Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels

by

Ariel Tamches

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin—Madison

May, 2001

Acknowledgments

During the course of my graduate career, I have been fortunate to receive advice, support, and encouragement from many people. Foremost is the debt of gratitude that I owe my thesis advisor Bart Miller. Not only was Bart responsible for introducing me to the Paradyn project, he also suggested the course of my dissertation by casually mentioning one day that it might be “fun” to investigate dynamic kernel instrumentation. Bart has a well-earned reputation for dedication to his graduate students, and without his constant encouragement, this dissertation could not have been written.

My other committee members (Marvin Solomon, Larry Landweber, Paul Barford, and Greg Moses) each offered time and effort to help improve my work. Special thanks is due to Professor Solomon for an outstanding (and incredibly detailed) review of every page in the dissertation.

I am grateful to the members of the KernInst and Paradyn projects, past and present, for many stimulating brainstorming sessions and hacking advice. In particular, Matt Cheyney provided invaluable (and impeccably timed) assistance in debugging kperfmon’s timer primitives, two days before the OSDI ‘99 conference submission deadline. He also helped brainstorm the tricky case of instrumenting tail calls. Alex Mirgorodskii ported KernInst to 64-bit platforms, and in the process greatly improved the quality of the software. Vic Zandy has begun the formidable task of an x86 KernInst port, greatly improving its control flow graph parsing and register analysis.

My thesis research has also benefitted from support from persons outside of the KernInst group. Pei Cao, Jussara Almeida, and Kevin Beach suggested that I try to measure and optimize the performance of the Squid proxy server, and provided the Wisconsin Proxy Benchmark. Mike Shapiro, Madhusudan Talluri, Bryan Cantrill, and Stephen Chessin have provided invaluable insight into the inner workings of the Solaris kernel. I would like to single out the outstanding assistance of Mike Shapiro, who in the past two years has graciously (and rapidly) responded to dozens of detailed questions concerning the core of the Solaris kernel. I would also like to thank members of Veritas Software, especially Randy Taylor, for actively showing an interest in KernInst, and for serving as its first users in the “real world”.

Above all, I would to thank my family for their love and support during the often trying times of completing the dissertation. Special thanks is due to my mother, who not only provided endless encouragement, but has inspired me through her victorious battle against cancer. Mom, this work is dedicated to you. I hope your boy has made you proud.

Abstract

Operating system kernel code is generally immutable. This trend is unfortunate, because a kernel that can insert (and later remove) code at run-time has many uses, including performance measurement, debugging, code coverage, run-time installation of patches, and run-time optimizations. The research in this dissertation investigates dynamic (run-time) kernel instrumentation and its applications in the areas of kernel profiling and code evolution. We have implemented dynamic kernel instrumentation, a kernel performance monitor, and a run-time kernel optimizer in a system called *KernInst*.

The first component of this dissertation investigates *fine-grained dynamic kernel instrumentation*, a technology to dynamically modify kernel code. We have designed two primitives for run-time kernel code modification, *splicing,* which inserts instrumentation code, and *code replacement,* which replaces a function's code. A part of the KernInst system called *kerninstd* implements fine-grained dynamic instrumentation for Solaris UltraSPARC kernels.

The second component of this dissertation is the collection of techniques and algorithms for using dynamic instrumentation to obtain kernel performance information. The first techniques are the design and implementation of efficient instrumentation code to obtain counts, elapsed times, and virtual times of kernel code. This dissertation also presents a means to effectively calculate an estimate of kernel control flow graph counts from basic block counts. These techniques and algorithms are embodied in a kernel performance tool called *kperfmon*. A case study describes how kperfmon helped to understand and improve the performance of a Web proxy server.

The final component of this dissertation introduces the concept of *evolving code* in a commodity operating system. An evolving kernel changes it code at run-time, in response to the measured environment. KernInst provides a technological infrastructure that enables commodity kernels to evolve. As a proof of concept, we describe an automated kernel run-time version of the code positioning I-cache optimization. We have applied run-time code positioning to the TCP read-side stream processing routine tcp_rput_data. Code positioning reduces the time that tcp_rput_data waits for I-cache misses by 35%, reduces its execution time by 17%, and improves its instructions per cycle by 36%.

Contents

List of Figures

Chapter 1

Introduction

Operating system kernel code is generally immutable. This trend is unfortunate, because a kernel that can insert (and later remove) code at run-time has many uses, including performance measurement, tracing, debugging, code coverage testing, security logging, run-time installation of patches, run-time optimizations, and process-specific resource management. This thesis research covers dynamic (run-time) kernel instrumentation, and its application in the areas of kernel profiling and code evolution. We have implemented dynamic kernel instrumentation, along with a kernel profiler and a run-time optimizer, in a system collectively called *KernInst*.

The first component of this dissertation, presented in Chapters 3 and 4, is a low-level technology called *fine-grained dynamic kernel instrumentation*. This technology provides two primitives for run-time kernel code modification: *splicing*, which inserts arbitrary instrumentation code at desired kernel locations; and *code replacement*, which atomically replaces a function’s code. A part of the KernInst system called *kerninstd* implements fine-grained dynamic instrumentation for Solaris 7 / UltraSPARC kernels.

A second component of this dissertation is the design and implementation of a run-time kernel performance profiler built using the dynamic instrumentation infrastructure, as discussed in Chapter 5. The tool, called *kperfmon*, demonstrates an application of fine-grained dynamic kernel instrumentation. More significantly, kperfmon is a powerful performance profiler, giving a user detailed performance information that can aid in the optimization of both kernel and user code. As a proof of this concept, Chapter 6 shows how kperfmon helped to understand and improve the performance of a Web proxy server.

The final component of this dissertation, presented in Chapter 8, introduces the concept of *evolving code* in a commodity operating system. An evolving system is one that can change its code during run-time, in response to the measured environment. An evolving system will identify a problem (such as poor performance or a security attack), formulate a solution (changing some existing code to address the problem), and install the solution at run-time. KernInst provides a technological infrastructure that enables commodity kernels to evolve. As a proof of concept, Chapter 8 also describes a prototype implementation of an application of this infrastructure, an automated kernel run-time instruction cache optimizer. This implementation is the first time that the code of a running kernel has been re-ordered, on-the-fly, for I-cache performance. The optimization benefitted from the results of Chapter 7, which show that control flow graph edge execution counts can usually be derived from basic block execution counts, deferring the need to implement a technically more complex edge splicing primitive.

We performed a case study of using run-time code positioning to improve the end-to-end performance of a Web fetch benchmark, by optimizing the key TCP read-side stream processing routine tcp_rput_data. The optimization reduces the time that tcp_rput_data waits for I-cache misses by about 35%, reduced its execution time by about 17%, and improved its IPC (instructions per cycle) by about 36%. The optimization of tcp_rput_data led to a 7% reduction in the benchmark's total run-time.

This dissertation's thesis statement is:

> It is possible to dynamically instrument an already-running commodity operating system kernel in a fine-grained manner; furthermore, this technology can be usefully applied to kernel performance measurement and run-time optimization.

The remainder of this chapter motivates fine-grained dynamic kernel instrumentation, and presents additional information on the core contributions of this dissertation.

1.1 Applications of Dynamic Kernel Instrumentation

Fine-grained dynamic kernel instrumentation is a powerful foundation on which a variety of tools can be built. As motivation, we describe some of these applications.

Kernel profilers can use dynamic instrumentation to insert code annotations that gather performance information. The contents of the inserted code snippet(s) depend on the desired performance metric being gathered. A simple count of the number of entries to particular a function or basic block can be obtained by inserting code that increments a counter at the appropriate location. Performance measurement code can also be inserted to accumulate events that occur in a particular function or basic block. For example, on entry to a function, instrumentation can read a counter representing the elapsed cycle count. On exit from that function, further instrumentation can re-read the counter and subtract it from the previous value to obtain the elapsed time (in cycles) for one execution of the code body; this value can be added to an accumulating total representing the total number of elapsed cycles that the kernel spends in this routine. More detail on using dynamic kernel instrumentation for performance measurement will be discussed in Section 1.3.

Kernel debuggers can be implemented using fine-grained dynamic instrumentation. For example, breakpoints can be inserted at any machine code instruction by splicing in code snippets that display some kernel state to the console and (optionally) pause the executing thread and informs the debugger. Breakpoints can be conditionalized by adding a desired test.

Dynamic instrumentation can also be used for kernel tracing. At run-time, logging code can be spliced at desired tracing locations. Kernel code can be restored to its original state once the trace has been collected, so the overhead of tracing is incurred only when actually collecting a trace. Dynamic instrumentation contrasts with a static instrumentation system, such as a binary rewriter, which inserts code lasting for the entire run of the kernel.

Code coverage can be achieved using fine-grained dynamic kernel instrumentation. Coverage can be monitored by splicing in code that sets a flag (one per function, basic block, control flow graph edge, and/or call graph edge, as desired) to indicate that code has been reached. This instrumentation can be removed as soon as the flag is set so that the expense of code coverage actually *decreases* over time. Basic block and edge coverage are good examples of the utility of making instrumentation both fine-grained and dynamic.

Security checks, such as Department of Defense C2-level auditing [100] are another form of annotation that can be installed into a running kernel using dynamic instrumentation. Solaris can audit many events, including thread creation and deletion, file system path name lookups, file system vnode creation, successful and unsuccessful credential checks for super-user access, process forks, core dumps, stream device operations, file opens, closes and chdirs. However, auditing code is bracketed in the source code with #ifdef C2_AUDIT, which may not be enabled for a given kernel. Normally, turning it on requires changing the flag, recompiling the kernel, and rebooting. With dynamic instrumentation, an auditing package can be distributed as an independent kernel add-on and installed into a running system. Run-time insertion of auditing code requires a fine-grained splicing mechanism because auditing checks often occur in the middle of kernel functions.

Dynamic instrumentation enables automated run-time code optimization based on performance feedback gathered by dynamic profiling annotations. One example is automated run-time function specialization [74] based an input parameter. A function can be dynamically instrumented at its entry point to collect a histogram of values for the desired parameter, which is later examined for a frequent value. The instrumentation code is then removed, and a specialized version of the function's machine code is generated, with constant propagation applied to the specialized parameter. The original function then has the following code spliced at its entry: “if input parameter equals expected value then jump to specialized version; else, fall through.” A further optimization can sometimes remove the condition; call sites to the function can be analyzed for an actual parameter that always equals the specialized value. Where true, the call site is altered to directly call the optimized version of the function.

A second example of run-time code optimization that can be performed with dynamic instrumentation is code positioning, or moving seldom-executed basic blocks out of line to improve instruction cache performance [61, 69]. First, a function can be tested for poor instruction cache performance by inserting instrumentation that measures the number of icache misses incurred. After a time, the instrumentation is removed, and if instruction cache performance is poor, the function’s basic blocks are instrumented for execution frequency. A new version of the function with infrequently executed basic blocks moved out of line is then spliced into the kernel. As with the specialization example, the jump overhead can often be eliminated by redirecting call sites to the new function. This technique is explored in Chapter 8.

Dynamic kernel instrumentation may also be used to change kernel functionality, such as installing a process-specific version of a kernel resource management routine. Extensible operating systems [10, 11, 21, 35, 36, 47] have focused on this subject. It is worthwhile to note that dynamic instrumentation can provide a similar operation in a commodity kernel by splicing the following code at the appropriate kernel policy function: “If current-pid equals some-pid then jump to customized version, else fall through”.

Transparent data modification, such as encrypting or compressing a file system [39], can be implemented using dynamic instrumentation. At the entry point of the file system’s read and write routines, code can be inserted that changes that data stream appropriately.

Dynamic instrumentation can also be used to detect and adapt to security attacks. One example is a TCP denial-of-service attack, where a server is inundated with TCP SYN packets having randomly forged IP addresses. The number of packets can overflow the server’s listen queue, preventing legitimate clients from starting connections [18]. Annotations can be dynamically inserted into the appropriate kernel routines to detect a possible attack, such as an unusually high connection attempt frequency, all with different (possibly spoofed) IP addresses. Instrumentation can then adapt to the attack by temporarily ignoring all TCP connection requests except from a small, trusted core of IP addresses.

1.2 Fine-Grained Dynamic Kernel Instrumentation

The first question that this dissertation seeks to answer is whether fine-grained dynamic instrumentation of an already running, unmodified commodity operating system kernel is possible. Several technical challenges had to be overcome, but the answer is yes. These challenges include attaching to an already-running kernel, changing kernel code, decoupling code generation from splicing, and ensuring that splicing is not vulnerable to a race condition when kernel thread(s) are executing in the vicinity.

Fine-grained dynamic kernel instrumentation has been implemented in a component of KernInst called kerninstd, a privileged user-level process that runs on the machine whose kernel is being instrumented. With occasional assistance from a pseudo device driver /dev/kerninst, kerninstd is an instrumentation server for applications wishing to instrument the kernel. Kerninstd is decoupled from such instrumentation clients. The two do not need to execute on the same machine; they can communicate over a remote procedure call interface.

The primary kernel instrumentation feature of KernInst is *splicing*, the insertion of instrumentation code that performs a desired monitoring or adaptation task. This dissertation's research contributions in the area of instrumentation are:

- **Dynamic.** KernInst is loaded and instruments the kernel entirely at run-time, without any need to recompile, reboot, or even pause the kernel.
- **Fine-grained.** *Instrumentation points,* the locations where kernel code can be inserted, can be almost any machine code instruction within the kernel.
- **Runs on a commodity kernel.** This enables instrumentation under real-world workloads. It is worthwhile to note that much recent operating system research has taken place on custom kernels [10, 11, 21, 34, 35, 36, 37, 47, 59, 61, 62, 68, 73, 74, 75, 81, 82, 83, 84]; this dissertation shows that run-time instrumentation is feasible on a commodity kernel.
- **Runs on an unmodified kernel.** This contribution is important, because requiring a modified or somehow customized kernel, even an otherwise commodity one, would likely preclude an instrumentation tool's widespread use.

1.3 Kernel Performance Measurement

The second component of this dissertation is the design and development of a kernel performance measurement tool that uses dynamic instrumentation. The implementation, in a tool called kperfmon, is both a concrete demonstration of the utility of fine-grained dynamic kernel instrumentation, and a powerful performance tool, with several novel features. This section gives an overview of kperfmon and its research contributions; Chapter 5 contains the complete description.

1.3.1 Overview of Kperfmon

Kperfmon is a run-time kernel performance measurement system for the Solaris kernel running on an UltraSPARC processor. Its user interface is modeled in part from the Paradyn Performance Tools for user-level performance measurement [44]. The key concepts in kperfmon are *metrics* and *resources*. A metric is a time-varying measurement, such as elapsed time, processor time, or number of instruction cache misses. A resource is a component of the kernel's code, either a function or a basic block. Measurements in kperfmon are taken by combining a metric and a resource, which generates corresponding instrumentation code snippets, and the locations where they are to be inserted. Kperfmon then has kerninstd perform the actual instrumentation.

The major features of kperfmon, which will be explained in greater detail shortly, are:

- Kperfmon can measure an unmodified, already running, commodity kernel, without the need for kernel source code, because it uses kerninstd as its instrumentation engine.

- It directly measures performance; as such, it can be more accurate than tools based on statistical sampling [3, 40, 70, 93, 107].
- It provides a broad range of performance metrics based on any *event counter*.
- It can measure wall time events or virtual time events.

There are two major research contributions of kperfmon. First, kperfmon shows that a wide range of kernel performance measurements can be gathered via dynamic instrumentation of an unmodified commodity operating system kernel. Second, kperfmon is the first kernel profiling tool that can virtualize (exclude from the measurement the time that is spent context switched out) any wall time metric.

1.3.2 Explanation of Major Kperfmon Features

Instead of only being able to measure time spent in a desired piece of code (such as a function or basic block), kperfmon can, more generally, accumulate *events* occurring in that code. An event logically corresponds to the execution of some code, though many events are counted in hardware. Examples of hardware-counted events include the number of elapsed cycles, data cache misses, and cycles stalled on branch mispredictions. Examples of software-countable events include completed database transactions, retransmitted TCP packets, and failed file opens.

