Collecting Whole-System Reference Traces of Multiprogrammed and Multithreaded Workloads

Scott F. Kaplan
Department of Mathematics and Computer Science
Amherst College
Amherst, MA 01002-5000
sfkaplan@cs.amherst.edu

ABSTRACT

The simulated evaluation of memory management policies relies on *reference traces*—logs of memory operations performed by running processes. No existing approach to reference trace collection is applicable to a complete system, including the kernel and all processes. Specifically, none gather sufficient information for simulating the virtual memory management, the filesystem cache management, and the scheduling of a multiprogrammed, multithreaded workload. Existing trace collectors are also difficult to maintain and to port, making them partially or wholly unusable on many modern systems.

We present *Laplace*, a trace collector that can log every memory reference that a system performs. Laplace is implemented through modifications to a simulated processor and an operating system kernel. Because it collects references at the simulated CPU layer, Laplace produces traces that are complete and minimally distorted. Its modified kernel also logs selected events that a post-processor uses to associate every virtual memory and filesystem reference with its thread. It also uses this information to reconcile all uses of shared memory, which is a task that is more complex than previously believed. Laplace is capable of tracing a workload without modifications to any source, library, or executable file. Finally, Laplace is relatively easy to maintain and port to different architectures and kernels.

1. INTRODUCTION AND MOTIVATION

One approach to studying memory management policies is to simulate the servicing of memory requests that real processes would perform. A *reference trace* is a log of the memory references performed by a real application processing real input. With a set of reference traces, one memory management policy can be compared with another using *trace-driven simulation*, where both policies are made to service the identical reference sequences in a reproducible manner.

Trace-driven simulation is used to study every level of the memory hierarchy, from hardware caches to page replacement policies. Consequently, many *trace collectors*—tools that capture and log memory references—have been developed [1, 2, 7, 11, 14]. Below is a brief overview of collection techniques that have been widely used.

- **Microcode modification:** Alter the processing of instructions at the microcode level so that each instruction logs its own memory references (e.g. *ATUM* [1]).
- **Instruction set simulation:** Process the instructions of an executable in software, logging the memory reference instructions (e.g. *Shade* [2]).
- **Static code annotation:** Insert instructions that log memory references into the executable code to be traced (e.g. *ATOM* [14], *QPT* [7], *Etch* [11]).

These collectors have limitations that make them either insufficient for whole-system trace collection or unavailable for use on modern hardware.

- No existing tool logs kernel-level information to later reconstruct the sharing of memory, the CPU scheduling of threads, or the use of the filesystem cache.
- Only ATUM is capable of tracing a whole-system workload, including the kernel and all processes. ATUM is dependent on a microcode layer that no longer exists in most modern processors.
- No static code annotator can trace dynamically compiled code, and some cannot trace dynamically loaded library code. ATOM operates only on executables with sufficient symbol information.
- The instruction set simulators and static code annotators are complex, making them difficult to maintain and to port.

Existing trace collectors have also optimized *capturing overhead*, which is the slowdown incurred by the traced workload in order for the trace collector to obtain control at each reference. However, we show in Section 4 that *handling overhead*—the time required to store or process each capture reference—dominates total trace collection time. Thus, the capturing performance of these tools is irrelevant.

A new approach. *Laplace* is a trace collector that addresses these problems. It is designed to collect the references of the kernel and of all processes. It is also designed to collect sufficient information about kernel events so that it can, in a post-processing phase, reconstruct the state of the system at each moment. Laplace is implemented as three components:

1. **Modified machine simulator:** There are existing software packages that can simulate different processors and their typical supporting hardware [8, 9]. They are sufficiently complete to support the execution of an unmodified operating system. We extended a machine simulator to log the occurrence of each load, store, or instruction fetch.

2. **Modified kernel:** We want to associate each reference with a process and thread.[1] Furthermore, we want to determine which pages are shared and which pages are part of the filesystem cache. We have modified a kernel to log the information needed for such reconstruction.

3. **Post-processor:** Using the logs produced by the two previous components, this component interleaves the records, reconstructing the state of kernel, processes, threads, and mappings. This component can be made to emit reference traces with this combined information to support different types of simulation.

Laplace can gather traces for any application without any recompilation, relinking, or other modification to the executable, regardless of the manner in which the code for the application was generated. While it can collect a whole-system workload, the post-processor is capable of separating the reference stream of any one process or thread so that it may be used in uniprogrammed simulations.

The reference traces gathered by this tool can used for a wide range of memory management studies. Because of its accuracy, the traces gathered can be used for hardware cache simulations. Because of the ability to track the use of shared memory and filesystem caching, simulations of multiprogrammed virtual memory management and filesystem cache management are possible at a level of detail not previously achieved. We are aware of no other tool that captures both virtual memory references and filesystem cache references, thus allowing the simulation of memory management policies that must arbitrate between these two uses.

Our most surprising result is that, although Laplace is built upon a machine simulator that yields much slower executions than a real system (approximately 227-fold slowdown), the most limiting factor in trace collection is instead the processing and storage of collected references—a problem that will exist for any trace collector.

Road-map. In Section 2, we will provide more detailed information about existing tracing tools. Section 3 will describe thoroughly the components of Laplace. We will examine the performance of Laplace in Section 4, and address the speed limitations of collecting traces. Finally, we will review our development of Laplace and its uses, as well as present topics for future work, in Section 5.

[1]We use this term only for kernel-level threads.

