	112.5 (0.079)	21.39 (0.015)
ousterhout	———	1.93 (0.003)	18.62 (0.033)	21.72 (0.038)	61.39 (0.108)	19.12 (0.034)
sdet	———	20.14 (0.024)	25.18 (0.031)	18.09 (0.022)	104.6 (0.127)	41.25 (0.050)
xlisp	85.77 (0.061)	90.02 (0.064)	6.31 (0.004)	2.98 (0.002)	135.8 (0.096)	36.55 (0.026)

Table 6: 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.*

ence among the individual workload components, the sum of the individual miss columns is less than the *All Activity* column.

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 only considers 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, incidentally, performs much better in a cache only slightly larger.

As noted above, we compared Tapeworm miss counts from the user task components of each workload with Pixie-driven Cache2000 simulations for the purposes of validation. Wherever a comparison was possible (i.e., with the single-user-task workloads), the Tapeworm miss counts for the user portion of the workload were nearly identical to those reported by Cache2000. As we shall see in the next sections, measurement variation and bias makes validating Tapeworm results for the other workload components (e.g., the servers and kernel) is an inherently more difficult problem.

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. However, the precise sequence of traps that drive a Tapeworm simulation are impossi-

Workload	Misses ($\bar{x}$) ($\times 10^6$)	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%)

Table 7: 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. $\bar{x}$ 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.*

Size (KBytes)	Misses ($\bar{x}$) (x 10^6)	s (x 10^6)	
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%)

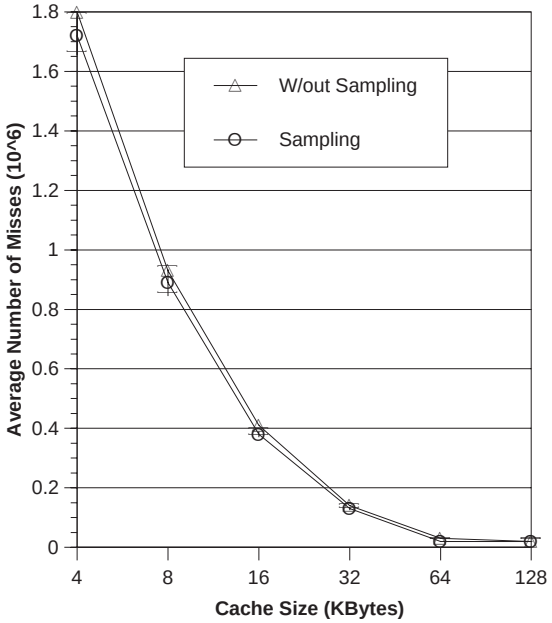


Table 8: Variation due to Set Sampling

This table isolates the degree to which set sampling can vary cache performance measurements. 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 of 16 trials each. The error bars on the plot represent one standard deviation.

Size (KBytes)	Misses ($\bar{x}$) (x 10^6)	s (x 10^6)	
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%)

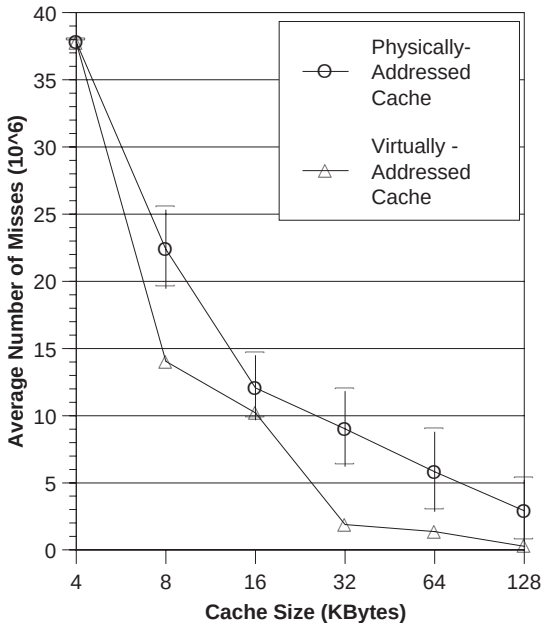


Table 9: Variation Due to Page Allocation

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.

Workload	Misses ($\bar{x}$) (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%)

Table 10: Measurement Variation Removed

These measurement were made as in Table 7, 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.

ble to reproduce from run to run because of dynamic system effects. For example, the distributions of physical page frames allocated to a task, which change from run to run, affect the sequence of addresses seen by a physically-indexed cache [Kessler92, Sites88]. This, in turn, causes variation in cache miss ratios. Another source of measured performance variance is caused by Tapeworm itself when it employs set sampling.