The general pattern of event accumulation instrumentation is illustrated in Figure 1.1. Given a metric and a code resource, instrumentation is generated for insertion at the code's entry and exit point(s). The entry instrumentation reads and remembers the underlying event counter value corresponding to the metric. The exit instrumentation re-reads the event counter and subtracts it from the value read on entry to obtain the number of events occurring during a particular invocation of the

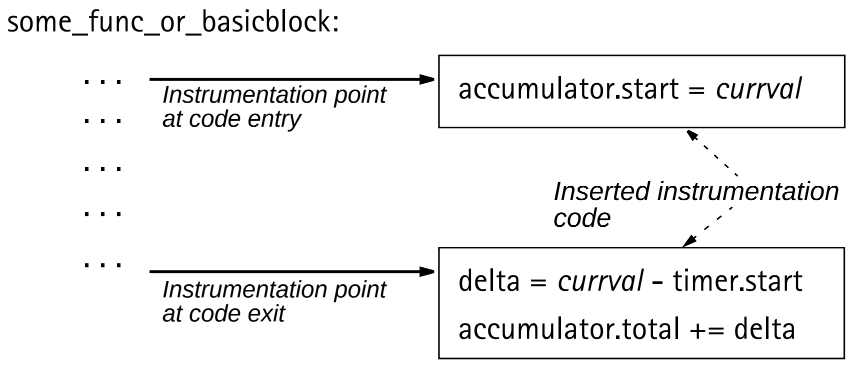

some_func_or_basicblock:
Instrumentation point at code entry
accumulator.start = currval
Inserted instrumentation code
Instrumentation point at code exit
delta = currval - timer.start
accumulator.total += delta

Figure 1.1: Interval Counter Accumulation Instrumentation
Curval is the current value of the metric's underlying event counter, such as elapsed cycles or number of cache misses. This diagram omits several details, such as making the instrumentation code thread-safe, that will be discussed in Chapter 5.

code. This value is then added to an accumulated total of the value for this given metric/resource combination. Because the entry and exit point(s) delimit an interval over which events are accumulated, metrics measured by this framework are *interval counter metrics.*

Kperfmon is extensible; it can create new performance metrics, given any software-readable, monotonically increasing event counter. New interval counter metrics are based on the instrumentation code template Figure 1.1. The only metric-specific part of the instrumentation is an appropriate implementation of code to read the current value of the underlying event counter.

The instrumentation of Figure 1.1 measures *wall time* events. Wall time metrics for a particular function (or basic block), as opposed to *virtual time* metrics, measure the elapsed number of events that occur between a thread's entry and exit to that code, including any events that may occur while a thread was context switched out. A virtual time metric, by contrast, measures only events that occur while a thread is actively running within that code, excluding any events occurring while context switched out. Wall time metrics are useful for measuring elapsed latency, such as response time or I/O events, and the time spent blocked on a synchronization object.

Kperfmon can also measure virtual time metrics, by dynamically instrumenting the kernel's context switch code. There are two parts to virtualization instrumentation. On context switch out, instrumentation stops those virtual accumulators that were started by the thread being switched out; on context switch in, all virtual accumulators that were stopped due to an earlier switch-out of this thread are restarted. In this manner, kperfmon can virtualize any wall time metric, a novel feature for a kernel performance profiler. This contribution is important because while it is relatively easy to create a wall time metric via instrumentation, it is hard to create a virtual time one. Virtualized metrics are useful when measuring events that only occur when the CPU is actively running the code being measured, such as cache events, branch mispredictions, and processor time.

The above instrumentation examples all yield *inclusive* metrics, which include events occurring while the code being measured is calling another function. An *exclusive* metric is one that only measures events occurring while the thread's program counter is within the function being measured. Kperfmon presently provides only inclusive metrics for two reasons. First, the instrumentation is simpler and has less perturbation; measuring an exclusive metric would require additional instrumentation before and after each procedure call, to stop and restart the event accumulation, respectively. Second, inclusive metrics help find performance bottlenecks more quickly; traversing the call graph with an inclusive time metric has been shown effective in automatically searching for bottlenecks in user code [14].

1.4 Evolving Kernels

The final contribution of this dissertation is a framework for *evolving kernels*. An evolving kernel is one that, automatically and at run-time, adapts and modifies its code in response to the run-time environment. Evolving kernels are motivated the the need for kernel up-time. Ideally, a kernel is ideally never taken off-line, or even momentarily paused, so modifications to its code should take place at run-time. Evolving kernels enable this 24x7x52 approach to operating systems

Perhaps the most obvious class of evolving kernel algorithms are those that perform some sort of run-time kernel optimization in response to measured performance. However, optimization is but one of many potential uses for an evolving system. Another use is to adapt to security attacks, such as a kernel that refuses new TCP connections except from a selected known and trusted group in response to a measured TCP SYN flooding denial-of-service attack.

There are three general phases to an evolving kernel framework. The first phase is measurement or detection, where data is gathered to identify a problem. The second phase creates a solution—improved code to be inserted into the kernel. The third phase installs this code into a running kernel. These steps are repeated, since the run-time environment is continuously changing. The first step is well suited to dynamic instrumentation, as demonstrated by kperfmon, as is the third step, requiring only a few additional low-level instrumentation features to augment code splicing.

This dissertation presents two contributions in the area of evolving kernels. The first contribution is an infrastructure for changing a commodity's kernel code at run-time. This infrastructure is general enough to allow arbitrary code modifications, including optimizations as well as security patches. In addition, it allows any code to be modified, not just system calls. The infrastructure is also generic enough to allow modifications to be created by outside sources. This independence is in keeping with KernInst's philosophy of decoupling code generation from the mechanism for inserting such code into a running kernel.

The second contribution is the design of a prototype run-time kernel optimizer aimed at improving instruction cache performance of kernel code. It uses Pettis and Hansen's code positioning optimization [69] that reorders basic blocks to improve instruction cache performance. Although the speedups achieved thus far are small compared to a user-level evolving system that implements a full range of compiler optimizations at run-time [5], the implementation demonstrates an infrastructure for run-time optimization of a commodity operating system kernel.

The primary research contribution made by this dissertation is to demonstrate that an a running, unmodified, commodity kernel can be converted to an evolving kernel.

Chapter 2

Related Work

This chapter discusses work that is related to some portion of KernInst. The first research contribution of this dissertation is fine-grained dynamic instrumentation (Chapter 4). Related work in this respect is presented in Section 2.1. The second contribution is in run-time kernel performance measurement (Chapter 5); Section 2.2 discusses related work in this area. Work related to the third contribution, evolving kernels and run-time kernel optimizations (Chapter 8), is discussed in Section 2.3.

2.1 Kernel Instrumentation

To put related instrumentation work in context, we review the desirable features of a kernel instrumentation system. It should:

- Modify code dynamically (i.e., at run-time), since static modifications to a kernel require a reboot to take effect.
- Be able to insert arbitrary user-specified code.
- Be fine-grained, able to insert code at any desired kernel location.
- Be able to insert or remove code at any time, even if one or more threads are executing at or near the instrumentation location.

- Provide safety guarantees for the contents of instrumentation code.
- Run on a commodity operating system, enabling real-world workloads to be instrumented.
- Run on an unmodified kernel; it may be impractical to require a customized operating system.
- Insert (and remove) instrumentation quickly.
- Introduce little perturbation beyond the inherent expense of the instrumentation code being inserted.

Note that all the above goals are obtainable; in particular, the goal of inserting safe code precludes the ability to insert arbitrary user-generated code. KernInst's low-level instrumentation technology achieves all these goals except for code safety. The responsibility for code safety is delegated to tools that are built on top of kerninstd, since such tools generate the code being inserted [105, 106].

The following sections describe other systems that perform some form of instrumentation that can be compared to KernInst's. The focus is on kernel instrumentation, though user-level instrumentation tools are also discussed, since existing kernel instrumentation systems are few.

2.1.1 Extensibility in Research Kernels

Extensible operating systems such as SPIN [10, 11, 37, 68, 84], Exokernel [34, 35, 36, 47], VINO [81, 82, 83, 87], Cache Kernel [21], Scout [41, 59, 61, 62], and Synthetix [23, 24, 73, 74, 75] allow user-level processes to download code into the kernel, offering a coarse-grained means of changing kernel code dynamically.

SPIN allows applications to change the operating system's interface and implementation through *extensions*. SPIN extensions assign a handler routine to a specific *event* through a central dispatcher. An event is a message announcing a change in system state or a request for service [11]. Event handler routines are written in a type-safe language (Modula-3) and are downloaded into the kernel's address space. Any procedure exported from a Modula-3 interface is a potential SPIN event handler. Calling such a procedure raises that event. Although SPIN allows code to be downloaded into the kernel, code can only be installed along a chain of handlers for a given event, and thus only procedure-grained (not instruction-grained) "instrumentation" is possible. In addition, SPIN is a customized kernel, which precludes running some real-world programs. Furthermore, SPIN is coarse-grained; it can only replace routines that are exported in a Modula-3 module interface.

The VINO kernel has many similarities to SPIN; it provides extensibility by allowing processes to download code into the kernel. VINO's downloaded code is fault-isolated [102] to restrict memory access to addresses belonging to the process. In addition, downloaded code executes in a transaction environment. If a process misbehaves, such as grabbing a lock and going into an infinite loop, the kernel detects it and aborts the transaction. VINO allows C++ classes to specify methods that can be customized on a per-object basis [87], so not all code in the kernel can be customized. In addition, since it overrides object methods, VINO offers a limited number of instrumentation points, which are at whole-function granularity. Furthermore, the instrumentation sites are pre-coded to facilitate instrumentation. A C++ virtual function makes a call through a pointer, which VINO simply changes to point to the downloaded code.

The Synthetix system provides extensibility in a fundamentally different manner from SPIN and VINO. Instead of allowing process-based customization of kernel routines, Synthetix provides a single optimization (partial evaluation, or specialization) for commonly executed paths of certain kernel routines. Thus kernel routines are customized, but not directly under process control. Like SPIN and VINO, when Synthetix customizes the kernel, it customizes only entire routines. Synthetix customization is installed by simply changing the value of a function pointer (a level of indirection like VINO). Thus, routines that *might* be specialized must be pre-written to contain a level of indirection (call through a function pointer), incurring a slight performance penalty even when no customization at a given routine has taken place. Furthermore, requiring a special code sequence precludes general fine-grained (instruction-level) instrumentation, since it would be prohibitive to sprinkle calls through a function pointer throughout each kernel function.

The Scout operating system is communication-oriented, geared toward network devices such as video displays, cameras, and PDAs. Scout is meant to be configurable to the device on which it is running. The key concept in Scout is the *path* [62], which abstracts a route through its software layers through which information flows within a machine. When a path is created, Scout uses global knowledge to transform and optimize communication along it. In providing such run-time optimizations, Scout is most closely related to Synthetix. By focusing its optimization toward specific kernel components, Scout does not instrumentation of arbitrary kernel code.

The Cache Kernel and Exokernel operating system architectures specify a minimal kernel that protects the underlying hardware resources (such as physical memory pages and disk blocks), while allowing processes to manage them. Because most conventional operating system functionality is transferred to user-level processes, there is little need to instrument the kernel. However, an exokernel architecture can still require that custom application policies be downloaded into the kernel, so the untrusted policy code can be inspected for safety [47] and access permissions [34, 36]. The trusted inspection must happen on code that has been downloaded into the kernel, because in user space, there are no assurances that code will not be changed after it has been certified as safe.

KernInst's instrumentation is preferable to the function pointer level of direction used by extensible kernels because it is more fine-grained, and because there is no overhead until instrumentation code is actually inserted into the kernel. Furthermore, fine-grained dynamic instrumentation can easily duplicate their extensibility techniques by inserting code that checks the current process ID into a kernel policy function. However, unlike extensible kernels, KernInst does not ensure that instrumentation code is safe; it assumes that a trusted entity is supplying the code to be inserted into the kernel. Instrumentation code may not have to be trusted in the future. Because kerninstd allows instrumentation code to be generated by outside sources, it can also leverage recent research into safety-checking of untrusted machine code [64, 65, 66, 102, 105]. Dynamic instrumentation is complementary to, and could be used with, research kernels to provide additional fine-grained splicing capabilities.

2.1.2 Extensibility in Commodity Kernels

Modern UNIX operating systems already provide limited forms of process-based extensibility, realized by re-engineering specific parts of the kernel. The classic example is the *vnode* interface [48, 79], which allows file systems to be added without recompiling the kernel or rebooting. SVR4-based kernels also provide a form of dynamic linking; kernel modules, such as device drivers, can be loaded at run-time [101]. Unfortunately, these forms of extensibility are limited. In the vnode example, extensibility is provided only for file system calls. Kernel dynamic linking operates at an even coarser grain; only entire modules may be loaded at run-time, and no module can be modified while in use.

The SLIC project [39] has investigated extensibility in commodity operating systems. SLIC re-routes events crossing certain kernel interfaces (system calls, signals, and virtual memory) to extensions that have either been downloaded into the kernel, or run in a user-level process. It interposes extensions on kernel interfaces by rewriting jump tables (the classic technique of a level of indirection achieved via a call through a function pointer) or through binary patching of kernel routines. Extensions perform some processing on the intercepted event and then either return to pass the event to its original destination. Although SLIC, like KernInst, runs on a commodity kernel (Solaris 2.5.1), it has several limitations. First, its call through a function pointer requires a well-defined kernel interface on which to intercept events. For example, intercepting system calls is done by changing the appropriate entry in the kernel variable **sysent[]**, which contains a table of pointers to routines. These events are coarse-grained, so the number of instrumentation points is on the order of dozens, not hundreds of thousands as with KernInst. SLIC's method of binary patching is similar in spirit to that used in KernInst and in Paradyn [43]. Unfortunately, it appears to be flawed; when binary patching, SLIC replaces several instructions with a jump to the extension, making it susceptible to a race condition when some kernel thread is executing within the code being modified (see Section 4.3).

2.1.3 Static Instrumentation

Static binary rewriters such as EEL [53, 54], Rational's Object Code Insertion [76], Pixie [88], ATOM [95], and Etch [78] provide fine-grained instrumentation of user programs (and potentially of kernels).

Static rewriting is performed on the executable file before the program is run. In the context of a kernel, a reboot is necessary before such instrumentation can take effect. This problem is magnified when instrumentation code is successively refined. It is often most efficient (when searching for a performance bottleneck, for example) to instrument only a few top-level routines at a time. If the results show a bottleneck in a function, then the instrumentation for it can be removed and applied to the function's callees. This strategy can be used to automate the search for bottlenecks [14, 42], with little run-time overhead, because at any given time only a few functions are instrumented. This algorithm is prohibitive with any static instrumentation system, because the program would have to be re-run every time the instrumentation changes. Static instrumentation systems typically instrument *en* *masse*; any instrumentation code that might be of interest must be instrumented, prior to execution.

2.1.4 Dynamic Instrumentation in User Programs

This section reviews related work in dynamic instrumentation in user-level programs, tracing the development of the earliest such work in debuggers and performance measurement tools.

The earliest published works on dynamic instrumentation can be found in several debuggers from the early to mid 1960s, including as DEC's DDT [27, 30] and Berkeley's IMP [52]. These systems envisioned limited uses for code splicing, primarily for conditional breakpoints and tracing.

A variation on code splicing overwrites an instruction with a trap, which transfers control to a tool (such as a debugger) that performs some action before continuing the process. Several steps are required when using traps. First, the tool will temporarily replace the trap with the original instruction. The instruction is then single-stepped, a feature assumed to exist on the processor. The single-step traps back to the controlling tool when done, at which time the tool puts the trap instruction back in place, and continues the application. Any debugger with this capability can serve as the foundation for a performance tool, since the inserted code can contain (for example) performance measurement primitives [12]. The primary limitation of this mechanism for instrumentation is its run-time expense. Traps are often expensive to process, and single-stepping after the trap is even more so. If the program is being instrumented and controlled by an outside tool, several context switches between the program and this tool will take place each time the instrumentation point is reached, increasing the run-time expense further.

KernInst has similarities to the dyninstAPI [13], a library that enables an application (the “mutator”) to dynamically instrument another process (the “mutatee”). The dyninstAPI originated by decoupling the instrumentation mechanism of Paradyn [43, 44] from its GUI. KernInst’s instrumentation differs from the dyninstAPI in several ways. First, KernInst instruments the kernel, while dyninstAPI only instruments user programs. Second, whereas dyninstAPI in certain cases enables fine-grained instrumentation (partly influenced by KernInst), its support is presently incomplete. In particular, on the SPARC platform, the dyninstAPI instruments by overwriting up to three instructions. As discussed in Section 4.3, overwriting more than one instruction is dangerous because instrumentation code can cross basic block boundaries. A second limitation of using more than one instruction is that it is unsafe when code is executing within the sequence being replaced. The dyninstAPI tolerates this condition by detecting it. It pauses the application and performs a stack trace of all threads; instrumentation is deferred if a hazard is detected. However, this technique is inherently limited to user-level instrumentation, since the kernel cannot be paused. A third difference between KernInst and dyninstAPI is in the area of structural analysis. Because KernInst knows the live registers at an instrumentation site, it is often possible to emit instrumentation code without saving and restoring any registers.