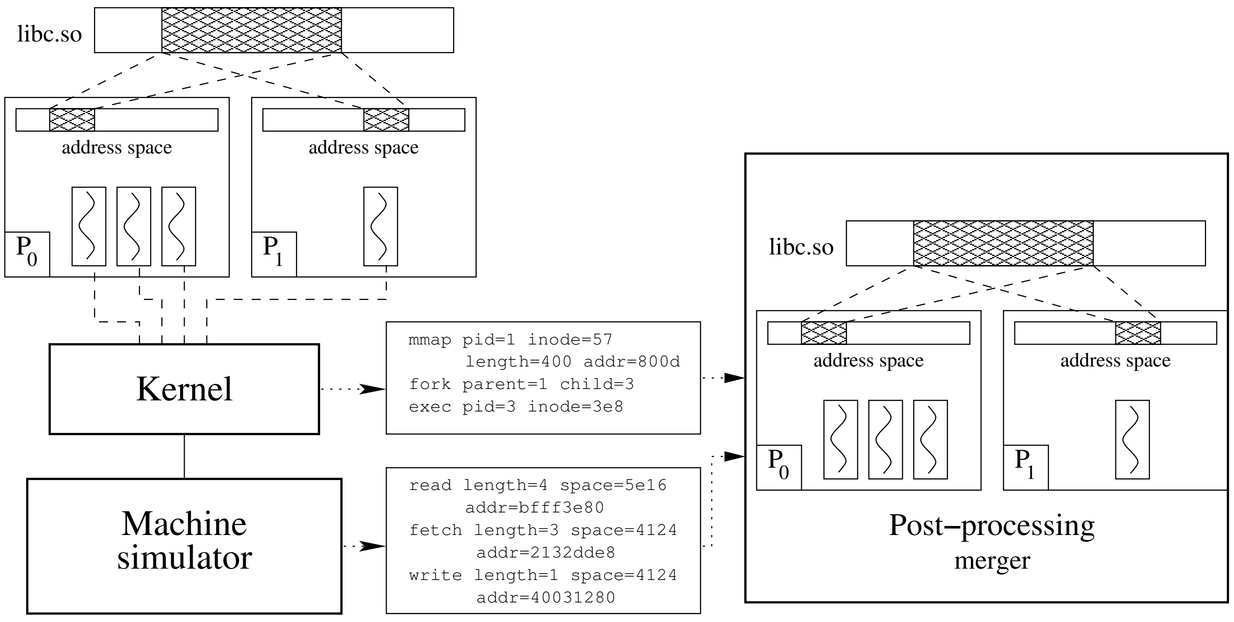
2. EXISTING METHODS

In this section, we will examine more closely the trace collection methods described in Section 1. There are a number of characteristics that describe a trace collector, and the importance of each depends on the context in which the traces will be collected and used. Below we have extended terminology codified in [15] to provide an overview of these characteristics:

- **Completeness:** Can a collector capture all references for any application? We break down this characteristic into a number of subcategories:
  - *Thoroughness:* Are all references traced? Some collectors may not be able to capture references to certain segments of the address space or references of certain types (e.g. instruction fetches).
  - *Applicability:* Can any application be traced? A method may operate on only a subset of applications, such as those that are linked with certain symbol information, or those that do not dynamically compile code.
  - *Concurrency:* Can multiprogrammed workloads be traced? A collector operate only on a single process at a time.
- **Distortion:** How accurately do the traces record the real reference stream? There are a number of ways in which the a logged sequence of references can differ from the real sequence:
  - *Discontinuities:* A collector may introduce a gap in tracing where references are not recorded, most often when emptying an output buffer.
  - *Time dilation:* The duration or even the ordering of recorded events may be altered from the untraced execution. This effect is typical, as tracing requires its own computation time, thus interferes with the normal timing.
  - *Memory dilation:* Tracing may alter the amount of space that a workload seems to use. It may also alter placement of data in the virtual address spaces of the workload.
  - *Self-tracing:* A collector may collect references to its own code or data.
- **Detail:** How much relevant information is logged by a collector? More information will allow for more accurate simulation.
- **Portability & maintainability:** How closely is the collector tied to a particular instruction set, hardware architecture, or operating system kernel? How complex is the collector's implementation?
- **Efficiency:** How much more slowly will an application execute while it is being traced?

2.1 Existing collectors

We will examine some of the most cited and used collectors, highlighting their trade-offs and limitations. Again, a more thorough treatment of these collectors can be found in [15].



```
mmap pid=1 inode=57
     length=400 addr=800d
fork parent=1 child=3
exec pid=3 inode=3e8
```

```
read length=4 space=5e16
     addr=bfff3e80
fetch length=3 space=4124
     addr=2132dde8
write length=1 space=4124
     addr=40031280
```

Figure 1: The modified machine simulator and modified kernel each emit logs that are consumed by the post-processor, which interleaves the records and reconstructs the relationships between processes, threads, and address spaces, as well as mappings of shared spaces in each virtual address space.

ATUM. This collector modified the microcode in a CPU for each instruction so that it would log its own memory references [1]. Because this approach operates at the CPU level, it is complete it all senses. ATUM is also exceedingly efficient for a trace collector, slowing execution only ten- or twenty-fold.

While ATUM collects perfectly detailed traces, the modified microcode introduces time dilation. Also, ATUM was never extended also to collect kernel-level events. Furthermore, in [15], Uhlig and Mudge note that ATUM can log task-switching events. However, some kernels (e.g. Linux 2.4) do not rely on the processor-level task management features, and instead make each process or thread "soft", performing task switching with common instructions. In such a case, ATUM would not be able to record those task-switching events.