The results in Table 7 measure the extent of these effects on our workload suite by reporting statistics from multiple experimental trials. These measurements are for simulations of a 16 K-byte, physically-indexed cache using 1/8th sampling of the cache sets. Note that the 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. New trials were then performed with and without sampling. The results are shown in Table 8 for an example with `espresso`. As expected, results without sampling show zero variance over multiple trials of the experiment. 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 increased slowdown of the non-sampled experiments.

Table 9 shows the degree that page allocation 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. The virtually-indexed cache simulation exhibits 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. Notice 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 percent of the mean) appears at a cache size of 32 K-bytes, which is roughly the size of program text used by `mpeg_play`. Variation decreases for both larger and smaller caches. This observation is consistent with a probabilistic model of cache page conflicts published in [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, and decreases for larger and smaller caches.

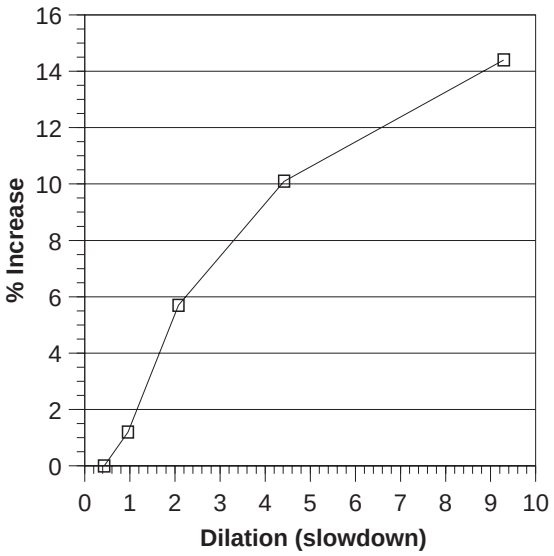
Finally, notice that variation due to page allocation is comparable to, if not larger than, that of set sampling. This suggests that the error introduced by sampling is a reasonable trade for increased speed when simulating physically-indexed caches. Of course, the combined effect of both sources of variance is greater than either in isolation, forcing a larger number of trials to be performed to increase the level of confidence in the mean value.

In addition to page allocation, we have observed other sources of memory system performance variation due to OS effects. For example, we have observed gradual (but substantial) increases in TLB misses due to kernel and server memory fragmentation in a long-running system. It is important to note that Tapeworm's sensitivity to these and other sources of performance variation, which necessitate multiple experimental trials, is not a liability. Performance variations due to page allocation and memory fragmentation are real system effects that should be considered. 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 10.

Sources of Measurement Bias

Tapeworm's presence in the kernel during a workload's operation raises questions of measurement bias. Although Tapeworm carefully avoids setting traps on its own code and data so that it never directly changes miss counts, there are indirect ways that Tapeworm can alter results. First, about 256 K-bytes of physical memory are allocated for Tapeworm at boot time. This removes 64 pages from the free memory pool, resulting in a possible increase in paging activity. We minimize this problem by adding enough additional physical memory so that paging is avoided altogether.

A second source of error is cause by Tapeworm slowdowns resulting in system *time dilation*, an effect that causes more clock interrupts during the run of a workload, leading to increased cache conflict misses. Figure 4 plots 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



Dilation (slowdown)	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%

Figure 4: 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.

workload, time dilation cannot be removed by a simple adjustment to the clock interrupt frequency as is done in [Borg90, Chen93b]. We are collecting time dilation curves for a larger set of workloads to determine if their shape and magnitude are the same as in Figure 4. If so, it should be possible to adjust simulation results to factor away this form of systematic error.

A final source of bias is related to 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 cache misses seen by Tapeworm. Because only the kernel runs with interrupts masked, this limitation only affects kernel references. Further, only a very small fraction of kernel code is affected, and special code around these regions helps Tapeworm to take their cache effects into account.

4.3 Portability

To ease portability, Tapeworm has been carefully partitioned into hardware-dependent and hardware-independent sections. Further, only a minimal amount of code actually runs in the kernel, controlled through a system call interface by a user-level X application (see Table 11). The hardware-dependent code consists primarily of modified kernel entry code and two routines, `tw_set_trap()` and `tw_clear_trap()` that can, in principle, be implemented on many machines (see Table 12). In practice, unexpected interactions between components in a memory system can hinder attempts to implement these primitives on some architectures. 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.