KernInst’s dynamic kernel instrumentation has its foundations in instrumentation for user code. As such, we have briefly reviewed the development of dynamic instrumentation in user code, from its origins in early debuggers, to more advanced uses in performance tools, and finally, recognition of the general-purpose nature of instrumentation in the creation of a library for run-time instrumentation of user code. As shown in Chapter 4, dynamic instrumentation cannot be trivially extended to the kernel; several technical obstacles must be overcome.

2.1.5 Dynamic Instrumentation in Kernels

This section reviews existing dynamic kernel instrumentation work. The most popular mechanism for kernel instrumentation today is trap-based, similar to the examples discussed above for user code. KernInst fulfills the need for a faster run-time splicing-based mechanism for an unmodified modern commodity operating system kernel.

The earliest known work on dynamic kernel instrumentation was in the kernel version of the DEC DDT debugger for TOPS-20 platforms, KDDT [85]. However, this work was never published and no studies of its use were ever published. KDDT, like DDT, was used primarily for conditional breakpoints and tracing.

An early tool called the Informer [28] provided a form of fine-grained kernel instrumentation, by allowing performance measurement code to be inserted into an operating system kernel. However, there are several key differences between KernInst and the Informer. First, Informer ran on a research platform, the Berkeley SDS-940, a 3-register machine with a 14-bit address space. Second, splicing in the Informer was simple because on the SDS-940, a single branch instruction could reach any range of its address space. In other words, the need for springboards did not arise (see Section 4.3). Third, instrumentation code in the Informer is always bracketed by code to save and restore all registers. While the cost was acceptable on the SDS-940, it would not be so on a modern platform. Thus, the Informer did not find a need to perform any structural analysis to determine register usage (see Section 3.3). Fourth, attaching the Informer to the SDS-940's kernel was facilitated by certain actions that privileged user code could perform; in particular, it could allocate directly from kernel memory by assuming a knowledge of the kernel's memory map. In other words, the Informer did not have to deal with bootstrapping onto a running kernel (see Section 3.2). According to one of its authors, the Informer was never ported to any other platform, and no case studies of its use were ever published [29].

The Solaris operating system contains a tool called lockstat [92] that instruments certain lock routines in the kernel, using a form of dynamic instrumentation. A user program, lockstat, communicates with a kernel driver, /dev/lockstat, which instruments the kernel by replacing a single instruction with two variations. In the first variation, the return instruction in a particular lock routine is replaced with a nop, which causes execution of that routine to continue past the return instruction to instrumentation code previously inserted after it. The instrumentation code will end by returning on behalf of the lock routine. The second variation of instrumentation replaces a return instruction with a branch to instrumentation code. This method is used when the instrumentation does not happen to fall immediately after the function, which in practice occurs when instrumentation code is shared between several routines.

The instrumentation technology used by lockstat is attractive because of its simplicity, but it is not as general as KernInst. The primary difference between lockstat and KernInst is that lockstat requires kernel modifications. Instrumentation code and labels identifying instrumentation sites are pre-compiled into the kernel. A second difference is that lockstat only instruments a small subset of kernel routines (13 instrumentation sites in Solaris 7 running on an UltraSparc). Third, there is a tight coupling between lockstat and the kernel's lock routines; they were developed in tandem. This coupling allows lockstat to avoid problems that arise in a general-purpose instrumentation system. These problems include bootstrapping the tool onto a running kernel (see Section 3.2), identifying the location of kernel code (see Section 3.3), and handling instrumentation code that is not located close enough to the instrumentation point to be reached with a single branch instruction (Section 4.3). In general, the splicing used by KernInst subsumes that done by the lockstat driver. However, lockstat can sometimes reach its instrumentation code a bit quicker than KernInst. When lockstat replaces a return instruction with a nop, only a single instruction of perturbation (the nop) is introduced, beyond the inherent expense of the instrumentation code.

Kitrace [51] is a tool for gathering kernel code traces. It replaces the instructions being traced with a trap instruction, which causes control to transfer to a custom handler. This handler appends an entry to the trace log and resumes execution. Because trap instructions can be inserted at most kernel instructions, Kitrace is fine-grained. However, KernInst differs from Kitrace in several respects. First, Kitrace does not instrument the kernel, in the sense that it does not attempt to insert code into the kernel. Second, resuming execution after a Kitrace trap is expensive. The trap instruction is temporarily replaced with the original instruction, a single-step is performed, and the trap instruction is then put back in place. KernInst's fine-grained dynamic instrumentation of any desired code subsumes Kitrace, which effectively provides insertion of trace-collecting code.

IBM's Dynamic Probes (DProbes) [60] allows coarse-grained breakpoints and tracing of both user and kernel code on an x86/Linux platform. Using essentially the same mechanism as Kitrace, DProbes overwrites a single instruction with a trap. When executed, this will transfer execution into a kernel handler, which identifies the specific instrumentation site (*probe point)* and executes the appropriate instrumentation code. The original instruction is then executed by temporarily restoring it, then single-stepping one instruction. Then, the trap instruction is put back. DProbes is an advance over Kitrace because the code executed by the trap handler can, to a limited extent, be user-specified. Unfortunately, general instrumentation cannot be achieved. Presently, a probe's instrumentation code (*probe program)* is written in a custom language that can specify only tracing and breakpoints. In addition, although not a limitation of the underlying technology, DProbes presently is not fine-grained since probe points can only be specified as symbolic names, so only the beginning of functions can be probed.

The single-step approach used by Kitrace and DProbes is unsafe on multiprocessors. When the original instruction is temporarily restored (for single-stepping purposes), a thread running on a different CPU that executes this code will execute the original sequence, thus missing the opportunity to trap and causing the corresponding instrumentation code to be skipped. This problem does not occur in the user-level variants of trap-based instrumentation discussed in the previous section, because the controlling tool is able to otherwise pause the application before replacing the trap instruction and single-stepping.

2.1.6 Instrumentation Summary

No system before KernInst can quickly instrument a running, modern commodity operating system, entirely at run-time, and at a fine grain. In fact, the closest related instrumentation work has been done for user programs, where fine-grained static (and, though somewhat less common, dynamic) instrumentation has been implemented in several research and industry tools.

2.2 Performance Measurement in Commodity Kernels

Chapter 5 presents the design and implementation of a kernel performance measurement tool, kperfmon, that is built on top of kerninstd. This section describes related work in the area of kernel performance measurement, divided into performance measurement done through sampling-only (Section 2.2.1) and performance measurement done via instrumentation (Section 2.2.2). As with the previous section, related work that operates with user programs is also discussed, because kernel performance measurement tools are few.

To put related performance measurement work into context, we review (from Chapter 1) the desirable features of a kernel measurement system. It should run on an unmodified operating system and entirely at run-time, providing precise, not approximate performance measurements, while introducing little perturbation into the system. It should have an extensible set of performance metrics based on event counting, with wall time and virtual time, and inclusive and exclusive variants. It should also be able to associate metrics with machine resources other than code, such as synchronization objects.

2.2.1 Sampling-Only Approaches to Kernel Performance Measurement

Some tools measure performance by periodically interrupting a program (or operating system kernel) to sample the present value of the program counter (PC) register. Each sample assigns the time since the last sample to the instruction corresponding to the sampled PC. Over a long period of time, the results are likely to yield an accurate profile of where the time is being spent in a program. This approach to performance measurement is attractive because it does not require instrumentation (a more complex engineering task, whether done for user programs or the kernel, and whether done statically or dynamically), and can have low perturbation that remains nearly constant. This general technique is applied to user-level code in prof [93] and gprof [40] (gprof uses some instrumentation to obtain exact edge counts in the program's call graph, but timing information is collected exclusively through sampling). Additionally, Compaq's Continuous Profiling system (dcpi) [3], Morph [107] for Alpha processors running on Compaq Alpha and NT platforms, and Intel's VTune Performance Analyzer for Windows [46] each use sampling to obtain both operating system and user-level code profiles. Both dcpi and VTune provide access to on-chip event counters (such as cache misses) to extend the set of performance metrics beyond simple timings. These tools can measure unmodified binaries and run on commodity operating systems with the following exceptions: gprof requires program recompilation and Morph runs on a modified version of Compaq UNIX operating system.

Although attractive for design simplicity, the sampling-only approach to performance measurement has fundamental limitations. The first difficulty with sampling is in accurately assigning events to instructions causing them. In dcpi, an event counter register is set to cause an overflow trap, whose handler has to identify the instruction that caused the trap and update its statistics accordingly. This identification is difficult on modern processors with imprecise interrupts (delivered several cycles after the instruction causing the interrupt has completed), and is impossible on processors whose imprecise interrupt delay is not fixed, such as the Alpha 21264 [31] and Pentium Pro [45]. A solution to this problem is presented in the ProfileMe project [25]. Periodically, ProfileMe chooses an instruction to profile. The CPU, with hardware support specially added to the Alpha 21264a chip [22], will gather detailed statistics on the instruction's execution through each stage of the pipeline, such as whether it hit in the I-cache or which data address it loaded or stored. When this instruction completes, a trap is generated so that software can read and process the statistics gathered for this instruction. Instrumentation code that reads hardware counters does not share this difficulty, because it does not rely on the (imprecise) interrupt mechanism.

A second fundamental limitation to sampling-only tools is that their timer metrics can only measure virtual events, not wall time events. This limitation precludes any metric that inherently includes events while blocked, such as I/O latency and mutex blocking time. Wall time cannot be measured purely by sampling the PC register because a thread will never have its program counter sampled while it is context switched out. If *all* threads are presently switched out, then a sample will be of a PC in the system's idle loop. However, if any one thread is actively running, its PC, and not that of any switched-out thread, will be sampled and have profile time assigned to it.

It is conceivable to measure wall events with a sampling-only approach, but only by doing significantly more processing during each sample. Specifically, during each sample, a stack back-trace of all switched out threads can be taken, with events assigned to the routine(s) that have blocked. The overhead of such an approach would be prohibitive, because accurate results dictates frequent sampling (5200 per second for dcpi [3], 1024 per second for Morph [107]), which mandates that very little work can be done on each sample to maintain acceptable perturbation. Even if the expense of a stack back-trace per blocked thread on each sample were not prohibitive, additional complications would arise. Back-traces are often incomplete in optimized code; a function B (called from A) that calls another function C as its last action can tear down its stack frame before calling C, causing C to return directly to A. (This tail-call-optimization and its implications in dynamic kernel instrumentation are discussed in Chapter 4). Interestingly, the instrumentation approach used by KernInst can easily measure wall time, but requires additional work to measure virtual time. However, while additional instrumentation makes virtual time measurement possible in KernInst (see Chapter 5), sampling tools cannot easily measure wall time.

A third limitation of sampling-only profilers is that while they can gather exclusive metrics with reasonable accuracy, they have difficulty measuring inclusive metrics. In particular, gprof only achieves inclusive virtual time by instrumenting the program to collect counts along the edges of the program's call graph, and assuming that every call to a function takes the same amount of time [40]. This dubious assumption can introduce unbounded errors into its reported results that cannot be mitigated through more frequent sampling [70]. Dcpi does not attempt to report inclusive time. If it did, results would be even more susceptible to error because unlike gprof, dcpi uses statistical sampling for both timings and counts. Thus, dcpi presently gives only a "flat profile", not attempting to include the time spent in a routine's callees.

A fourth limitation of sampling-only measurement is that metrics can only be associated with code resources, not data resources. For example, frequently it is useful to measure blocking time for a particular mutex lock object. A sampling tool cannot provide this metric for two reasons. First, this is inherently a wall time metric, which as discussed above cannot easily be measured by sampling. More fundamentally however, associating the time spent in a mutex routine with a particular mutex object requires being able to identify the mutex object at any time during the mutex entry routine. Unfortunately, due to compiler optimizations that re-use registers for different purposes throughout a routine, it may be impossible to sample the mutex object.

A fifth limitation of sampling-only approaches is that the set of available performance metrics is fixed, limited to metrics whose underlying value can readily be sampled, such as the on-chip counters for cache and branch mispredict misses, or pre-calculated counters in memory. In contrast, dynamic instrumentation allows KernInst to create new counters in software, which can then be sampled. Thus, the instrumentation and sampling-only approaches to performance measurement are essentially complementary.

A sixth limitation of sampling-only profilers is that, in order to take a periodic sample, the tools often require exclusive access to the system’s performance timers. For example, dcpi writes to hardware performance counters to cause an overflow after a certain amount of time, as the trigger for sampling. Unfortunately, this means that any tool also wishing to use these counters cannot be run simultaneously with dcpi.

In the final analysis, sampling profilers are well-suited to a limited class of performance measurements. For example, dcpi is well-suited for identifying processor stall cycles due to certain events (cache misses and branch mispredictions) down to individual instructions; such measurements are most useful on already-optimized programs that wish to examine their detailed pipeline behavior. Such fine-grained pipeline measurements have traditionally been the exclusive domain of simulators. Unfortunately, its limitations in measuring inclusive metrics, in measuring wall time, in associating metrics with non-code resources, and in creating new metrics prevent dcpi from being a general purpose profiler.

2.2.2 Instrumentation Approaches to Kernel Performance Measurement

The most straightforward means of measuring kernel performance is to add measurements directly into the kernel’s source code. This approach has several obvious disadvantages. First, the overhead of the instrumentation code is borne whether or not the data it collects is of interest. Second, changes require access to the kernel source code, which may not be available. Third, the set of measurements is fixed; adding or removing data collection requires kernel source code alteration, recompiling, and rebooting.

Despite the limitations, manual instrumentation is sufficiently popular (in its simplicity, primarily) that a package called kstats [18, 91] exists in the Solaris kernel for standardizing certain performance measurements, and access to them from user code. Similarly, another package called TNF [94] exists to standardize kernel trace collection through manual instrumentation. However, since this section is concerned with performance measurement, the discussion is limited to kstats.

The Solaris kstats facility collects counts and timings for hundreds of kernel events. The kernel’s source code contains calls to macros that increment a counter or keep track of times when certain events occur, such as writing a virtual memory page to disk, or TCP packet retransmission. These counters and timers can be read by user code using the kstat library. Although this library provides access to much performance information, the approach has several drawbacks. First, the overhead of instrumentation is incurred each time the event takes place, even if no performance tool is interested in the data. Second, kstats presently collects system-wide totals; the source code does not attempt to conditionalize kstat collection to a specific process or kernel thread, for example. Third, adding a new kstat counter or timer, or a new instrumentation point, requires recompilation of kernel code. Dynamic instrumentation can augment kstats because it can insert kstat counting code at desired kernel locations during run-time, with the advantage that instrumentation overhead is present only when a tool desired to collect that information. As shown in Chapter 5, dynamic instrumentation can also *virtualize* counters and timers by instrumenting the context switch routines to temporarily stop and restart the accumulation of events when a thread or process in question is switched out. Dynamic instrumentation can be used in concert with the existing kstats framework, by providing the necessary virtualizing framework.

2.2.3 Summary

Existing work in kernel performance measurement tends to fall under two categories. In the first, sampling is used to obtain performance information without kernel modification. However, such tools are not well suited for entire classes of performance measurements (wall and inclusive metrics), and can only build metrics using pre-existing event counters. On the flip side are tools for instrumenting the kernel, such as Solaris' kstats. This approach can create new event counters and measure wall time, but requires kernel source code modification. KernInst attempts to achieve the best of both worlds, allowing accurate performance instrumentation to be inserted into a stock commodity kernel, while providing the infrastructure for creating software event counters on which new metrics may be built.

2.3 Run-time Kernel Code Optimization and Evolving Kernel Code

This section discusses related work in run-time code optimization and evolving kernels. As with earlier sections, work performed at the user level is occasionally discussed, since comparatively little research into run-time kernel adaptation has been done to date.

2.3.1 Run-time Compiler-Like Optimizations in User Programs

Run-time code optimizations are a type of evolving algorithm. This section discusses existing work in this area, for user-level code.

Dynamo [5] is a user-level run-time optimization system for HP-UX programs running on PA-8000 workstations. To run under Dynamo, an otherwise unmodified binary is linked with the Dynamo shared library, and transfers control to Dynamo with a call to this library. Dynamo then interprets the program (without instrumenting it), collecting sequences of hot instructions (*traces*). At that time, Dynamo will inject an optimized version of the trace (a *fragment*) into a software cache. Then, when the code executes this trace, it will execute at full speed. Dynamo fragments can be interprocedural, spanning basic blocks and procedure boundaries. Should control exit the set of traces in the fragment cache, Dynamo resumes interpretation until it reaches an address known to exist in the fragment cache. Assuming that the program exhibits temporal code locality, Dynamo expects to reach a steady state where execution takes place almost entirely from the fragment cache.