Uhlig and Mudge also observe that ATUM can capture page table updates, and thus emit new virtual-to-physical mapping records. As we will see in more detail in Section 3.2, such information is insufficient to reconstruct the sharing of memory between processes.

The greatest limitation to ATUM is its portability. While it is a simple matter to port ATUM to different microcode-based architectures, it is impossible to bring ATUM to modern processors that do not use microcode. It may be possible to bring this approach to a modern, reprogrammable CPU such as the Transmeta Crusoe. However, there would still exist a need for a modified kernel, and as we will see in Section 4, the potential performance gain over a simulated processor will be negated by the bottleneck of storing the reference trace.

Shade. This trace collector is a sophisticated instruction set simulator [2]. When an application is run under Shade, the executable code is dynamically recompiled from the simulated instruction set to the instruction set of the underlying machine. During recompilation, Shade can insert additional instructions. The recompiled code is then executed on the host processor.

One great strength of Shade is its efficiency, slowing execution by a factor of approximately 10 to 15. Another is its thoroughness and applicability: all references of a process are collected, and any application, even those that dynamically compile code at runtime, can be traced.

Shade introduces both time and memory dilation. Since the code is recompiled and new instructions interleaved, the memory layout of the code and the timing of the instructions is affected. Shade cannot trace concurrently, and so it cannot trace multiprogrammed workloads or kernel behavior. Shade has also never been integrated with a kernel in order to log mapping events that would allow the detection of shared memory spaces.

Shade is, in principle, portable and maintainable. However, its complexity changes those characteristics in practice. Shade is tightly tied not only to the architecture that it is simulating, but also to the host architecture and operating system on which it is running. The efficiency that it achieves is a result of involved optimizations that would be difficult to bring to another platform. Due to its complexity, Shade is no longer maintained, and does not simulate modern processors.

ATOM, QPT, and Etch. These collectors perform static code annotation, inserting code that logs references around every instruction, thus making an application produce its own reference trace [7, 11, 14]. These tools are sufficiently similar to one another that we will review them together.

These collectors share a number of strengths with Shade.

They are roughly as efficient, slowing execution by factors of about 10 to 50. They are also thorough, tracing every load, store, and instruction fetch. They introduce only modest distortion in the form of time dilation due to trace gathering overhead itself. However, these collectors do not share Shade's applicability, as annotation cannot always be performed on dynamically loaded libraries, and can never be performed on dynamically compiled code. Worse, these collectors need some symbol information to be available, and will not operate on "stripped" executables.

These collectors share additionally many of Shade's weaknesses. They cannot trace multiprogrammed workloads. They cannot record page mapping, thread scheduling, or other kernel-level information.[2] They are tightly tied to a given instruction set, and are additionally tied to an executable format. Consequently, many of these tools are either unavailable or work only partially on modern systems.[3]

2.2 Filesystem caching and shared memory

The literature on reference tracing tools typically ignores two important aspects of memory management: *filesystem caching* and *shared memory.* The study of filesystem caching has largely been isolated from the study of virtual memory. However, the filesystem cache and the virtual memory system must compete for the same memory resources. In real kernels, filesystem caching is often subsumed into the VM system, and filesystem cache pages are managed in the same manner as virtual memory pages. However, no existing tool has logged the use of both types of memory simultaneously.

Furthermore, there is no discussion for existing tracing tools about the shared memory used by a process. Shared spaces have not been examined for their interaction with different policies, as no trace collector has identified and correlated those pages shared by multiple processes.

3. LAPLACE IN DETAIL

The first two components of Laplace—the modified processor simulator and the modified kernel—each produce a separate log. The third component—the post-processor—reconciles the information contained in those two logs and produces a single output trace that can be used as input to a memory system simulator. In this section, we will describe these three components in detail.

3.1 Processor simulation

We use *Bochs* [8], an open-source, Intel `ia32` machine simulator, as the basis for the first component of Laplace. Bochs was suitable because the direct access to the source code made modifications simple. However, the modifications required at this level are small in number and simple in nature, thus making any machine simulator a feasible choice. *Simics* [9] is a commercial product that simulates a number of different architectures and could also serve as the basis for this level of Laplace. Similarly, there are processor simulators such as *Dynamic Simple Scalar* [3] that provide architecturally accurate simulations of hardware CPUs. By using such a simulator, Laplace could avoid a key form of time dilation.

Capturing references. For Laplace, the machine simulator is expected to generate only a *raw reference trace*—a log of every reference. We needed only to insert a call to a *logging function* at two essential locations in the machine simulator's code: at the data read/write function, and at the instruction fetch function. At these locations, execution is made to jump into a module responsible for emitting the log of references.

The elements of a reference event. Laplace logs the following information about each reference:

- **Type:** A load, store, or instruction fetch. This information can be determined at the point where the logging function is called.
- **Virtual address:** The referenced memory location, obtainable at the point where the logging function is called.
- **Kernel or user:** Whether the kernel or a user-level process performed the memory reference. This determination is specific to a kernel and its layout of kernel and process memory, and can be determined by examining the combination of address space and address information.
- **Timestamp:** The time, taken from the CPU's cycle counting register, at which the reference was performed.