Code	Lines	%
Machine-dependent Kernel Code	343	5%
Machine-independent Kernel Code	889	13%
Machine-independent User Code	5652	82%

Table 11: Tapeworm Code Distribution

Intentional hardware support for these primitives could help to avoid these problems, and also reduce the time to set and clear traps. In our implementation, these operations are performed by issuing a convoluted sequence of control instructions to the memory-controller ASIC that implements the ECC logic. Piecing together the memory address of an ECC error (i.e., the address of a Tapeworm cache miss) also requires a dozen load, shift, add and mask instructions for what could be supported by a single load. We believe that a cleaner interface to the diagnostic functions of the memory ASIC could reduce the total miss-handling time to about 50 cycles, further increasing Tapeworm's speed by another factor of 5.[1] Although more expensive, direct support in the form of a trap bit for each memory location could further decrease the cost of setting and clearing traps. Such support would be useful for other applications, such as debuggers and distributed shared memory [Appel91].

Despite these problems, ports of Tapeworm now run in the OSF/1 and Mach 3.0 operating systems under DECstation 3100s, DECstation 5000/200s, and 486-based Gateway PCs.

4.4 Flexibility

With respect to flexibility, Tapeworm has trouble simulating memory structures that do not fit a cache model. For example, write buffers, which are queues that only hold their contents for only a short time, cannot be simulated with the Tapeworm algorithm. This limitation restricts simulations to a write-back write policy. Furthermore, unlike trace-driven simulation which can easily and efficiently be extended to the simulation of other architectural structures, such as instruction pipelines, the trap-driven approach seems to be limited to the simulation of memory system hierarchies and their components.

Other problems of flexibility are not inherent to trap-driven simulation, but are related to the specific limitations of the host hardware. For example, on the DECstation 5000/200, ECC bits are checked on 4-word cache line refills. This effectively limits the simulation of Tapeworm cache line sizes to multiples of 4 words on this machine. Our attempts to implement data cache simulation on this particular machine were hindered by its no-allocate-on-write policy, which causes ECC traps to be cleared without invoking the Tapeworm miss handlers. On machines that use an allocate-on-write policy, data cache simulations are possible [Reinhardt93]. Finally, although the miss counts provided by

1. A similar operation performed by the miss handler of the R3000 software-managed TLB requires only about 20 cycles.

Privileged Operation	MIPS R3000	MIPS R4000	SPARC	DEC Alpha	Tera	Intel i486	Intel Pentium	AMD 29050	HP PA-RISC	Power PC
Memory Parity or ECC Traps	Yes	Yes	Yes	Yes	Yes		Yes			
Instruction Breakpoint	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Data Breakpoint	No	No	No	No	Yes	No	No	No	No	No
Invalid Page Traps	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Variable Page Size	No	Yes	No	Yes		No	Yes	Yes	Yes	Yes
Instruction Counters	No	No	No	Yes		No	Yes	No		No

Table 12: Privileged Operations on Modern Microprocessors

The entries in this table were taken from a variety of sources including data books, text books and Microprocessor Report [MReport92, MReport93]. A given entry may not be true for every implementation of a given processor. Some features, such as memory-parity-error traps are actually system-implementation dependent. For these features, an affirmative entry means that we found at least one system with the given microprocessor that implements the feature. A blank entry means that insufficient data was available.

Tapeworm are useful metrics in their own right, some studies require other measures, such as miss ratios or misses per instruction (MPI). We obtain instruction counts using a logic analyzer, but a much more convenient method would be an on-chip instruction counter. In each of these cases, intentional hardware support for trap-driven simulation primitives could overcome these problems.

5 Summary and Future Work

The development of Tapeworm demonstrates that "on-the-fly" cache and TLB simulation driven by kernel traps can greatly simplify the problem of evaluating memory structures under workloads including multiple tasks and operating system loads. Moreover, our measurements of Tapeworm's performance show that these simulations can be performed with rather small degradation in the overall system performance. This opens up important new areas for consideration:

- Fast simulation creates the possibility of examining a wider range of alternative configurations, and investigating the variability of results for repeated runs of the same workload.
- Simulations can be driven by the memory references generated during an actual user's session, because Tapeworm slowdowns can be made imperceptible to the user. This makes it possible to watch for interesting cases that cannot be identified by traditional batch simulations.
- The use of continuous monitoring and simulation opens up the possibility of using these results to perform real-time hardware and software tuning.