Although similar in spirit to KernInst's evolving kernel framework, there are several differences between the underlying mechanisms used by KernInst and

Dynamo. First, Dynamo only runs on user-level code, whereas KernInst runs in the kernel, and could be extended to user code. It would be difficult to port Dynamo to an operating system kernel because Dynamo uses interpretation to find hot traces, rather than instrumenting or sampling. Taking control of the kernel at run-time and interpreting it would be more difficult than doing the same in user space. Even if it were possible, the overhead of kernel interpretation may be unacceptable because the entire system is affected by a slow kernel. By contrast, a slowed process usually has an isolated effect.

A second issue with Dynamo is code expansion. Because Dynamo inserts entire fragments into its software cache, the same basic block can appear multiple times. This has the advantage of enabling optimization on interprocedural path traces, but can result in a code explosion when the number of executed paths is high. It it possible that the amount of code is acceptable on HP-PA 8000, which has an unusually large (1 MB) L1 I-cache, but not on the UltraSparc-I or II, which have 16 KB L1 I-caches [96, 97]. The Dynamo authors note that interpretation increases the chances for code explosion, because higher interpretation overhead leads Dynamo to more aggressively speculate the contents of hot traces, so they may be optimized and put into the fragment cache quickly. Dynamo tolerates possible flooding of the fragment cache by periodically flushing the cache when it detects frequent additions to it, on the theory that under such circumstances, a new working set of code is being formed.

A third difference between the KernInst and Dynamo approaches to evolving code centers on generality. Dynamo uses interpretation until a hot trace is found, at which time native code is executed. This works well as long as code does not need to be interpreted for long before executing it natively, which is the case when using Dynamo for code optimizations. It would be less practical, however, to use Dynamo for other kinds of evolving code scenarios, such as detecting and adapting to a security attack. Under this example, Dynamo would interpret the system, with corresponding slowdown, until a security attack is detected, if ever. Thus, Dynamo cannot be used as the backbone of a general-purpose evolving code framework. Dynamo's evolving framework has several fundamental steps; one of which, measurement, is its weak link. More general-purpose evolving systems are discussed in Section 2.3.2.

Compaq's Wiggins/Redstone (W/R) dynamic optimization system is similar in spirit to Dynamo. It measures an application and emits optimized code, which is injected into a running program. W/R appears to remove Dynamo's weak link by obtaining hot paths through sampling (using a variant of dcpi [3]), without the need for interpretation. W/R is at this time unpublished, though a presentation with its key points exists [26]. Both Dynamo and W/R envision only a limited form of evolving code, the optimization of code through compiler transformations.

A project being developed at Microsoft Research called Mojo [16] is also similar in spirit to Dynamo, though it runs x86 programs, not PA-RISC programs. The other significant difference between Mojo and Dynamo is the means of executing "cold" blocks: Dynamo interprets cold code, while Mojo performs direct-execution. To simulate the effects of a basic block, Mojo uses a lightweight disassembler to identify the block's boundaries, alters its ending control flow instruction so that it returns control to the Mojo dispatcher, and is emitted into a buffer for direct execution. A software cache helps amortize the cost of the disassembly. As a whole, however, Mojo has not yet achieved the speedups of Dynamo, showing speedups on only two of the SPEC benchmarks [16].

2.3.2 Run-time Wholesale Changes in User Programs

The above section has given an overview of run-time changes by the application of compiler optimizations. While useful, such changes are less general than a system that allows more wholesale changes to running code, such as the installation of an entirely new version of an algorithm.

Run-time compilers [4, 55] and run-time code generators [33] are an excellent match for one step of the evolving framework: creation of new code. However, monitoring, deciding on new code, and installing that code, are left unaddressed in these projects. This is not necessarily a limitation; it may be desirable to be able to use independent products when building an evolving system—a fact made possible because the generation of code is orthogonal to the issue of injecting the code in a running system and monitoring it.

Berkeley's Dynamic Feedback project [32] presents a static compiler-based approach to evolving user code, allowing the best of several different algorithms to be measured and chosen at run-time. The most important feature of the Dynamic Feedback system is that it can choose between several radically different implementations, as opposed to an approach that only can tune one or more control variables of a single algorithm [20, 80], or one that only performs run-time compiler-type optimizations to an otherwise unmodified algorithm (like Dynamo and Wiggins / Redstone).

In Dynamic Feedback, the compiler generates several different versions of an algorithm from the same source code, each using a different optimization policy. This requires a compiler that knows different implementations of the algorithm. Dynamic Feedback's compiler automatically parallelizes C++ code, choosing from among three distinct synchronization algorithms. In Dynamic Feedback, *each* algorithm is generated. During run-time, the program alternately performs *sampling* and *production* phases. In the sampling phase, *each* version is executed for a fixed interval, and its performance measured through static instrumentation. The longer production phase then follows, using the version deemed fastest from the sampling phase. Changing algorithms at run-time is relatively simple: each iteration of the program's main loop switches on a variable that indicates the present algorithm. After the production phase has run for a fixed internal, the program goes back to the sampling phase, allowing the program to adapt to an environment change since the last sampling phase.

The infrastructure of Dynamic feedback forms a generic evolving user-code system because the algorithms that it can switch between can be arbitrarily distinct. However, there are several limitations compared to the infrastructure used in KernInst. In the measurement phase of an evolving system, Dynamic Feedback uses static instrumentation generated by the compiler, while KernInst does not require any compiler support. A second difference is in the installation phase of an evolving system. Dynamic Feedback chooses among the three algorithms by simply switching on a control variable. This level of indirection, with slight variations (such as a call through a function pointer), exists in the extensible kernels discussed in Section 2.1.1. The problems with these levels of indirection are the same: code must be built *a priori* with the level of indirection, a slight performance cost is incurred each time that code is reached, and the number of sites where installation of new code can take place is limited to places where the level of indirection exists.

2.3.3 Run-time Optimizations and Evolving Code in Research Kernels

The Synthesis kernel [56, 72] provided an early research example of kernel specialization. In Synthesis, a *creation* kernel call (such as a file `open`) generates and returns specialized code for *executive* kernel calls (such as file `read` and `write`), with future access through executive calls. Synthesis was purely a research kernel; for example, it did not support virtual memory. Also, Synthesis required that all specialization policies be built into the kernel (as pre-compiled *code templates* that await only the values of certain parameters to become executable code).

The extensible operating systems discussed in Section 2.1.1 and Section 2.1.2 allow process-specific policies to be installed into a running kernel, providing a limited form of code adaptation. Extensible kernels allow processes to generate new code, which is one step of an evolving system. They are also able to install this code, though with a number of limitations: only a few functions can be replaced, and only where a pre-existing level of indirection (typically a call through a function pointer) allows for easy redirection.

A more complete evolving framework is proposed for the VINO extensible kernel [82]. To make the kernel self-measuring, it envisions a combination of static compiler-generated performance instrumentation, hard-wired statistics gathering throughout the kernel, and run-time installed code to gather traces and logs of the performance data. Code built into the kernel would detect high resource utilization. If so, an off-line heuristic system would suggest an algorithmic change, to be examined by simulating its execution using inputs from the previously gathered traces and logs. If the algorithm is deemed superior to the current one for a given process, the extensible kernel would install it.

Although presently unimplemented, the framework as proposed is powerful. However, a key assumption, that a custom kernel is required for certain steps, is incorrect, because KernInst can perform them on a commodity kernel. These steps are installing measurement and trace-gathering code at run-time, simulating *in situ* a proposed new policy algorithm, and installing that algorithm in place of the existing one.

2.3.4 Run-time Optimizations and Evolving Code in Commodity Kernels

The Synthetix project [23, 24, 73, 74, 75] is a follow-up to the Synthesis project, performing specialization on a modified version of a commodity operating system, HP-UX. Synthetix allows the implementation of an algorithm (the `read` system call in particular) to be specialized on a per-file and per-process basis. The specialization can be *incremental*, meaning the level of specialization can change at run-time. The specialization can also be *optimistic*, allowing possibly invalid assumptions to boost the amount of specialization, provided extra code in the kernel called *guards* handle un-specializing the specialized code as soon as any of the optimistic assumptions become invalid. In a limited context of the read system call, Synthetix has shown that a customized, though otherwise commodity kernel, can adapt its code at run-time.

There are several differences from the framework used by Synthetix and that used by KernInst. First, Synthetix runs on a modified version of an operating system. Second, Synthetix requires that specialized code templates and guards be precompiled into the kernel. Third, Synthetix invokes the common technique of a pre-existing level of indirection, a call through a pointer, to be able to easily change implementations of a function. This means that a slight performance penalty is incurred whether or not specialized code has been activated. This also limits the number of points in the kernel that can be specialized (presently a single point in Synthetix, the read system call). Fourth, the synchronization algorithm that Synthetix uses to avoid a race condition when specializing code on a multiprocessor assumes that there is only a single thread per process.

Existing commercial kernels have limited, hard-wired examples of adaptive execution or evolving code. An example is in the Solaris kernel, which contains a hard-wired implementation of adaptive execution in its mutex locks called adaptive locks [57, 58]. An adaptive lock in Solaris will either spin or block when the lock is being held by another thread, depending on the holding thread's state. If the holding thread is presently running (on another processor), then the primitive will spin; otherwise, the primitive will block. This policy works well in the spin case assuming that locks are generally held for less time than the overhead of two kernel thread context switches, which are inherently incurred whenever waiting for and then acquiring a mutex lock through blocking.

2.3.5 Summary

This section has reviewed related work in the area of evolving systems. No system before KernInst provides the underlying mechanism that allows an unmodified, already-running, commodity operating system kernel to become an evolving one. Also unlike any existing work, KernInst uses one technology, fine-grained dynamic kernel instrumentation, for both measuring to identify a problem, and installing new code into the kernel. A unified technology is important, because kernel modification is challenging; performing difficult engineering mechanisms with a single technology can therefore be considered a qualitative advancement.

2.4 Related Work Summary

This chapter has presented an overview of related work in the areas of dynamic kernel instrumentation and two of its applications, performance profiling and run-time optimization. No system before KernInst provides the technological infrastructure for fine-grained dynamic instrumentation of an already-running commodity operating system kernel. The remaining chapters of this dissertation describe describes the design and implementation of this infrastructure (Chapter 4), followed by a manageable and coherent study of some of its many applications: performance profiling (Chapter 5 and Chapter 6), and evolving kernels and run-time kernel optimizations (Chapter 8).

Chapter 3

Introduction to KernInst

This chapter gives an overview of the KernInst system components (Section 3.1), and then describes the steps that kerninstd performs on startup. These steps are attaching to a running, unmodified Solaris kernel (Section 3.2) and performing a structural analysis of the kernel's machine code (Sections 3.3, 3.4, and 3.5).

The structural analysis information that kerninstd calculates contains the kernel's run-time symbol table (function names and their starting addresses in memory), a per-function control flow graph of basic blocks at the machine code level, a call graph, and information about live and dead registers at each instruction. A benefit of working with machine code is that the effect of compiler optimizations, which can reorder and remove code, are visible.

Kerninstd makes this structural analysis information available for instrumentation clients. Most applications will find kerninstd's structural analysis sufficient for their purposes, without requiring further access to kernel source code. For example, an application wanting to profile file system pathname-to-vnode translations in Solaris need only know that the relevant function is named lookuppn. (End users of such an application do not even need to know this much.) Also, a code positioning optimizer works entirely at the machine code level, without source code. However, some applications may benefit from kernel source code, which KernInst does not have access to, and cannot make available. Fortunately, such applications tend to be used by kernel developers, who already have access to kernel source code. For example, a kernel developer using KernInst to trace a specific source code line needs the compiler's debugging line number information to map the line number into an address.

3.1 KernInst Architecture

The structure of the KernInst system is shown in Figure 3.1. Applications that wish to instrument the kernel interact with *kerninstd*, a user-level daemon with super-user privileges. Kerninstd communicates with a small pseudo-device driver /dev/kerninst. This driver allocates the code patch and data heaps, which are used to hold instrumentation code and the data that it uses. Kerninstd maps both heaps into its address space via /dev/kmem, the standard means in UNIX systems for accessing kernel virtual memory from privileged user programs.

KernInst's instrumentation functionality is mostly within kerninstd; only when kerninstd needs to perform actions in the kernel's address space does it enlist the assistance of /dev/kerninst. Placing most functionality in kerninstd has two positive effects. First, it minimizes KernInst's presence in the kernel. Second, because the

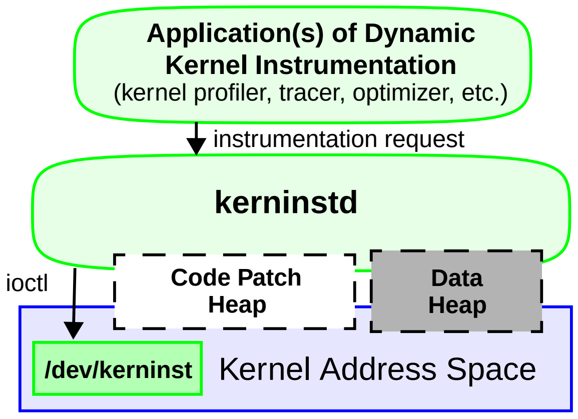


Figure 3.1: KernInst System Architecture
Kerninstd acts as an instrumentation server, performing kernel instrumentation requests that arrive from applications.

driver executes from within the kernel's address space, it gives kerninstd the ability to perform certain operations that cannot be performed by a user-level process.

3.2 Bootstrapping

Before kernel instrumentation can take place, kerninstd must attach itself on to the running kernel, by allocating the code patch area heap, parsing the kernel's run-time symbol table, and obtaining permission to write to any portion of the kernel's address space. These actions present several technical challenges.

Kerninstd cannot directly allocate kernel memory, so it can not allocate the code patch heap. Instead, it has /dev/kerninst perform the necessary kmem_alloc call. Once allocated, kerninstd can write to this memory through /dev/kmem.

To instrument, kerninstd needs to know where functions reside in memory. Thus, it needs access to the kernel's run-time symbol table. The static symbol table used at boot time, from /kernel/unix, is insufficient for two reasons. First, it is incomplete; most of the kernel is contained in modules that are loaded at boot-time or run-time.

Second, all kernel modules are relocatable files; their location in memory is unknown until loaded into the kernel. /dev/kerninst reads the kernel's run-time symbol table directly from kernel memory on behalf of kerninstd. (Solaris contains a linked list of kernel modules, containing symbol locations.) Solaris already provides a similar interface through the driver /dev/ksyms. Kerninstd does not use /dev/ksyms, which returns the symbol table in ELF format, without associating global functions with their respective modules. Kerninstd obtains a copy of the kernel's symbol table from /dev/kerninst in about 275 ms.

Both emitting into a code patch and splicing require writing to the kernel's address space. Kerninstd uses /dev/kmem when possible. Unfortunately, when running on an UltraSPARC processor, Solaris 7 (and earlier versions) cannot write to a kernel area known as the *nucleus*. The nucleus is a single 4 MB super-page I-TLB entry, of which 2 MB is reserved for kernel code. The kernel's run-time linker in Solaris 7 puts kernel modules in this 2 MB area whenever possible to minimize kernel I-TLB misses. In Solaris 2.6, only the kernel's core modules (unix, the architecture-specific part of the kernel; krtld, the kernel's run-time linker; and genunix, the architecture-independent part of the kernel) are placed in the nucleus. Because writing using /dev/kmem only succeeds when the virtual address is directly writable, it cannot change code within the nucleus, which is not mapped into the D-TLB. To circumvent this problem, kerninstd instructs /dev/kerninst to map the appropriate page (with a call to segkmem_mapin), perform the write, and then unmap it (using segkmem_mapout).

Although later releases of Solaris have a /dev/kmem driver that can write to the nucleus (using the same solution that /dev/kerninst employs), the limitations that KernInst had to work around have taught us some valuable lessons. First, modifying kernel code is hard. Second, despite the ubiquity of /dev/kmem in UNIX systems, the kernel designers may not have anticipated its use in hot-patching the kernel's code. Third, even bypassing limited interfaces may not always work; /dev/kerninst could not trivially write to kernel code within the nucleus. Fortunately, the fourth lesson was that our back-door access to the kernel while bootstrapping, /dev/kerninst, can work around such problems.

3.3 Structural Analysis: Control Flow Graph

On startup, kerninstd performs a structural analysis of the kernel's machine code, creating a per-function control flow graph (CFG) of basic blocks, calculating a call graph, and performing an interprocedural live register analysis. Higher-level tools built on top of kerninstd (such as kperfmon) may perform run-time compiler-like transformations. As such, these tools would benefit greatly from information that compilers and linkers discard in whole or in part. Kerninstd's structural analysis phase attempts to reconstruct a useful subset of this information, without the benefit of kernel source code. This section discusses the first of these structural analysis steps, constructing the kernel's control flow graphs.