Implementation complexity. The modifications required to Bochs are small in number and simple in nature, thus making any machine simulator a feasible choice. None of the original lines of code are altered or removed; we only added code totaling 873 lines. 830 of these lines are for a new module that interacts with modified kernel, buffers the captured references, and emits the reference and kernel event traces. The remaining 43 lines are inserted into the existing *Bochs* code, calling on the new module at each memory reference and providing the user with the ability to toggle reference trace collection.

None of the modifications to Bochs are architecture specific, and could easily be applied to other CPU simulators. *Simics* [9] is a commercial product that simulates a number of different architectures and could also serve as the basis for this level of Laplace. There are also CPU simulators such as *Dynamic Simple Scalar* [3] that provide architecturally accurate simulations of hardware CPUs. This type of simulator would allow Laplace to gather reference traces with timing information that may be relevant for the smallest hardware caches that are closest to the CPU.

3.2 Kernel modification

If one seeks to simulate a multiprogrammed workload, it will be insufficient for the input trace only to be a log of the memory references performed by a processor. Such a simulation must attribute each reference to a particular thread. Moreover, it must correlate pages that are shared among multiple processes and identify pages that are used for filesystem caching.

[2]Some of the information could be gathered by logging the system calls performed by the process. However, information about virtual memory mappings for fork-induced copy-on-write sharing and filesystem caching cannot be obtained by capturing system calls.

[3]Specifically, Etch and QPT are no longer distributed, and old version do not work on current platforms.

Such information can be obtained only from the kernel, and so our implementation of Laplace uses a modified version of *Linux 2.2.21* to log relevant events into a *kernel trace*. We will list those relevant events below, and we will describe the mechanism by which those events are logged.

Process/thread events. In Linux, the basic unit of execution is the *task*, where each task corresponds to one kernel-level thread from some process. In order to correlate a task with its references, Laplace logs the time at which each task is created (via the `fork()` system call), is scheduled on the CPU, is assigned a new virtual address space, and is terminated (via the `exit()` system call). Additionally, in order to correlate each task with an executable, Laplace logs each loading of an executable image (via the `exec()` system call).

Memory mapping events. We want to know which portions of each address space are shared at each moment. It may seem sufficient to emit each virtual-to-physical page mapping to discover shared pages. To see that this information is lacking, consider two virtual address spaces, A and B. Let a page from a particular file be mapped into A at address x, and let that same file page be mapped into B at address y. If x and y are both mapped to physical address k at the same time then it will be possible to determine that these two virtual pages are shared. However, from page table information alone we cannot correlate these pages with the specific file page that is the basis for their shared space.

Worse, consider that x may map to k at one moment, and that the mapping may be removed so that k can be used to cache some other page. Later, y could be mapped to physical page j. Although the same underlying file page is being mapped into both address spaces, the two processes never map their virtual pages to the same physical page at the same time, making it impossible to recognize the sharing from virtual-to-physical mappings alone.

To obtain more complete shared space information, we must log the events in the kernel that establish memory mappings. We will show, in Section 3.3, how Laplace uses this information to reconstruct accurately each shared mapping. First, we will examine the ways in which Linux establishes new memory mappings:

1. **File mapping:** The `mmap()` system call maps a portion of a file into a virtual address space. If more than one process maps the same portion of the same file, that space becomes shared. Laplace logs any call to `mmap()`, identifying both the virtual space and the file space being mapped. It also logs calls to `munmap()` in order record the removal of these mappings.

2. **IPC shared memory:** A *System V Inter-Process Communication (IPC) shared memory segment* can be mapped into a virtual address space just as a file can. If two address spaces map the same segment, then that mapped space is shared. Laplace logs each use of the system calls `shmat()` and `shmdt()`, which respectively establish and destroy IPC shared segment mappings. Both the virtual space and the shared memory segment are logged.

3. **Copy-on-write (COW):** When the `fork()` system call is used, a new virtual address space created for the child process, and it is made to share the pages of the parent's address space. These pages become *read-only*, and any attempt to write to them will cause a page fault, at which time the VM system will create a new, unshared copy of the page for the faulting process. Laplace logs the duplication of the parent's address space during a `fork()` call. Furthermore, Laplace will log any page fault that is handled as via the COW mechanism to record that a page is no longer shared.

4. **Anonymous page allocation:** The `brk()` system call, typically called by allocator functions such as `malloc()` or `new`, creates new mappings to *anonymous* pages—ones that are not associated with a file. An anonymous page is not initially shared, but a `fork()` system call can cause it to be shared, as described above.

Filesystem cache events. Laplace logs the allocation and deallocation of pages for filesystem caching. Since only the kernel is allowed to access a filesystem cache page, Laplace needs only to emit the identity of the file, the offset into that file, and the virtual page in the kernel's address space being used to cache the file page. Therefore, references by the kernel to that virtual page can be correctly associated with the underlying file page. When the page is reclaimed, the virtual address of the page is logged to indicate that it no longer holds a file page.

System call events. Other system calls may be valuable to the simulation of a multiprogrammed workload. Laplace can easily be made to log use of any system call, recording any relevant kernel information such as the arguments passed by the process. This capability can be used, for example, to note that a blocking system call, such a synchronous I/O request, has been invoked by a process. Such information makes it possible to simulate more accurately the scheduling of threads.

The kernel/simulator interface. It is undesirable for the kernel to store its trace records by performing normal I/O operations to a filesystem that it has mounted. I/O operations will be slow for a kernel running on a simulated processor. Furthermore, the memory references performed by these I/O operations would be captured by the machine simulator component of Laplace and logged in the raw reference trace, thus introducing self-tracing distortion.