Future generations of simulators and monitors driven by kernel traps would benefit from better hardware support for generating traps on both reads and writes to particular memory locations. Better support will result in even faster and more flexible simulations.

We are continuing to develop and add features to the Tapeworm simulator. We are currently adding data-cache simulation capabilities and are porting Tapeworm to other architectures, including DEC Alpha-based workstations and SPARC-based machines.

6 Acknowledgments

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

Bibliography

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

[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.

[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.

[Anderson91] Anderson, T. E., Levy, H. M., Bershad, B. N., et al. *The interaction of architecture and operating system design*, In 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 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. E. and Wall, D. W. *Generation and analysis of very long address traces*, In The 17th Annual International Symposium on Computer Architecture, IEEE, 1990.

[Chen93b] Chen, B. *Software methods for system address tracing*, In The 4th Workshop on Workstation Operating Systems, Napa, California, 1993.

[Chen93a] Chen, B. and Bershad, B. *The impact of operating system structure on memory system performance*, In Proc. 14th Symposium on Operating System Principles, 1993.

[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 SIGMETRICS, Nashville, TN, ACM, 128-137, 1994.

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

[Eggers90] Eggers, S. J., Keppel, D. R., Koldinger, E. J., et al. *Techniques for efficient inline tracing on a shared-memory multiprocessor*, In SIGMETRICS Conference on Measurement and Modeling of Computer Systems, ACM, 34-47, 1990.

[Flanagan92] Flanagan, K., Grimsrud, K., Archibald, J., et al. *BACH: BYU address collection hardware*. Brigham Young University. TR-A150-92.1. 1992.

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

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

[Hsu89] Hsu, P. *Introduction to Shade*. Sun Microsystems. 1989.

[Kessler91] Kessler, R. *Analysis of multi-megabyte secondary CPU cache memories*. 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.

[Larus90] Larus, J. R. *Abstract Execution: A technique for efficiently tracing programs*. University of Wisconsin-Madison. 1990.

[Larus93] Larus, J. R. *Efficient program tracing*. *IEEE Computer* **May, 1993** : 52-60, 1993.

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

[Magnusson93] Magnusson, P. S. *A design for efficient simulation of a multiprocessor*, In MASCOTS '93 - Proceedings of the 1993 Western Simulation Multiconference on International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, La Jolla, California, 1993.

[Martonosi92] Martonosi, M., Gupta, A. and Anderson, T. *MemSpy: Analyzing memory system bottlenecks in programs*, In 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 SIGMETRICS, Santa Clara, California, ACM, 248-259, 1993.

[Mattson70] Mattson, R. L., Gecsei, J., Slutz, D. R., et al. Evaluation Techniques for Storage Hierarchies. IBM Systems Journal 9 (2): 78-117, 1970.

[MReport92] Report, M. Sebastopol, CA, MicroDesign Resources, 1992.

[MReport93] Report, M. Sebastopol, CA, MicroDesign Resources, 1993.

[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 Fourth 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*. The University of Michigan. CSE-TR-147-92. 1992.

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

[Nagle94] Nagle, D., Uhlig, R., Mudge, T., et al. *Optimal Allocation of On-chip Memory for Multiple-API Operating Systems*, In 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*. University of California, Berkeley. 1992.

[Puzak85] Puzak, T. *Cache-memory design*. University of Massachusetts. 1985.

[Reinhardt93] Reinhardt, S., Hill, M., Larus, J., et al. *The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers*, In SIGMETRICS 93 (Special Issue of Performance Evaluation Review), Santa Clara, CA, ACM, 48-60, 1993.

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

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

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

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

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

[Talluri94] Talluri, M. and Hill, M. *Surpassing the TLB Performance of Superpages with Less Operating System Support*, In ASPLOS-VI, San Jose, CA, ACM, In this proceedings, 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 Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, ADM, 162-174, 1992.

[Uhlig94a] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., *Kernel-based Memory Simulation (Extended Abstract)*, In SIGMETRICS, Nashville, TN, University of Michigan, 286-287, 1994.

[Uhlig94b] Uhlig, R., Nagle, D., Stanley, T., S. Sechrest, T. Mudge, R. Brown, *Design tradeoffs for software-managed TLBs*. ACM Transactions on Computer Systems. To appear in Fall, 1994.