Kerninstd uses the control flow graphs for several purposes. First, it is needed to calculate live register information (Section 3.5). Second, a CFG provides, more accurately than the runtime symbol table, the exact locations of a function's code, and hence the set of allowable instrumentation points. Accurate determination of instrumentation points is important, to ensure that splicing only takes place at code, as opposed to data embedded within a function or gaps between functions.

Higher-level tools built on top of kerninstd, such as kperfmon, may also have use of the control flow graphs. For example, a basic block code coverage tool needs to know the addresses of each basic block. Similarly, kperfmon needs to find a function's entry and exit point(s) when inserting instrumentation to start and stop an event accumulator. A function's exit point(s) are easily determined given its CFG: the ends of basic blocks having no successors.

3.3.1 Building a CFG: Basic Rules

Kerninstd creates one control flow graph per kernel function, directly reading and parsing kernel machine code from memory at run-time. The only *a priori* information that kerninstd relies upon for parsing are the entry addresses of each kernel function, from the kernel run-time symbol table. No source code is required, and no other information from the kernel's run-time symbol table is used.

A basic block is a single-entry and single-exit sequence of contiguous instructions. (Kerninstd makes a few exceptions to this definition, which are discussed in the next section.) Each block has predecessor and successor edges that represent the possible flow of control into and out of that block, respectively.

Construction of a function's control flow graph begins with its entry address, from the run-time symbol table. The basic block with that address is then parsed. With some exceptions that will be discussed, a basic block includes all instructions up to and including a control flow transfer instruction. If the control flow instruction keeps control within the function (e.g., is not a return or a procedure call), then its destination address(es) are recursively parsed in a depth-first manner as new basic block(s). If a basic block with a particular start address was already parsed, then only graph edges are added. If an address falls in the middle of an already-parsed basic block, that block is split in two, with control flow edges showing the first block falling through to the second one.

There are some cases where parsing of a basic block ends before reaching a control transfer instruction.

- If an instruction that would otherwise be included in the block is part of an another, already-parsed block, then basic block parsing ends prematurely. This situation occurs because the CFG is created using a depth-first traversal; a function's code is not scanned sequentially.
- Some assembly language functions have multiple entry points: kerninstd recognizes each entry point as a different function. To prevent overlapping functions, kerninstd marks an edge in the CFG for the first function as falling through to another function, ending the basic block from the first function prematurely. This pattern, which occurs 114 times in the Solaris kernel, can create some difficulties for instrumentation clients. For example, kperfmon places some timing instrumentation code at a function's exit, under the assumption that the function has completed. For a function that falls through to another, the instrumentation would be placed at the point of fall-through, missing the time spent in the next function. (Instrumentation cannot simply be placed at the exit of the next function, because it would be executed not only when the first function falls through to it, as desired, but also whenever the next function was called directly.)

3.3.2 Building a CFG: Handling Control-Flow Instructions

This section describes the parsing of control flow instructions. There are simple cases such as conditional branches, returns, and calls, as well as more complex register-indirect jumps, which can implement jump tables (e.g., from C switch statements) and calls through a function pointer.

Most SPARC control transfer instructions are delayed, with the succeeding instruction (its *delay slot instruction*) executing before the control transfer takes effect. The motivation behind delay slots is to hide some branch latency by executing a useful instruction in the interim. One class of SPARC control transfer instructions, branches, have a bit that can annul (not execute) the delay slot instruction. A conditional branch with the annul bit set will execute the delay slot instruction if and only if the branch is taken. An unconditional branch with the annul bit set will ignore the delay slot instruction.

Normally, kerninstd includes a non-annulled delay slot instruction in the same basic block as its control transfer instruction. However, if the delay slot instruction also happens to be the first instruction of another basic block, then it remains in that other block. As we will see in Section 4.2, splicing cannot occur in such cases. Fortunately, the situation only arises four times in the Solaris 7 kernel. (An earlier study [98] found that it also occurs four times in the Solaris 2.5.1 kernel.) Kerninstd does not permit splicing at these four instructions.

As with the non-annulled case, conditionally annulled delay slot instructions are normally kept in the same basic block as their branch control transfer instruction, although the delay slot instruction is only executed when the branch is taken.

Logically, this delay slot instruction should be in its own basic block, reachable only along the if-taken branch, as in Figure 3.2. However, kerninstd does not use this organization, because it would greatly increase the number of basic blocks (21,105 of the 72,454 conditional branch instructions in the kernel have the annul bit set), and therefore memory usage. This space optimization requires that algorithms working with the control flow graph interpret conditionally annulled delay slot instructions carefully. For example, the live register analysis algorithm (Section 3.5) must interpret registers that are *definitely* written by such a delay slot instruction as *maybe* written. To aid the live register analysis algorithm, an exception to this parsing organization rule is made when the conditionally annulled instruction is a SPARC register window save or restore instruction (see Section 3.5.1). This exception was necessary because with such instructions, there is no means to conditionally interpret the effect of the instruction on register state. In these instances, which occur about 300 times in the kernel, a single-instruction basic block holding the conditionally annulled delay instruction is created, as in Figure 3.2.

As will be discussed in Section 3.3.4, there are some cases where a control flow instruction cannot be analyzed. Although there is a safe backup to understanding a complex control transfer instruction, marking a block's successor edge as unanalyzable, it is important to parse control flow instructions accurately, for four reasons. First, control flow graphs, and in particular the boundaries of each basic block, specify the locations of all kernel code, and thus the set of allowable instrumentation points. Basic block parsing cannot continue when the destination address(es) are unknown. If the jump will keep control within the function, then the

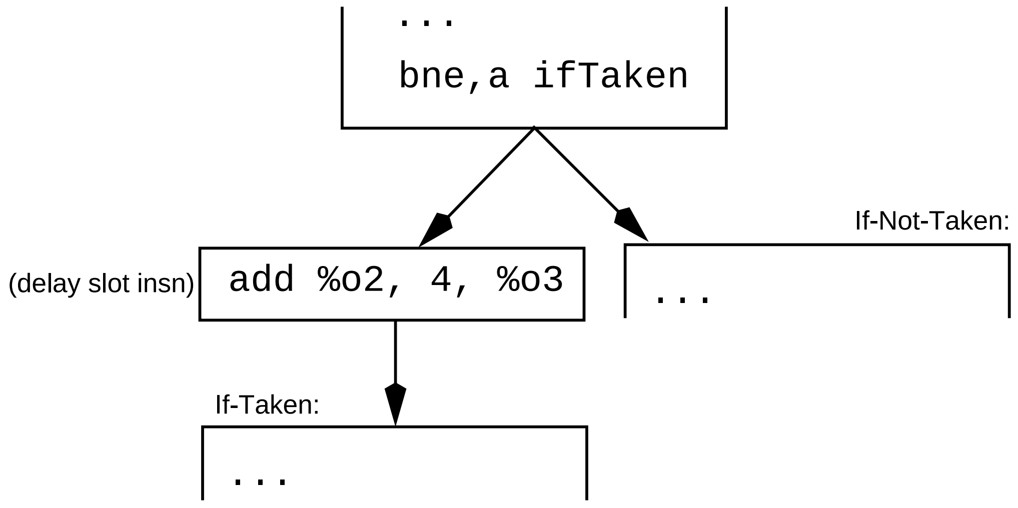


Figure 3.2: Conventional Placement of Conditionally Annulled Delay Instruction
The delay slot instruction (add %o2, 4, %o3) is placed in its own basic block, reachable only when the branch is taken. Unfortunately, this single-instruction basic block is not an efficient use of memory for KernInst's control flow graph or live register analysis.

opportunity to parse some basic block(s) may be lost. Second, making an over-cautious analysis may lead algorithms using the CFG, such as live register analysis, to return conservative results. Third, some operations may lead to inaccurate results. For example, kperfmon would have to (arbitrarily) guess whether an unanalyzable edge exits the function, thus deciding whether instrumentation placed at a function's exit site(s) belongs there. Fourth, there are some algorithms, such as the run-time basic block reordering optimization discussed in Chapter 8, that cannot work with any unanalyzable edges in the graph. In this example, the problem is that the jump instruction may implement an unknown kind of jump table. Correctly relocating a jump table sequence when its destination blocks have changed order requires making corresponding changes to the jump table data. Kerninstd is presently very cautious when it encounters an unanalyzable jump, giving up on parsing the function's CFG entirely. To be safe, kerninstd presently precludes instrumentation anywhere in an unanalyzable function. (Section 3.3.4 summarizes the set of unanalyzable kernel functions.)

The control transfer instructions that kerninstd recognizes are:

- **Branches**. A branch instruction will always be at the end of a basic block. The branch destination is recursively parsed as a new basic block in the CFG if the branch is local. Interprocedural branches occur about 600 times in the kernel.

  If the branch was conditional, the fall-through address is then recursively parsed, except where the fall-through address is the start of another function.
- **Jumps to a Constant Address.** Code that jumps to a register having previously been set to a constant address is detected and treated like an unconditional branch. Detection is done by taking a backwards slice on that register and checking whether the instructions of that slice set that register to a constant value. Even though this pattern occurs only two times in the kernel, recognizing it leads to a more precise analysis.
- **Returns**. A return sequence is always found at the end of a basic block. There are three standard return instructions on the SPARC: ret, return, and retl. Not all returns use the standard instructions. Kerninstd also recognizes another kind of return sequence, a jump to a register whose value is a saved copy of the link register on function entry. This unusual sequence is found by examining a backwards slice on the register; it occurs five times in the kernel.
- **Procedure Calls.** According to strict definition, a normal procedure call (i.e., one that is expected to return) should end a basic block, because it transfers control. However, kerninstd uses a modified definition of a basic block: once entered, all code within the block will execute. This allows instructions after the call to be included in the same basic block, assuming the procedure call will eventually return. The call edge is represented elsewhere, in kerninstd’s call graph

(Section 3.4). Algorithms such as live register analysis must manually check for procedure calls within basic blocks, and handle them accordingly.

Including procedure calls into the middle of a basic block reduces the amount of storage required for the CFG and live register analysis; of the 65,506 procedure calls in the kernel, 57,247 of them are located in the middle of a basic block. A variant of a procedure call, where the destination is specified in a register (this usually implements a call through a function pointer) occurs 2,389 times in the kernel. Code traversing the CFG must note such unanalyzable calls and make conservative assumptions about the possible destination(s). It can assume, however, that the call will return; this sequence should not be confused with the unanalyzable jump discussed above.

- **Tail Calls.** A *tail call* is one that will cause the called function to return not to its caller, as is the norm, but to where the caller itself would normally return. A tail call has the same semantics as a call followed by a return, but may execute faster. It is important for kerninstd to properly recognize and parse tail calls, because they serve as both call sites and return points.

  Two forms of tail calls are common in optimized SPARC code. In the first form, the calling routine has its own stack frame (using the save instruction). A restore in the delay slot of the call tears down this frame, which includes the link register. Figure 3.3 illustrates this sequence, which occurs about 3,000 times in the kernel. In the other tail call variant, found about 800 times, the calling routine does not have a stack frame, and simply jumps to the callee (without writing to the link register). This kind of tail call is an interesting technical example, showing that under compiler optimizations, certain C functions can make procedure calls even without having a stack frame (so long as all of its calls are tail calls). A variant of the above patterns, where the callee address is in a register (as in a tail call through a function pointer), is not presently analyzed by kerninstd; the ramifications are discussed in Section 3.3.4.

- **Intraprocedural calls.** A call whose destination is within the same function occurs 38 places in the kernel, and is usually recursion. Another three calls are

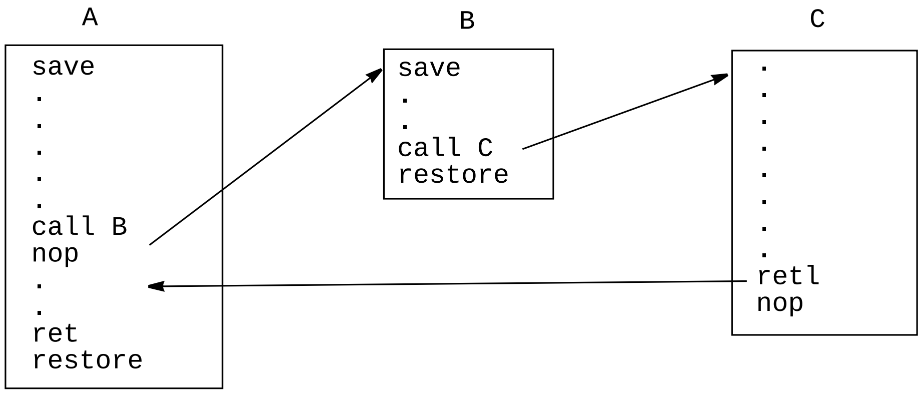


Figure 3.3: Tail Call
In this example, function B (called from A) makes a tail call to C. B has a restore instruction in the delay slot of the call to C, which changes to A's stack frame. The link register now contains the address in A where it calls B, so C will return directly to A.

intraprocedural yet are not recursive. In these cases, kerninstd assumes that the callee is a function that was missing in the symbol table, and re-parses accordingly. (Unfortunately, it cannot give the new function a meaningful name.) Another odd form of procedure call has a destination address of the current address plus two instructions. This sequence can be found in assembly code that wishes to read the program counter, by using a side effect of the call instruction, writing the current program counter to the link register. Although it was found in earlier versions of the kernel, this sequence does not occur in Solaris 7, which uses a SPARC v9 instruction [103] for reading the program counter directly. It is important to recognize such sequences to avoid the spurious recognition of a new function.

- **Jump Tables.** A jump table usually implements a C switch statement. There are several variants; all end by jumping to an address in a single register or to an address that is the sum of two registers. The individual cases are identified by performing a backwards slice on those register(s), and examining the slice for a particular pattern. It is important to accurately recognize jump tables, because an unrecognized jump instruction will be considered unanalyzable, causing kerninstd to give up parsing (and the ability to instrument) many functions.

The slice approach, modeled after a similar design in EEL [54], facilitates the search for particular jump table patterns by filtering out instructions that could not have affected the jump. Once a match is made for a particular kind of jump table, enough information has been obtained to know the possible destination addresses, which are recursively parsed as successor basic blocks. In the CFG, a basic block ending in a jump table looks much like a block ending in a conditional branch, except that the number of successor blocks is variable: the number of unique jump table entries.

The kinds of jump tables that kerninstd recognizes are:

❍ **Offset Jump Table.** This sequence sets one register to the start of the jump table data. Another register, typically the C variable controlling the switch, is used to calculate a byte offset within the jump table. The jump table entry is loaded, obtaining an offset that is added to the start of the jump table to produce the destination basic block address. Offset jump tables occurs 143 times in the kernel, having on average 9.0 successor basic blocks.

❍ **Simple Jump Table.** The contents of the jump table are 32-bit absolute addresses, not offsets from the start of the jump table. One register is set to the constant address of the start of the jump table data. Another register is used to determine the byte offset within the jump table. A jump table entry is loaded from the sum of these registers; the loaded value is an address that is then jumped to. This kind of jump table occurs 12 times in the kernel, having on average 5.2 successor basic blocks.

❍ **Tagged Jump Table.** Each tagged jump table entry consists of a four byte tag, against which the control variable is compared, and a destination address associated with the tag. Potentially, the entire jump table needs to be searched to find a match. Although previous versions of the Solaris kernel contained this type of jump table, it is not present in Solaris 7, due to compiler changes.

- **Non-Loading Jump.** This not a jump table per se, since no indirect load is performed to calculate the destination address. Instead, one register is set to a base location in code, the second register produces an offset, and control is transferred to that location. This kind of jump is used when the destination basic blocks are of equal size and are laid out sequentially. In Solaris 7, this occurs in some assembly routines: blkclr (two occurrences) and hwblkclr, having on average 12.3 successor basic blocks.