To avoid these problems, the kernel performs its logging by passing information to the machine simulator which, in turn, stores the kernel trace record. Both the modified kernel and the machine simulator define, as part of their code, a simple `kernel_trace_record` structure that contains the fields necessary for representing a kernel trace record. The kernel is compiled with a single, statically allocated instance of this structure. The processor simulator is then given the address at which this structure will be loaded within the kernel.

When the kernel needs to log an event, it fills the fields of this structure with the appropriate information, including a *tag* that indicates the type of event. The kernel then references a *trigger field*—one whose address can be monitored by the machine simulator as a signal to store the kernel trace record currently reflected in the structure.

Thanks to this interface, modifying a kernel for use with Laplace is simplified. The kernel needs only to store values into the fields of a structure, and no interaction with the kernel's own I/O functions is needed. Furthermore, the I/O tasks are performed quickly by the underlying machine simulator which itself is running on real hardware. Also, because the location and size of the structure is known to the machine simulator, it can ignore references to that structure as it generates the raw reference trace. In this way, the memory dilation of Laplace is reduced only to the displacement of other data structures in the kernel by the `trace_data_structure`. As a final benefit, a modified Laplace kernel will run on a normal processor; its operations on the `trace_data_structure` are simply ineffectual.

Implementation complexity. The changes to the kernel are modest. None of the original lines of code were altered or removed, and only 317 lines of new code were added. 100 of these lines composed a new header file that contained a description of the shared structure used to communicate with the underlying CPU simulator. The remaining 217 lines are distributed across 20 files. These lines are placed where the critical kernel events are logged, assigning values into the aforementioned structure and referencing the trigger field.

While these lines of code are specific to our chosen kernel, the modifications to other kernels should be equally simple and small. The modified kernel can be run on a normal CPU (or an unmodified CPU simulator), where the uses of the kernel event structure will simply be ignored.

3.3 Post-processor

The raw reference trace and kernel trace must be interleaved in order to reconstruct the state of the processes, threads, and memory mappings. Each record in both traces contains a timestamp that is taken from the processor's cycle counter, thus making such an interleaving possible. As the state is reconstructed, selected events can be emitted into a new trace. We have implemented a *post-processor* that performs the reconstruction and can be extended to output any desired final trace format. We will describe here how the state is reconstructed and how extension can be used to generate new trace formats with selected information.

3.3.1 Canonical pages

Before describing the tracking of shared memory spaces, we must introduce the concept of a *canonical* page [16].[4] If we wished to ascribe an identity to a page,[5] we would *not* use the physical page number as an identifier, as that identifier could change over time. Instead, we would choose to use the virtual page number as an identifier since that value remains constant irrespective of its mapping to a physical page.

However, even a virtual page number is not necessarily a persistent identifier of a page. Consider mapping a page from a file to some virtual page number x. Then consider unmapping x and then mapping the same page from the same file to virtual page number y. That particular page, taken from a file, cannot be persistently identified by a virtual page number. Instead, we must identify the page by the file offset of that page within its file. We consider the file page to be a *canonical* page whose identity persists irrespective of its mappings to virtual pages. The same assertion can be made for IPC shared memory segments. By recognizing two virtual mappings to the same canonical page, we identify instances of shared memory.

[4]We have chosen not to follow Wilson in referring to these pages as *logical* pages because we believe that the term "logical" is overloaded and likely to be confused with the pages of a "logical" file.

[5]We are concerned with the page itself as a container for data, not with the data stored on that page at any given moment.

Canonical page identification. Although we were able to use the file page identity as a canonical page identity in the example above, some canonical pages do not have such readily usable identifier. Anonymous pages are not associated with a file, but may be shared in a copy-on-write fashion due to a `fork()` system call. The virtual page number is insufficient as an identifier because other anonymous pages in other virtual address spaces may use the same virtual page number.

The post-processor, therefore, must be able to assign identities to each canonical page represented in the raw reference and kernel traces. It maintains a single, large, *canonical address space.* As we can see in Figure 2, when a new canonical page of any type is encountered, it is given an identity by allocating it canonical address space. In this way, all canonical pages can be identified by a single, scalar, canonical page number, irrespective of the mechanism used to create that canonical page at execution time.

3.3.2 Reconstruction of system state

We will now examine how the post-processor tracks the mapping of virtual pages to canonical pages so that is may recognize all instances of shared memory, and how it associates each reference with a task.

Processes, threads, and tasks. Each task is equivalent to a kernel-level thread, and is assigned a virtual address space that can be identified by the address of its page table. When the kernel switches between tasks, it re-assigns a register that contains this address, thereby switching to the virtual address space used by the newly scheduled task. At the kernel level, Laplace logs each assignment of an address space to a task. At the processor level, Laplace logs the address space identifier along with each reference. The post-processor correlates tasks with address spaces, thus determining which tasks compose a process and attributing each reference to its task.

Virtual and canonical page tracking. The post-processor tracks the correlation between virtual and canonical pages. In Section 3.2, we listed the mechanisms by which new virtual mappings are established. Given the information gathered by the modified kernel, the post-processor can reconstructing all types of mappings: memory mapped, IPC shared memory segments, fork-induced copy-on-write, and anonymous pages, as shown in Figure 2.

By simply maintaining and using these mappings, the post-processor can emit canonical page identifiers in a final, post-processed reference trace. Simulators that consume such a trace can therefore identify that two virtual pages are shared, irrespective of the mechanism that caused the sharing, and handle references to those virtual pages accordingly.