3.3.3 Control Flow Graph Results: What Was Parsed

The CFG is useful to many instrumentation applications, yielding estimates on perturbation and the number of instrumentation points that may be required, for example. This section summarizes the contents of the kernel's control flow graphs, presenting code size, the number of functions and basic blocks, and the number of control edges into and out of basic blocks. Because kernel module sizes vary greatly, and because certain applications may be interested only in functions from specific modules, the information is presented per module. The result is contained in Figure 3.4, with modules sorted by their overall code size. The first column contains the module's name and description, followed by the module's code size, calculated by summing the size of each parsed basic block. As a result, this figure correctly excludes any dead code and embedded data, and so can be more accurate than the symbol table's function size field. The next column shows the number of functions in the module that were successfully parsed, giving (for example) the number of simultaneous instrumentation points needed to measure the number of function calls made to code in this module. The column's additional number in parenthesis, if any, is the number of functions that were not successfully parsed. The next column gives the number of parsed basic blocks, which is useful in estimating the number of instrumentation points needed to measure code coverage, for example. Next are columns for the average number of basic blocks per function and average number of instructions per basic block, which are useful in estimating the perturbation of instrumentation code. The final two columns give information about the edges in the CFG: the average number of successors and the average number of predecessors of a basic block in this module, useful for estimating the amount of instrumentation needed to measure edge coverage, for example.

Module	#bytes	# fns (+unparsed)	# bbs	bbs/fn	insns/ bb	succs/ bb	preds/ bb
genunix	646116	2589 (+113)	32360	12.5	5	1.5	1.3
afs (afs syscall interface)	516176	922	17654	19.1	7.3	1.5	1.4
unix	279264	1758 (+37)	12705	7.23	5.5	1.5	1.2
ufs (filesystem for ufs)	146776	337	6861	20.4	5.3	1.5	1.4
nfs (NFS syscall, client, and common)	139336	479	6526	13.6	5.3	1.5	1.3
ip (IP Streams module)	131092	373	7230	19.4	4.5	1.5	1.4
md (Meta disk base module)	113872	390	5556	14.2	5.1	1.5	1.3
tcp (TCP Streams module)	75764	159	3748	23.6	5.1	1.5	1.4
procfs (filesystem for proc)	69804	174	3431	19.7	5.1	1.5	1.4
sd (SCSI Disk Driver 1.308)	65948	115	3005	26.1	5.5	1.5	1.4
rpcmod (RPC syscall)	51408	209 (+12)	2516	12	5.1	1.5	1.3
sockfs (filesystem for sockfs)	48664	149 (+2)	2482	16.7	4.9	1.5	1.3
pci (PCI Bus nexus driver)	39656	127	1614	12.7	6.1	1.5	1.3
hme (FEPS Ethernet Driver v1.121)	39636	97	1773	18.3	5.6	1.5	1.3
se (Siemens SAB 82532 ESCC2 1.93)	38184	69	1750	25.4	5.5	1.5	1.4
fd (Floppy Driver v1.102)	35200	54 (+1)	1609	29.8	5.5	1.6	1.5
zs (Z8530 serial driver V4.120)	31996	48	1401	29.2	5.7	1.5	1.4
uata (ATA AT-bus attachment disk controller Driver)	29296	127 (+2)	1228	9.67	6	1.5	1.2
krtld	28780	127 (+1)	1475	11.6	4.9	1.5	1.3
rpcsec (kernel RPC security module.)	27204	122 (+3)	1297	10.6	5.2	1.5	1.2
ufs_log (Logging UFS Module)	27196	131	1191	9.09	5.7	1.5	1.3
xfb (xfb driver 1.2 Sep 7 1999 11:46:39)	23820	99	1154	11.7	5.2	1.5	1.3
audiocs (CS4231 audio driver)	22964	83	942	11.3	6.1	1.5	1.3
dad (DAD Disk Driver 1.16)	22340	56	992	17.7	5.6	1.5	1.3
tmpfs (filesystem for tmpfs)	20184	66	901	13.7	5.6	1.5	1.3
ldterm (terminal line discipline)	19740	45	1122	24.9	4.4	1.5	1.4
afb (afb driver v1.36 Sep 7 1999 11:47:45)	18356	60	630	10.5	7.3	1.5	1.4
scsi (SCSI Bus Utility Routines)	17388	78 (+9)	827	10.6	5.3	1.5	1.3
tl (TPI Local Transport Driver - tl)	16756	57	893	15.7	4.7	1.5	1.3
specfs (filesystem for specfs)	15704	48 (+2)	653	13.6	6	1.5	1.3

Figure 3.4: Summary Structural Analysis Kernel Information

Modules vary greatly in their number of functions, basic blocks, and bytes of code. Surprisingly, they also vary on the average number of blocks per function, and to a lesser extent, the average block size. However, control flow edge information (numbers of successors and predecessors) is nearly uniform.

Module	#bytes	# fns (+unparsed)	# bbs	bbs/fn	insns/ bb	succs/ bb	preds/ bb
arp (ARP Streams module)	15600	67 (+1)	885	13.2	4.4	1.5	1.3
SUNW,UltraSPARC-IIi	15488	60 (+3)	584	9.73	6.6	1.5	1.3
vol (Volume Management Driver, 1.85)	15360	23	752	32.7	5.1	1.5	1.5
doorfs (doors)	14480	52	713	13.7	5.1	1.5	1.3
su (su driver 1.24)	13700	31	592	19.1	5.8	1.5	1.4
udp (UDP Streams module)	13616	43	720	16.7	4.7	1.5	1.3
timod (transport interface str mod)	13584	36	669	18.6	5.1	1.5	1.4
kerninst (kerninst driver v0.4.1)	13360	116	618	5.33	5.4	1.4	1.1
fifofs (filesystem for fifo)	11608	40 (+2)	575	14.4	5	1.5	1.4
kb (streams module for keyboard)	10804	36	624	17.3	4.3	1.5	1.3
tnf (kernel probes driver 1.47)	10768	55	491	8.93	5.5	1.5	1.2
pm (power manager driver v1.65)	10052	29	501	17.3	5	1.6	1.4
TS (time sharing sched class)	9936	36	501	13.9	5	1.5	1.3
devinfo (DEVINFO Driver 1.24)	9632	35	425	12.1	5.7	1.5	1.3
ipdcm (IP/Dialup v1.9)	9516	52	516	9.92	4.6	1.5	1.3
ttcompat (alt ioctl calls)	8480	14	457	32.6	4.6	1.5	1.4
diaudio (Generic Audio)	8220	28 (+1)	486	17.4	4.2	1.5	1.4
elfexec (exec module for elf)	8008	12	283	23.6	7.1	1.6	1.5
shmsys (System V shared memory)	7728	19	316	16.6	6.1	1.6	1.4
ptc (tty pseudo driver control ‘ptc’)	7412	16	401	25.1	4.6	1.6	1.4
tlimod (KTLI misc module)	6468	21	360	17.1	4.5	1.5	1.3
winlock (Winlock Driver v1.39)	5484	39	284	7.28	4.8	1.4	1.2
hwc (streams module for hardware cursor support)	5428	11	250	22.7	5.4	1.5	1.2
ms (streams module for mouse)	5340	17	304	17.9	4.4	1.5	1.3
ptem (pty hardware emulator)	4996	13	294	22.6	4.2	1.5	1.3
simba (SIMBA PCI to PCI bridge nexus driver)	4728	18	172	9.56	6.9	1.5	1.2
seg_drv (Segment Device Driver v1.1)	4544	25 (+2)	215	8.6	5.3	1.5	1.2
sad (Streams Administrative driver’sad’)	4528	23	234	10.2	4.8	1.5	1.2
namefs (filesystem for namefs)	4464	32	165	5.16	6.8	1.4	1.1
lockstat (Lock Statistics)	4436	27	206	7.63	5.4	1.5	1.2
ptsl (tty pseudo driver slave ‘ptsl’)	4396	14	215	15.4	5.1	1.5	1.3
rootnex (sun4u root nexus)	4320	19	230	12.1	4.7	1.5	1.2
dada (ATA Bus Utility Routines)	3988	30 (+3)	209	6.97	4.8	1.5	1.3
dada_ata (ATA Bus Utility Routines)	3880	30 (+3)	203	6.77	4.8	1.5	1.3
md5 (MD5 Message-Digest Algorithm)	3672	8	28	3.5	33	1.4	1
sysmsg (System message redirection (fanout) driver)	3556	15	187	12.5	4.8	1.5	1.3
mm (memory driver)	3440	13	175	13.5	4.9	1.5	1.3
wc (Workstation multiplexer Driver ‘wc’)	3384	18	186	10.3	4.5	1.5	1.2
ebus (ebus nexus driver)	3336	13	136	10.5	6.1	1.5	1.2
ptm (Master streams driver ‘ptm’)	3052	12	149	12.4	5.1	1.5	1.3
pts (Slave Stream Pseudo Terminal driver ‘pts’)	2944	12	148	12.3	5	1.5	1.3
RT (realtime scheduling class)	2780	24	139	5.79	5	1.4	1.1
iwscn (Workstation Redirection driver ‘iwscn’)	2740	19	124	6.53	5.5	1.5	1.2
fdfs (filesystem for fd)	2668	17	124	7.29	5.4	1.4	1.2
eide (PC87415 Nexus driver v2.0)	2584	19	111	5.84	5.8	1.5	1.1
conskbd (Console kbd Multiplexer driver ‘conskbd’)	1972	14	121	8.64	4.1	1.4	1.1
todmostek (tod module for Mostek M48T59)	1872	10	33	3.3	14	1.4	1
log (streams log driver)	1768	9	83	9.22	5.3	1.5	1.2
sy (Indirect driver for tty ‘sy’)	1704	12	80	6.67	5.3	1.4	1
consms (Mouse Driver for Sun ‘consms’)	1700	14	105	7.5	4	1.4	1.1

Figure 3.4: Summary Structural Analysis Kernel Information

Modules vary greatly in their number of functions, basic blocks, and bytes of code. Surprisingly, they also vary on the average number of blocks per function, and to a lesser extent, the average block size. However, control flow edge information (numbers of successors and predecessors) is nearly uniform.

Module	#bytes	# fns (+unparsed)	# bbs	bbs/fn	insns/ bb	succs/ bb	preds/ bb
kstat (kernel statistics driver)	1580	12	84	7	4.7	1.4	1
pckt (pckt module)	1480	11	92	8.36	4	1.5	1.3
ksyms (kernel symbols driver)	1480	11	61	5.55	6.1	1.4	1
inst_sync (instance binding syscall)	1368	11	57	5.18	6	1.5	1.2
power (power driver v1.4)	1336	11	66	6	5.1	1.4	0.98
cn (Console redirection driver)	1196	13	60	4.62	5	1.4	1
sysacct (acct(2) syscall)	1168	6	46	7.67	6.3	1.5	1.2
clone (Clone Pseudodriver ‘clone’)	992	7	54	7.71	4.6	1.4	1.1
intpexec (exec mod for interp)	896	5	51	10.2	4.4	1.6	1.3
pseudo (nexus driver for ‘pseudo’)	872	10	44	4.4	5	1.5	0.98
ipc (common ipc code)	564	5	38	7.6	3.7	1.5	1.1
pipe (pipe(2) syscall)	472	4	13	3.25	9.1	1.3	1
connld (Streams-based pipes)	312	6	16	2.67	4.9	1.4	0.94
options (options driver)	280	7	24	3.43	2.9	1.3	0.75
redirmod (redirection module)	244	6	12	2	5.1	1.2	0.58
TS_DPTBL (Time sharing dispatch table)	80	5	5	1	4	1	0
IA (interactive scheduling class)	76	3	5	1.67	3.8	1.2	0.4
RT_DPTBL (realtime dispatch table)	56	3	3	1	4.7	1	0
platmod	40	5	5	1	2	1	0
Totals:	3110636	10637 (+197)	142641	13.4	5.5	1.5	1.3

Figure 3.4: Summary Structural Analysis Kernel Information

Modules vary greatly in their number of functions, basic blocks, and bytes of code. Surprisingly, they also vary on the average number of blocks per function, and to a lesser extent, the average block size. However, control flow edge information (numbers of successors and predecessors) is nearly uniform.

The results contain a few surprises. First, even excluding the kernel’s core (unix, krtld, and genunix), the modules vary greatly in total size. Also surprising was the wide discrepancy in the average number of basic blocks per function. The modules with the greater number of total instructions (near the top of the table) also tended to have more basic blocks per function. The average number of instructions in a basic block also varied, though not as widely. The moderate variance is in line with another study of operating system basic block lengths [15]. The average number of instructions per basic block can aid in estimating the perturbation introduced by per-block instrumentation (such as a basic block code coverage tool). For example, if instrumentation contains twelve instructions, one would expect code being profiled in such a manner to approximately triple in size, when including the size of instrumentation. There was near uniformity in control flow edges; a basic block averages about 1.5 successors and 1.3 predecessors.

A summary of the time it takes to create the control flow graph is shown in Figure 3.5. Since this action is performed only when kerninstd starts, the cost (under five seconds) is quite reasonable.

Parsing Step	Time (seconds)
Read function's code from kernel memory	0.45
Add this code to the function object (makes a copy of the code)	0.15
Parse basic blocks, creating CFG	2.6
Update the call graph (Section 3.4)	0.65
Other actions	0.45
Total	4.3

Figure 3.5: Time to Parse Function Control Flow Graphs
The above times are for kerninstd compiled with g++ 2.95.2, optimization level -O2, and assertions disabled running on a 440 MHz UltraSPARC processor. After some preliminary actions, most notably reading code from kernel memory, CFG parsing requires less than three seconds. Together, this phase of kerninstd startup takes about 4.3 seconds.

A greater concern with the control flow graph is its memory consumption. Structures to hold the basic blocks consume 3.6 MB of memory, which includes the block's address boundaries, its predecessor and successor edges, and information to facilitate searching for basic blocks by address; it does include not the code within the block. The actual kernel code, as originally parsed, is kept in a per-function structure, and consumes 3.3 MB of memory. Storing kernel code is unnecessary; kerninstd could always read the code on demand via /dev/kmem. However, certain operations such as live register analysis need fast access to kernel code.

3.3.4 Control Flow Graph Results: What Was Not Parsed

Of the 10,834 functions, 197 are not presently parsed by kerninstd. This section analyzes the reasons behind these parsing failures, and the ramifications.

Three kernel functions could not be parsed because they make calls to routines that will not return, such as to panic. Kerninstd's parsing can fail in this case because it assumes that the call will return. It continues to parse, running off the end of the function. If a list of functions that will not return were known to kerninstd, then these routines could be parsed.

Another four kernel functions perform a jump that behaves like a return, but are not recognized as such because a backwards slice on the destination register does not show it being initialized with a copy of the link register on function entry.

Twelve assembly functions perform unique patterns of unanalyzable jumps. They will likely be the most difficult routines to parse correctly.

The remaining 177 un-parsed functions all have a common form of tail call that kerninstd does not parse. These routines have no stack frame of their own, and make an indirect tail call by simply jumping to the contents of a register, without setting a link register. Kerninstd does not currently recognize this sequence because there is no set pattern to the backwards slice on the register containing the destination address. To parse these routines, kerninstd would have to assume that all unanalyzable jumps to a register are in fact this kind of tail call—an incorrect assumption for some of the sequences discussed earlier in this section. In addition, if a new jump table format were introduced into the kernel code, which often happens when a kernel upgrade contains code generated with a new compiler version, hundreds of misidentifications could occur.

Nevertheless, it should be possible to properly recognize and parse most, if not all, of these functions with some additional work. Although there may not be one fixed pattern to which they conform, there may be a manageable set that can match these sequences without adding undue complexity to kerninstd.

The repercussions of unanalyzable jumps are presently minor. The 177 indirect tail calls are normally stub functions that simply perform one or more loads to obtain a function pointer, which is used for a jump. Stub routines are presumably of little interest for performance measurement and optimization. On the other hand, a few such indirect tail calls are within non-trivial functions that may be of interest. The other 20 unanalyzable functions are mostly, though not entirely, routines of no interest because they are only invoked during booting or a kernel panic. In all, despite 197 un-parsed functions, kerninstd is able to function well.

3.4 Structural Analysis: Call Graph

The call graph represents interprocedural calls and branches, providing a complement to the control flow graphs, which maintain intraprocedural control transfers.

The call graph is constructed by traversing the basic blocks of all control flow graphs, identifying calls and interprocedural branch instructions with fixed destination addresses. Register-indirect calls or jumps (such as through a function pointer in C) having a statically unknown destination are not put in the call graph.

Construction of the call graph, updated incrementally after each function’s control flow graph is parsed, takes 0.65 seconds. The call graph consumes about 0.6 megabytes of memory in kerninstd.

3.5 Structural Analysis: Live Register Analysis

During startup, kerninstd performs a dataflow analysis on the kernel, yielding the registers that are live (contain a value that may be used in the future) and dead (will definitely be written before read) at each instruction. The register liveness information lets applications generate code snippets that use dead registers at an instrumentation point, when possible, instead of spilling to the stack to free up some scratch registers. Live register information is also used internally by kerninstd to obtain a scratch register, when needed, for jumping to and from a code patch, and for relocating the overwritten instruction at the instrumentation point.

3.5.1 Dataflow Representation and Basic Algorithms

A *dataflow function* [1] represents the effects of executing code (such as an individual instruction, a basic block, or an entire function) on the liveness of each register. For example, a dataflow function may indicate that %o0 and %o1 are killed, %o3 and %o7 are made live, and all other registers are unaffected.

Dataflow functions can be composed to represent sequential execution. Successive composition of dataflow functions for each instruction of a basic block, in reverse order, yields a dataflow function for the basic block as a whole. Two dataflow functions may be merged (union of registers made live; intersection of registers killed; intersection of registers unaffected) to combine the effects of both the if-taken and if-not-taken successors of a conditional branch, for example.

Dataflow functions usually require two bits per register [1]. KernInst represents 32 integer registers, 64 floating point registers, and several other registers (such as condition codes, program counter, and processor interrupt level) in its dataflow functions, totaling 16 bytes. However, the SPARC architecture complicates matters.

Most of the SPARC integer registers are organized in partially overlapping *register windows* [103]. At any time, a program can access eight non-windowed integer registers called global (%g0-%g7), and 24 windowed integer registers called input (%i0-%i7), local (%l0-%l7), and output (%o0-%o7). A save instruction creates a new window by shifting the register window as shown in Figure 3.6. The register window stack is conceptually infinite, though only the topmost eight windows are kept on both UltraSPARC-I [96] and UltraSPARC-II [97] implementations. A save that overflows the on-chip window stack causes a trap; the handler writes the oldest window to memory. A restore is the opposite of save. It can trap by underflowing the on-chip window; the handler reads one window from memory.

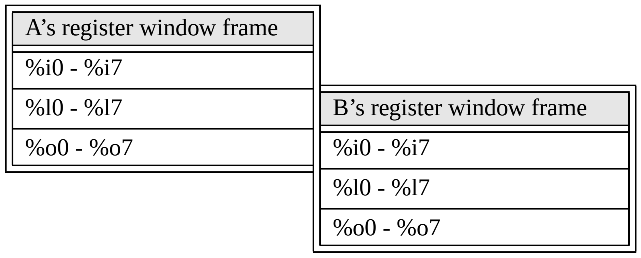


Figure 3.6: SPARC register window overlapping

In this example, A has its own register window frame. It then calls B, which sets up its own frame by executing a save instruction. B gets a new set of %l and %o registers, and its %i registers overlap with (i.e., are aliases for) the previous frame's %o registers.

Register windows make live register analysis challenging because different places in the code may (implicitly) use different windows. A KernInst dataflow function is therefore a stack of dataflow function windows, with one level (using 16 bytes of storage) for each register window accessed by the code being analyzed.

3.5.2 Startup Algorithm: First Phase

Per-block dataflow functions are calculated after kerninstd parses the kernel’s control flow graphs. Each dataflow function represents the effect on register liveness of executing a routine up to the beginning of that basic block. When the startup algorithm completes, a dataflow function for the effect of executing an entire routine is available in that routine’s entry block dataflow function.

The startup algorithm is as follows. Kerninstd first calculates dataflow functions for each basic block in isolation (as if blocks had no successors or predecessors). As mentioned in Section 3.5.1, the calculation iterates backwards through a block, composing the dataflow effect of each instruction. Calls can be present in basic blocks (see Section 3.3.2). The effect of a callee is considered before the effect of the call’s delay slot instruction, which is considered before the call’s side-effect of setting the link register. (This ordering is consistent with the temporal nature of backwards dataflow problems.) A callee’s effect is determined by recursively performing live register analysis on it if needed, in a depth-first traversal. Thus, kerninstd’s live register analysis is interprocedural.

A conservative dataflow result is used for an unanalyzable callee. This situation occurs for functions whose CFG could not be parsed, for calls through a pointer, and for callees that could not be recursively analyzed due to a cycle in the call graph. SPARC register windows make calculating a conservative result for a routine difficult. Each window in the dataflow function (see Section 3.5.1) will report all registers made live, but the number of such windows must be chosen. Most routines have balanced saves and restores, requiring a one-window dataflow function. But some routines are imbalanced, and incorrectly guessing the number of stack windows confuses live register analysis. In particular, it is unclear how to merge two dataflow functions with a disagreeing number of windows. Kerninstd is able to work around this problem by hard-wiring the number of stack windows needed for the dataflow functions of certain routines. Only a single routine, stubs_common_code, presently needs to be so kludged. This routine, which performs a restore but no save, is a commonly invoked wrapper for calling a function in another module, which may need to be loaded and locked down. Because stubs_common_code is indirectly recursive, a conservative result is needed during the depth-first traversal of the call graph. Using a conservative dataflow function with the proper number of stack windows for stubs_common_code enables analyzing over 500 routines that otherwise would fail. There are many such unbalanced routines in the kernel, and all but stubs_common_code had this attribute determined automatically.

Other control flow effects (both intra- and interprocedural branches, jump tables, etc.) are considered along control flow graph edges, not within basic blocks, and thus are not considered during the first phase.

3.5.3 Startup Algorithm: Second Phase

The second phase of the startup algorithm calculates per-block dataflow functions representing execution of a routine up to the beginning of each block (not in isolation, as in phase one). Intermediate results from the first phase are used in this calculation, then discarded. The dataflow function for a basic block is re-calculated, whenever the dataflow function for one of its successor blocks changes, using the following steps. First, the results calculated thus far for each successor block are gathered. Results for interprocedural edges (such as an interprocedural branch or an interprocedural fall-through) are gathered like the results for calls were gathered in the first phase: by recursively analyzing the destination routine, if needed. Next, the dataflow function for this basic block in isolation (from phase one) is applied to the result for each edge. These results are then merged to obtain a new dataflow function for execution up to the beginning of the basic block. The block's predecessor(s) then have their dataflow functions re-calculated. Phase two repeats until convergence.

Conceptually, it would be ideal to store the dataflow functions for every instruction, making it trivial to query the set of scratch registers available at an instrumentation point. However, the required storage would be prohibitive, so kerninstd only keeps this information at the top of each basic block. Dataflow functions at individual instructions are calculated on demand at run-time (see Section 3.5.4).

Even throwing out all but the dataflow functions at the top of each basic block requires 11.5 MB of storage in kerninstd. A further time/space trade-off could be made, such as storing only dataflow functions at the entry of each function. Dataflow functions at instruction granularity can still be calculated at run-time on demand.

The startup phases of interprocedural live register analysis complete in about 3.2 seconds.

3.5.4 Runtime Algorithm

Given dataflow functions representing the effects of execution up to the top of every basic block, it is straightforward to calculate dataflow functions up to the beginning of a desired instruction at run-time. This is the same information that was calculated at intermediate points during the second phase of the startup algorithm. First, a dataflow function for execution up to the bottom of the basic block is calculated by merging the dataflow functions of the block's successors. Next, a dataflow function representing execution of *part* of the basic block in isolation—from just before the instruction in question to the end of the basic block—is calculated, using the same logic as phase one of the startup algorithm. Finally, this function is composed with the dataflow function for the bottom of the basic block to obtain the desired result: a dataflow function representing execution of that routine up to, but not including, the instruction in question.

The time to execute the runtime algorithm depends on the number of instructions that need to be analyzed, which is the distance from the instruction in question to the end of the basic block, and the number of successors, which determines how many dataflow functions need to be composed and then merged. A typical case, with two successor blocks and five instructions in the basic block to be analyzed, completes in about 30 µs. About 16 µs of this time is to calculate the effect of the (partial) basic block, 4 µs for the two compositions, and the remainder to merge the compositions.

3.5.5 Live Register Analysis: Results

Knowing how many integer registers are dead at various instrumentation points is useful to instrumentation clients that hope to generate code snippets using only dead registers. Two categories of instrumentation points, function entry and exits, are of special interest because they are the ones most commonly instrumented by kperfmon. This section summarizes the results of kerninstd's interprocedural live register analysis, presenting the average number of dead registers at various points in the kernel. The number of dead registers tends to vary between modules,

Module	#save entry points	Avg. dead regs	#non-save entry points	Avg. dead regs	#exit points	Avg. dead regs	Avg. dead regs for every insn
genunix	1957	1.2	632	2.0	5487	13.1	6.2
afs (afs syscall interface)	922	0.3	0	--	1874	15.0	9.1
unix	1023	1.3	735	2.6	2993	9.4	7.0
ufs (filesystem for ufs)	300	1.2	37	1.9	936	14.5	5.7
nfs (NFS syscall, client, and common)	379	0.8	100	1.5	1480	14.0	6.2
ip (IP Streams module)	303	1.3	70	1.9	1099	13.6	7.6
md (Meta disk base module)	349	1.2	41	2.0	1028	14.7	7.3
tcp (TCP Streams module)	144	1.4	15	2.1	533	14.2	7.1
procfs (filesystem for proc)	150	1.5	24	1.8	444	14.3	6.1
sd (SCSI Disk Driver 1.308)	108	2.1	7	2.6	423	15.0	6.1
rpcmod (RPC syscall)	150	1.0	59	1.6	568	12.8	5.4
sockfs (filesystem for sockfs)	120	0.8	29	1.5	465	14.1	6.2
pci (PCI Bus nexus driver)	89	2.1	38	2.4	302	13.0	6.9
hme (FEPS Ethernet Driver v1.121)	85	1.2	12	3.0	322	13.7	7.7
se (Siemens SAB 82532 ESCC2 1.93)	62	1.1	7	3.0	200	14.7	7.6
fd (Floppy Driver v1.102)	50	1.5	4	3.5	159	15.1	6.9
zs (Z8530 serial driver V4.120)	36	1.4	12	2.2	136	14.0	6.0
uata (ATA AT-bus attachment disk controller Driver)	94	1.3	33	1.5	274	13.2	8.9
krtld	87	1.4	40	1.4	253	11.3	6.2
rpcsec (kernel RPC security module.)	94	0.9	28	2.0	385	14.2	7.3

Figure 3.7: Dead Registers in the Solaris 7 Kernel

Dead registers are available for scratch use in instrumentation code. Kerninstd's interprocedural live register analysis shows that function entry points have few dead registers and exit points have many dead registers. Interestingly, the average number of dead registers throughout the kernel (over every instruction) approximately splits the difference between the entry point and exit point averages. Functions that were not successfully parsed into control flow graphs are not included in these numbers.

Module	#save entry points	Avg. dead regs	#non-save entry points	Avg. dead regs	#exit points	Avg. dead regs	Avg. dead regs for every insn
ufs_log (Logging UFS Module)	106	1.2	25	2.6	229	13.5	5.8
xfb (xfb driver 1.2 Sep 7 1999 11:46:39)	86	1.0	13	2.2	287	14.7	6.6
audiocs (CS4231 audio driver)	68	2.2	15	2.6	157	13.3	9.0
dad (DAD Disk Driver 1.16)	48	2.1	8	1.2	190	14.1	6.4
tmpfs (filesystem for tmpfs)	55	1.0	11	2.5	168	14.4	6.0
ldterm (terminal line discipline)	34	1.1	11	2.5	151	12.8	7.1
afb (afb driver v1.36 Sep 7 1999 11:47:45)	53	2.3	7	2.9	103	14.3	7.3
scsi (SCSI Bus Utility Routines)	59	1.2	19	2.7	147	12.3	5.8
tl (TPI Local Transport Driver - tl)	46	1.4	11	2.9	207	14.0	7.0
specfs (filesystem for specfs)	40	1.3	8	2.9	115	13.9	5.6
arp (ARP Streams module)	54	1.6	13	2.1	181	13.6	7.7
SUNW,UltraSPARC-IIi	25	2.1	35	3.4	76	8.1	8.3
vol (Volume Management Driver, 1.85)	23	1.8	0	--	70	15.4	5.3
doorfs (doors)	39	0.9	13	2.3	128	13.9	5.5
su (su driver 1.24)	25	1.6	6	3.2	89	14.7	5.7
udp (UDP Streams module)	37	1.7	6	2.3	187	14.4	7.9
timod (transport interface str mod)	32	1.2	4	3.5	113	14.7	6.8
kerninst (kerninst driver v0.4.1)	72	1.7	44	2.0	191	11.8	8.9
fifofs (filesystem for fifo)	31	1.1	9	1.7	81	13.7	5.1
kb (streams module for keyboard)	25	0.6	11	2.6	147	13.0	9.0
tnf (kernel probes driver 1.47)	47	1.4	8	2.4	129	14.2	9.6
pm (power manager driver v1.65)	25	1.2	4	1.8	95	15.2	5.6
TS (time sharing sched class)	22	1.4	14	2.6	88	12.0	7.5
devinfo (DEVINFO Driver 1.24)	30	1.8	5	2.6	73	14.1	6.4
ipdcm (IP/Dialup v1.9)	44	0.9	8	1.9	117	13.9	6.8
ttcompat (alt ioctl calls)	8	1.5	6	4.0	40	12.4	12.8
diaudio (Generic Audio)	21	0.7	7	2.7	70	13.6	5.2
elfexec (exec module for elf)	9	1.0	3	4.3	19	13.4	6.5
shmsys (System V shared memory)	16	1.1	3	2.7	39	14.5	4.5
ptc (tty pseudo driver control ‘ptc’)	15	1.9	1	5.0	64	15.7	4.7
tlimod (KTLI misc module)	16	1.1	5	1.8	79	14.1	9.6
winlock (Winlock Driver v1.39)	32	1.4	7	2.1	80	13.8	7.5
hwc (streams module for hardware cursor support)	8	1.1	3	4.3	71	14.8	9.0
ms (streams module for mouse)	9	1.2	8	1.9	45	12.0	7.3
ptem (pty hardware emulator)	9	0.9	4	4.2	74	14.5	9.3
simba (SIMBA PCI to PCI bridge nexus driver)	14	1.1	4	2.0	46	14.5	9.4
seg_drv (Segment Device Driver v1.1)	16	1.4	9	2.1	55	13.2	5.2
sad (Streams Administrative driver’sad’)	16	1.2	7	3.7	74	13.3	7.4
namefs (filesystem for namefs)	26	1.2	6	2.7	50	13.5	5.6
lockstat (Lock Statistics)	18	1.2	9	2.7	59	11.9	8.7
ptsl (tty pseudo driver slave ‘ptsl’)	10	1.6	4	3.8	39	12.9	5.3
rootnex (sun4u root nexus)	14	1.3	5	2.4	63	14.1	9.3
dada (ATA Bus Utility Routines)	25	1.1	5	2.4	39	13.3	7.0
dada_ata (ATA Bus Utility Routines)	25	1.1	5	2.4	38	13.3	7.1
md5 (MD5 Message-Digest Algorithm)	3	1.3	5	3.0	9	7.8	5.3

Figure 3.7: Dead Registers in the Solaris 7 Kernel

Dead registers are available for scratch use in instrumentation code. Kerninstd’s interprocedural live register analysis shows that function entry points have few dead registers and exit points have many dead registers. Interestingly, the average number of dead registers throughout the kernel (over every instruction) approximately splits the difference between the entry point and exit point averages. Functions that were not successfully parsed into control flow graphs are not included in these numbers.