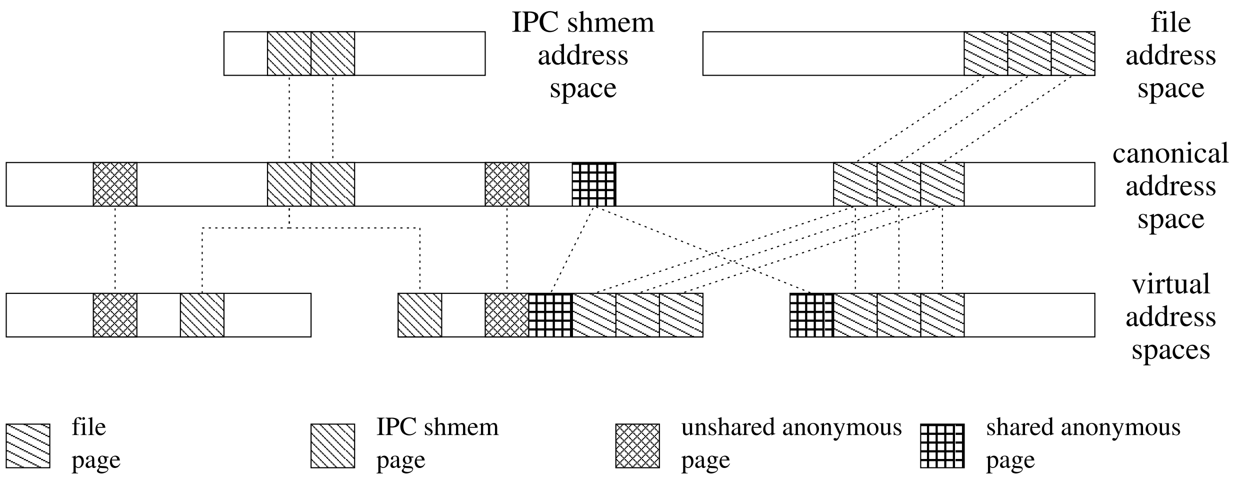


Figure 2: Each canonical page is associated either with a file page, an IPC shared memory page, or an anonymous page (which can be shared or unshared). Every virtual page must be mapped to some canonical page, and the post-processor uses that canonical page to determine the type of the page and its shared status.

The Linux buffer cache. There are two filesystem caches in Linux: the *page cache*, which corresponds to the caching of file pages that we have thus far referred to as *the filesystem cache*, and there is the *buffer cache*, which caches disk blocks. The difference is that, in the buffer cache, a page may contain some number of disk blocks that do not store data for the same file. Loosely speaking, buffer cache blocks are used to store and organize lower-level data for filesystem operations. Since standard `read()` and `write()` system calls operate on file pages and not directly on disk blocks, we have focused more on the use of the page cache.

Laplace does log the mapping and unmapping of pages in the kernel's address space used for the buffer cache. However, Laplace does not currently record the identity of the data stored into these buffers. It is therefore possible for the post-processor to determine the *amount* of space allocated to the buffer cache at each moment, but it cannot correlate the cached disk blocks with their canonical space. While it would be possible for Laplace to emit sufficient information for correlating disk blocks in the buffer cache with canonical space, it would require additional complexity that we have chosen not to implement.

Kernel space and references. There is one exception to the usual independence of virtual address spaces, and that is the kernel's virtual address space. In Linux, an upper portion of each virtual address space is reserved for kernel use, and that upper portion is shared across *every* virtual address space. Consequently, the post-processor must specially handle all references to addresses in this range, identifying them as references to the same virtual space irrespective of the currently scheduled task.

3.3.3 Output, simulation, and extension

The core of the post-processor performs the interleaving and reconstruction described above. However, it does not emit a new, merged trace, nor does it perform any kind of memory management simulation. Instead, it is designed to be extended so as to emit different reference trace formats, where a format may use only a desired subset of the available information at each step. To make clear how these final formats can differ, we provide two examples, both of which we have implemented.

The Global example. One possible output format for the post-processor that we have implemented is designed to strip away the per-task information, emitting only a sequence of referenced page numbers that could then be processed with a simple page replacement simulator as though the trace were derived from a single process. Such a global reference trace would be desirable for use with existing replacement policy simulators intended for uniprogrammed traces.

When emitting the Global format, the post-processor simply resolves each reference to its canonical page number and emits that value. Thus, the final reference trace contains only references to canonical pages, thereby allowing a simulation based on such a trace to reflect the management of shared pages and filesystem cache pages while ignoring scheduling issues. This type of simulation may be useful to gather quickly information about the general referencing characteristics of a multiprogrammed workload.

The Per-Process example. Another possible output format separates the original traces on a per-process and per-thread basis. For each process represented in the original traces, the post-processor will create a folder. For each thread of a given process, the post-processor will produce a new trace file that is located in the folder of its process. That file will contain all references that occurred for that thread, as well as all system calls performed by that thread. For each reference, the virtual and canonical page numbers are provided, thus enabling a simulator to recognize sharing will still tracking spatial locality.

By performing this separation into processes threads, it becomes possible to simulate the management of the entire workload. Such a simulator would be able to implement a simulated scheduler, interleaving the referencing behavior of the threads in way that reflects the interaction between virtual memory management and scheduling. This type of simulation was one of our primary motivations for developing Laplace.

4. PERFORMANCE

Any trace collector will incur substantial overhead because it must interfere at some level with the normal processing of a workload. The total overhead of trace collection can be divided into two components: *execution overhead* is the penalty incurred for running a workload within the modified environment of the trace collector, and *reference handling overhead* is caused by the processing performed on each reference captured by the collector. Previous work on trace collection has focused on the execution overhead, but we have found that the reference handling overhead dominates the execution overhead. In this section, we will show that Laplace's higher execution overhead compared to existing trace collectors is irrelevant, as their performance would all be constrained by the reference handling overhead.

We executed a few different programs to measure the execution overhead of Laplace. These programs were taken from the SPEC2000 benchmark suite, and chosen because they perform substantially different tasks. Indeed, the overheads show large variance because Bochs' simulation of some system components is more efficient than others. We ran these programs on a 1 GHz Pentium III. We then ran the same programs on Bochs, which in turn was running on the same 1 GHz Pentium III system.

As we see in Table 1, unmodified Bochs executes a mean of 227 times slower than the machine on which it runs. This value represents the capturing overhead for Bochs because the simulated system has control at the moment of every reference.

We also see two examples of trivial reference handling strategies. When Bochs is made to capture references but *not emit them at all*, the slowdown increases to a mean of 285-fold, which is an additional 25% handling overhead. If the captured references are emitted to `/dev/null`, the mean slowdown becomes 342-fold, which is 51% handling overhead. Note that Laplace was implemented to perform this type of handling efficiently: references are buffered and periodically emitted as binary values that require no formatting. Even these exceedingly simple forms of handling contribute substantially to the total trace collection overhead.

Reference production rates. Given the collection overhead of a 227-fold slowdown for Laplace, and given our test system using a 1 GHz Intel Pentium III CPU, the traced workload is processed at a rate of approximately 4.4 million instructions per second (MIPS). The fastest CPU currently available is approximately 3.5 times faster than this CPU, and so Laplace could run at about 15.4 MIPS with no modification.

Since Laplace collects each instruction fetch, then each instruction incurs at least one memory reference. As a rule of thumb, approximately one in four instructions is a load or store operation. Therefore, a mean of 1.25 memory references occurs per instruction. Since each reference implies one trace record to store the event, there are approximately 1.25 trace records per instruction. By multiplying the MIPS rate by this factor, we find that our test system could capture 5.5 million records per second (MRPS), and a fast system could produce 19.25 MRPS.

Streaming to disk. First we consider streaming the reference trace directly to disk. Each trace record will contain information on the type of reference, the number of bytes referenced, the virtual address referenced, and a timestamp. In our currently implementation, all of this information requires 17 bytes per trace record.

A fast disk can write at a rate of approximately 30 MB/s. By dividing this rate by the 17 bytes in each trace record, we find such a disk can store approximately 1.76 MRPS, which is 3.24 times more slowly than Laplace on our test system can capture them. Put differently, we would need a 93.5 MB/s disk to keep up with our test system, and a 327.25 MB/s disk to keep up with a fast system.

It is possible to use a higher-performance storage equipment in order to overcome this limitation. A fast RAID device is capable of storing more than 500 MB/s, and thus is capable of storing trace records faster than Laplace can produce them. However, given the growing disparity in CPU and disk speeds, the ability of even a RAID device to match the production rate of Laplace will not last long.

Streaming the raw reference trace directly to disk will consume storage space rapidly. Since our test system can produce records at 93.5 MB/s, even a 100 GB disk space will be filled in approximately 1,070 seconds of trace collection time. Given that this simulated system is processing at a 227-fold slowdown, then that storage corresponds to approximately 4.71 seconds of real execution. Therefore, direct storage to disk is not likely to be desirable for many contexts.

Compression and reduction. Next, we consider handling the captured references by compressing or filtering them. Because reference traces grow large quickly, it is typical to compress them with text compression utilities or filter them with custom tools designed for reference traces. However, any such processing itself has bandwidth limitations.

For example, consider the LZ77 compression algorithm used in the popular `gzip` utility. On our 1 GHz Pentium III machine, this compressor is capable of processing at approximately 7.7 MB/s, or 0.45 MRPS, which is more than an order of magnitude slower than reference capturing. Even the LZO compression algorithm [10], which is designed for speed, was capable of only 29.2 MB/s, or 1.72 MRPS, on our system. Although faster processors will improve the performance of these compressors, those faster processors would also improve the performance of the trace collector.

More complex trace filtering and compression techniques such as SAD and OLR[6], stack deletion [13], and difference encoding [4, 12] perform more complex analysis of their inputs and thus are unlikely to outperform LZO. We have used an implementation of SAD in conjunction with Laplace to enable us to store longer running executions, but it causes to total overhead to exceed 3,000-fold.

Online simulation. Trace collectors are sometimes connected to simulators so that the simulation can be performed online, thus avoiding the need to store the reference trace at all. Unfortunately, simulators also have bandwidth limitations. Even a simple LRU simulator performs more complex calculations than a fast text compressor, and more complex simulations will execute even more slowly.

Because the reference handling overhead is the limiting factor in trace collection speed, the tools addressed in Section 2 that exhibit lower execution overheads could not col-

Application	**Bochs slowdown**	**Laplace/Bochs no output slowdown**	**Laplace/Bochs into `/dev/null` slowdown**
equake	195x	275x	377x
gcc	85x	136x	157x
gzip	402x	444x	491x
Mean	227x	285x	342x

Table 1: The factor slowdown when compared to direct hardware execution when using unmodified Bochs and Laplace-modified Bochs emitting references to `/dev/null`.

lect traces any faster than the current implementation of Laplace, except when very fast RAID storage devices are used. Reference handling overhead is relevant whether the trace is stored or not, as any non-trivial computation on the references is likely to become a bottleneck.

5. CONCLUSIONS

Laplace is a reference trace collector that is capable of logging memory references for whole, multiprogrammed, multithreaded systems, including the kernel, all processes, and all kernel-level threads. It also logs kernel-level events sufficient to reconstruct memory mappings, process or thread schedulings, and uses of system calls. This information is sufficient for simulation of a workload where the threads and processes may be scheduled differently in the simulated system than they were in the original system. Furthermore, the information allows such a simulation to identify and to track accurately the use of shared memory and the filesystem cache. Laplace has been implemented and is available for download.

Because Laplace collects memory references at the level of a simulated CPU, applications require no modification to be traced. Any application, including those that dynamically load libraries and those that dynamically compile and or self-modify their code at runtime can be traced. The referencing information collected is detailed and complete.

These capabilities will make it possible to simulate the management of complete reference sequences observed by systems. With the detailed information of Laplace traces, it will be possible to simulate different memory management strategies without ignoring some critical, competing consumers of memory, such as multiple processes, the filesystem cache, and shared libraries. Furthermore, it will be possible to simulate the interaction between memory management and CPU scheduling, as Laplace traces contain information about blocking I/O operations, making it possible to simulate different schedulings and their consequences.

Laplace is only slightly tied to a given instruction set architecture. The modifications to a given processor simulator are simple, and could easily be ported to another such simulator. Laplace is somewhat more dependent on a given OS kernel, since it must record and reconstruct the manner in which a kernel assigns address spaces, shares memory, allocates filesystem cache pages, and creates, schedules, and destroys processes and threads. However, these features are common to all kernels, and should require only modest work to port.

We observe that the substantial impact of reference handling on performance is in some sense disappointing, as it seems to imply an undesirable overhead on *any* trace collection method. However, a reference trace needs to be collected only once, and is then likely to be reused for years and multiple experiments. Therefore, the high overhead is tolerable for some because of the value of the results.

Future work. Because we found that trace collection speed is most limited by the processing of collected references, we believe that further study on trace reduction and data compression will be valuable. There has been a good deal of work on compression techniques that allow for fast *decompression*, and some work [5] that has stressed the need for equally fast compression and decompression. Here we observe a rare instance in which extremely fast but effective *compression* is needed, and slower decompression is permissible.

The inherently high overhead of reference trace collection highlighted by this paper may also suggest a method for avoiding that overhead. Since reference handling overhead limits performance, faster trace collection must be achieved by *capturing fewer references*. We believe that it is possible, by modifying an OS kernel's virtual memory manager, to have a trace collector that is invoked only for some references. While this kind of approach would introduce a kind of inaccuracy that is unacceptable for some types of simulation (such as hardware cache simulation), it may be quite appropriate for other types. Specifically, trace reduction methods such as SAD and OLR [6] rely on the observation the virtual memory management policies do not need information about references to the most recently used pages. We believe a trace collector can be built that will never capture any of these references at all, thus allowing the system to run more quickly than Laplace.

For Laplace itself, we would like to port it to a processor simulator like Simics, as that package is capable of simulating a number of different instruction set architectures. We would also like to port Laplace to different kernels. While we believe that it will be simple to port Laplace to common UNIX-style kernels such as the BSD variants, we also believe that it should be ported to a Windows kernel. Such porting would allow for trace collection and study of some of the world's most used workloads.

Availability. The current implementation of Laplace can be obtained at `<http://www.cs.amherst.edu/~sfkaplan/research/laplace>`. That site includes our collection of Laplace traces as well as a number of utilities for trace reduction during reference handling.

6. REFERENCES

[1] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: a new technique for capturing address traces using microcode. In *Proceedings of the 13th annual international symposium on Computer architecture*, pages 119–127. IEEE Computer Society Press, 1986.

[2] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137. ACM Press, 1994.

[3] X. Huang, J. B.Moss, K. S. Mckinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin Department of Computer Science, February 2003.

[4] E. E. Johnson and J. Ha. PDATS: Lossless address trace compression for reducing file size and access time. In *Proceedings, IEEE International Conference on Computers and Communications*, pages 213–219, 1994.

[5] S. F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, Aug. 1999.

[6] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Trace reduction for virtual memory simulations. In *1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 47–58. ACM Press, June 1999.

[7] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software—Practice & Experience*, 24(2):197–218, 1994.

[8] K. Lawton, B. Denney, and G. Alexander. Bochs: An `ia32` machine simulator. `http://bochs.sourceforge.net`.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[10] M. F. X. J. Oberhumer. LZO real-time data compression library, 1997.

[11] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and optimzation of Win32/Intel executables using etch. In *Proceedings of The USENIX Windows NT Workshop*, pages 1–8, Aug. 1997.

[12] A. D. Samples. Mache: No-loss trace compaction. In *ACM SIGMETRICS*, pages 89–97, May 1989.

[13] A. J. Smith. Two methods for the efficient analysis of address trace data. *IEEE Transactions on Software Engineering*, SE-3(1), Jan. 1977.

[14] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, pages 196–205. ACM Press, 1994.

[15] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *Computing Surveys*, 29(2):128–170, 1997.

[16] P. R. Wilson. The GNU/Linux 2.2 virtual memory system. `http://mail.nl.linux.org/linux-mm/1999-01/msg00027.html`.