Module	#save entry points	Avg. dead regs	#non-save entry points	Avg. dead regs	#exit points	Avg. dead regs	Avg. dead regs for every insn
sysmsg (System message redirection (fanout) driver)	10	1.3	5	3.6	35	14.7	6.4
mm (memory driver)	7	1.6	6	3.3	28	11.8	5.6
wc (Workstation multiplexer Driver ‘wc’)	10	1.3	8	2.8	54	12.3	6.8
ebus (ebus nexus driver)	12	1.2	1	5.0	41	15.5	7.7
ptm (Master streams driver ‘ptm’)	8	1.1	4	3.8	27	12.1	3.8
pts (Slave Stream Pseudo Terminal driver ‘pts’)	8	1.4	4	3.8	29	12.3	4.0
RT (realtime scheduling class)	10	1.0	14	2.5	42	10.7	7.5
iwscn (Workstation Redirection driver ‘iwscn’)	13	1.1	6	2.5	32	11.1	5.8
fdfs (filesystem for fd)	11	1.3	6	2.7	29	12.8	9.4
eide (PC87415 Nexus driver v2.0)	15	1.5	4	3.2	40	14.4	9.3
conskbd (Console kbd Multiplexer driver ‘conskbd’)	10	1.0	4	2.2	43	12.3	9.6
todmostek (tod module for Mostek M48T59)	7	4.6	3	4.7	13	11.2	7.7
log (streams log driver)	5	0.8	4	3.5	25	12.6	5.9
sy (Indirect driver for tty ‘sy’)	8	1.6	4	3.5	28	13.9	5.4
consms (Mouse Driver for Sun ‘consms’)	10	1.0	4	2.2	38	11.9	9.1
kstat (kernel statistics driver)	7	0.9	5	3.2	33	11.5	5.0
pckt (pckt module)	6	0.5	5	3.2	23	12.7	6.9
ksyms (kernel symbols driver)	9	1.9	2	3.0	23	13.2	7.4
inst_sync (instance binding syscall)	8	1.0	3	4.3	14	12.5	7.2
power (power driver v1.4)	9	1.0	2	3.0	28	14.4	6.9
cn (Console redirection driver)	7	1.3	6	2.5	26	10.8	7.3
sysacct (acct(2) syscall)	2	1.0	4	3.0	12	10.8	5.3
clone (Clone Pseudodriver ‘clone’)	2	1.5	5	2.4	16	8.5	5.2
intpexec (exec mod for interp)	2	0.5	3	4.3	12	12.7	9.8
pseudo (nexus driver for ‘pseudo’)	2	1.0	8	2.1	21	7.7	7.0
ipc (common ipc code)	2	0.0	3	4.3	13	13.0	9.2
pipe (pipe(2) syscall)	1	1.0	3	3.0	4	5.5	4.7
connld (Streams-based pipes)	3	0.7	3	4.3	7	10.6	3.6
options (options driver)	1	2.0	6	2.0	14	3.3	3.9
redirmod (redirection module)	3	0.7	3	3.0	8	10.9	6.5
TS_DPTBL (Time sharing dispatch table)	0	--	5	2.8	5	1.8	2.5
IA (interactive scheduling class)	1	1.0	2	3.0	4	9.0	9.2
RT_DPTBL (realtime dispatch table)	0	--	3	3.3	3	2.3	2.9
platmod	0	--	5	0.6	5	0.6	0.6
Totals:	8147	1.2	2490	2.3	24708	13.2	7.1

Figure 3.7: Dead Registers in the Solaris 7 Kernel

Dead registers are available for scratch use in instrumentation code. Kerninstd’s interprocedural live register analysis shows that function entry points have few dead registers and exit points have many dead registers. Interestingly, the average number of dead registers throughout the kernel (over every instruction) approximately splits the difference between the entry point and exit point averages. Functions that were not successfully parsed into control flow graphs are not included in these numbers.

depending on how much code is written in C versus assembly. For compiled code, the number of dead registers also depends on the compiler used to generate the code, and at what optimization level it used. The results are summarized in Figure 3.7. The first column contains the module’s name and description, with the same ordering of

Figure 3.4 (by code size). Next is the number of functions having their own stack frame, and the average number of dead registers at the entry of such functions. The next columns give similar entry point information for functions that do not begin with a save. Next is information about the number of dead registers at exit points. For functions bracketed with save and restore instructions, an exit point usually has 15 dead registers: all %l and %o registers (the non-overlapping part of a register window), except for %o6, the stack pointer. For exit points that are optimized tail calls, the number of dead registers assumes the ability to de-optimize this sequence into a regular call and return sequence, which will be discussed in Chapter 4. The final column shows the average number of dead registers over every machine code instruction in the kernel.

The results contain a few surprises. First, there are few dead registers at the entry to functions that set up their own stack frame using save and restore (an average of 1.2). In such functions, the entry point is just *before* the save, which accesses a different register window than the rest of the function. Nevertheless, due to register window overlapping, some registers are usually dead before a save. In particular, any dead %o registers just after a save will appear as dead %i registers just before the save.

Entry points for functions that do not set up their own register windows have more dead registers than entry points functions with a save, but still few (an average of 2.3 dead registers instead of 1.2). The ratio of functions having their own register window to those that do not varies greatly; for afs, every function begins with a save (indicating compilation without optimization, a surprise for a commercial driver).

As expected, exit points usually have many registers free, though such a high kernel-wide average (13.2 dead registers) was surprising. The most pleasant surprise was in the final column, showing a kernel-wide average of 7.1 dead registers, which is more than enough to execute any of the code snippets generated by kperfmon. And finally, it is interesting to note that average number of dead registers through all kernel instructions roughly splits the difference between the average number of dead registers at function entries and exits. It is possible that the decrease in dead registers at an instruction is linear in the distance from an exit point. This would suggest that applications having some freedom where to place instrumentation code should choose to instrument as close to an exit point as possible. For example, an application instrumenting every basic block for code coverage may find more scratch registers when placing instrumentation at the ends of basic blocks, rather than at block entries.

The numbers presented for dead registers is conservative because kerninstd ignores assumptions that are normally allowed by an application binary interface (ABI), such as registers that are volatile across a function call. This conservative assumption can affect the result when a function does not make any reference to such a register. Because not all kernel code follows the SPARC ABI, such assumptions would be dangerous. It would be useful to strengthen kerninstd's analysis by identifying the majority of code that does follow the SPARC ABI. The existence of user-level ABI verification tools such as appcert [90] suggests that automated identification of the set of functions that follow an ABI is possible.

3.5.6 Live Register Analysis: What Could Not Be Analyzed

Live register analysis cannot be performed for any function that was not successfully parsed into a control flow graph (see Section 3.3.4). Several other routines could not be analyzed. Two functions, debug_flush_windows and flush_user_windows, flush the SPARC register window contents to the stack. These functions cannot be analyzed because they perform save and restore instructions in a loop, making a fixed number of window levels in its dataflow functions impossible to set. A few other routines (less than 10) do not analyze correctly because the if-taken and if-not-taken successors to a conditional branch have dataflow functions with a disagreeing number of window levels. This presently occurs in some assembly language functions that call panic; the call is preceded with a save instruction, but with knowledge that panic never returns, a corresponding restore was intentionally omitted. This omission causes kerninstd to identify this part of the function with a dataflow function having a save without a corresponding restore, causing confusion when this dataflow function is later merged with code having a balanced save and restore count.

A function that is successfully parsed into a CFG but failed in live register analysis is still instrumentable, under the conservative assumption that all registers are live throughout the function. (These functions are included in Figure 3.7, but since they are few, they do not significantly alter the averages.)

3.6 Summary and Research Contributions

This chapter has introduced the KernInst system structure, and described the actions that are performed by kerninstd at start-up. The unique characteristic of KernInst described in this chapter is the ability to load a performance tool onto an unmodified commodity kernel at run-time. KernInst startup has three components: allocating the code patch heap, obtaining a structural analysis of the kernel's code, and obtaining the ability to write anywhere in the kernel's address space. After start-up, kerninstd is able to perform run-time kernel splicing, the subject of the next chapter.

Chapter 4

Fine-Grained Code Splicing and Code Replacement

This chapter presents the design and implementation of the two dynamic kernel instrumentation primitives in KernInst: *fine-grained code splicing*, which inserts arbitrary instrumentation code at machine code instruction granularity, and *code replacement*, which installs a new version of a function.

Splicing overwrites one machine code instruction at the desired *instrumentation point* with a jump to a *code patch.* In the simplest case, the code patch contains the instrumentation code generated at run-time, the overwritten instruction, and a jump back to the instruction following the instrumentation point, as shown in Figure 4.1. The net effect inserts the generated code before a desired machine code instruction.

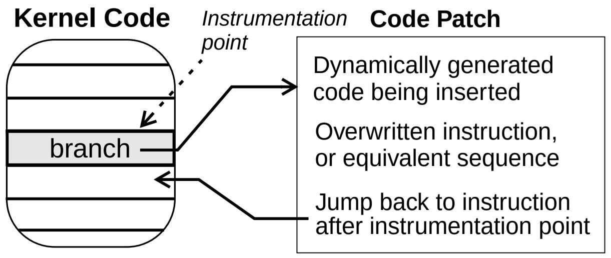

Kernel Code
Instrumentation point
Code Patch
branch
Dynamically generated code being inserted
Overwritten instruction, or equivalent sequence
Jump back to instruction after instrumentation point

Figure 4.1: Code Splicing

One machine code instruction is overwritten with a branch to patch code, which contains the desired instrumentation code, the overwritten instruction, and a branch back to the instruction stream.

Code replacement instruments the entry point of a specified function to jump to the new version of the function. To reduce run-time overhead, code replacement also alters statically identifiable call sites to the function so they directly call the new version, avoiding the overhead of the inserted jump.

An important feature of KernInst is that its splicing and code replacement mechanisms are independent of code generation. KernInst is general enough to splice in, or use as a replacement, machine code that has been created from code generation packages such as VCODE [33], dynamic compilers [4, 55], an interpreter performing just-in-time compilation, or precompiled position-independent code.

A second important feature of KernInst's instrumentation mechanisms is that they do not require kernel source code. A splicing request is some machine code and the kernel code address where it should be inserted. A code replacement request is some machine code and the name of the function it should replace. A particular application that generates the machine code may benefit from access to the source code, but placing that code into the kernel via splicing or replacement does not require source code.

The remainder of this chapter discusses how KernInst performs fine-grained run-time instrumentation and code replacement of a Solaris 7 kernel running on an UltraSPARC processor. Sections 4.1 through 4.4 describe code splicing, and Section 4.5 describes code replacement. Section 4.6 discusses future work, including porting these primitives to other platforms. Section 4.7 summarizes the chapter, including its research contributions.

4.1 Splicing: Basic Code Patch Issues

This section presents some basic issues regarding the allocation of the code patch and filling it with desired instrumentation code. Section 4.2 discusses relocating the overwritten instruction at the instrumentation point into the code patch.

Kerninstd de-couples splicing from generation of instrumentation code. Kerninstd is an instrumentation server, accepting requests from instrumentation clients to splice machine code into the kernel at a particular instrumentation point. Before such a request is made, however, the instrumentation client is likely to query the set of dead registers at the instrumentation point; such registers can be used in the instrumentation code. The instrumentation client then generates two machine code version of the instrumentation code. The first version uses only the available scratch registers, if there are enough (otherwise, it will contain save and restore instructions to free some registers). A second version, to be used in case kerninstd decides to bracket the instrumentation code with save and restore instructions at its own behest, must also be generated. Kerninstd may use the second version because there can be multiple snippets of instrumentation code at a particular instruction, which as a whole may require more scratch registers than are available.

Because kerninstd may insert save and restore instructions into the code patch, it is unsafe to instrument some of the kernel’s trap handling code. In the register-window-overflow trap handler, inserting a save would cause an infinite loop. In other trap handlers, a save can cause a trap that can cause a panic by raising the number of nested traps above the processor’s maximum.

To splice, kerninstd must first allocate a code patch. The code patch size is the total size of the instrumentation snippet(s) presently installed at this point, plus room for the save and restore instructions if needed, space for the original instruction at the instrumentation point, and space to jump back to the instruction after the instrumentation point. For the returning jump, there are several possible code sequences, with varying numbers of instructions, depending on the required displacement and whether there is a scratch register available. Thus, the number of instruction bytes required for the code patch cannot be precisely determined until it has been allocated. The circular dependency is broken by assuming the maximum number of instructions needed to perform a jump (4 for a 32-bit kernel, 6 for a 64-bit kernel). Calculating the patch size usually takes about 20 µs, and allocating it typically takes 4 µs.

Once patch space has been allocated, kerninstd prepares to write to it by calling mmap on /dev/kmem, which typically takes 40 µs. The first write to mapped memory typically takes another 15 µs, due to the kernel's policy of deferring allocation of a physical backing page until needed. Although Solaris 7 cannot directly write to a code patch allocated within the nucleus via /dev/kmem (see Section 3.2), writing to it indirectly via mmap works. When emitting completes, the corresponding munmap takes about 20 µs.

The code patch begins with instrumentation code that was downloaded from a client. Although the client sends relocatable code (roughly similar to an ELF object file), emitting could not take place before code patch allocation, because the final representation of some instructions (such as SPARC calls) depends in part on their location in memory.

4.2 Splicing: Relocating the Overwritten Instruction to the Code Patch

The code patch ends with a relocated version of the instruction that was overwritten by the splicing branch, or a semantically equivalent sequence of several instructions. This section discusses the key concepts of relocation, and then demonstrates relocation of some of the more interesting cases. A key conclusion is that relocation is always possible; every SPARC instruction can be relocated to the code patch either verbatim, or with a semantic equivalent sequence of instructions.

An overview of relocation is presented in Section 4.2.1. Section 4.2.2 discusses the relocation of optimized tail calls (see Section 3.3.2), showing how kerninstd is able to instrument a point that previously did not even exist: after the callee returns, but before its caller returns. Section 4.2.3 discusses relocation of conditional branches, as an example of allowing instrumentation to be placed along control flow graph edges. Section 4.2.4 discusses the challenging case of instrumenting at delay slot instructions.

4.2.1 Basic Concept: Maintain Semantics

The relocated code must maintain the semantics of the overwritten instruction. In the simple case of most non-control-flow instructions, the instruction can be relocated verbatim to the code patch, and only needs to be followed with a code sequence to jump back to the instruction following the instrumentation point. Instructions that transfer control unconditionally, such as return, need not be followed by such a jump.

Maintaining semantics when overwriting a control transfer instruction having a delay slot generally requires relocating the delay slot instruction to the code patch as well. As before, only one instruction is overwritten during splicing. The original delay slot instruction, left in place, will no longer be executed. If the control transfer's delay slot instruction was itself a delayed control-transfer instruction, relocation would be much more complex. Fortunately, such a code pattern does not exist in the kernel.

Relocation often requires subtle changes when the semantics of the overwritten instruction depend on its location in memory. Consider an instruction that copies the address contained in the program counter to another register. To maintain the original semantics, the relocated version of this instruction must put the address of the instrumentation point into the destination register. Similarly, the SPARC jump-and-link instruction copies the program counter to a specified destination register before jumping. The relocated version maintains semantics with several instructions. First, the address of the instrumentation point is copied to the specified register (the "link" part of the jump-and-link). Then, the original jump is performed, though modified to ignore the link. Similarly, the SPARC call instruction always writes the program counter to %o7. The equivalent relocated sequence writes the address of the instrumentation point to %o7, then unconditionally jumps to the callee.

4.2.2 Creating An Instrumentation Point Where None Existed: Relocating Tail Calls

The exit point of tail call is the most technically challenging to splice, and the most useful. As discussed in Section 3.3.2, a tail call sequence causes the callee to return—

not to its caller, but to where its caller would itself normally return. Semantically, it performs a call then a return, but there is no place in the code to splice instrumentation after the callee returns but before the caller returns.

This section discusses relocating the tail call sequence in a way that allows instrumentation to be placed after the callee has returned. Examples in this section assume the most common SPARC tail call sequence: call; restore. Unwinding other tail call sequences are handled similarly.

A tail call can be instrumented before it returns by first de-optimizing it into a semantically equivalent call then return. Figure 4.2 shows the original code and the de-optimized version that will be placed in a code patch.

Original Tail Call	Equivalent Unwound Code

// arguments are in %i regs
 	
call C
restore reg1, reg2, reg3 // delay slot
 	
// Restore also adds reg1 and reg2
// (from the current register window)
// to reg3 (in the post-restore register
// window).
 	
// In the kernel, reg3 is always either
// %g0 (no add is done), or an %o
// register.
	// arguments are in %i regs
	
	add reg1, reg2, reg3'
	// reg3'=reg3, but changed from an %o to an %i register
	// Duplicates the part of the restore that did an add
	
	// Set up args for C:
	copy %i0-i5 to %o0-o5
	
	call C
	nop
	
	// Copy back result from C, if any
	copy %o0-o5 to %i0-i5
	
	Exit point instrumentation, if any, goes here. The ability to place instrumentation here did not exist in the original code.
	
	ret
	restore // plain restore, with no add

Figure 4.2: Tail Call Unwinding to Allow Exit Point Instrumentation

The original call; restore sequence is relocated to the code patch in an unwound, de-optimized, and semantically equivalent sequence that allows instrumentation code (in bold) to be placed just before this sequence returns. The ability to instrument at the exit point did not exist in the original code.

Tail call unwinding is also performed by Paradyn [44], which does it in the following order: (1) execute a restore instruction to unwind the caller's stack frame, (2) save the link register %o7 onto the stack, (3) do a normal call, (4) execute desired exit point instrumentation, (5) restore the link register from the stack, and (6) return. Paradyn's algorithm is attractive because, like the original code, the restore is performed before the call, using one less register window frame than KernInst's method. Unfortunately, the stack location used to save the link register cannot safely be used as scratch storage; it is space reserved for the operating system to spill a register window frame, if needed. The latest version of Paradyn uses the same unwinding sequence as KernInst.

4.2.3 Relocating Conditional Branches

SPARC conditional branch instructions are an interesting case to instrument. The user can request that instrumentation code be executed in three cases: if the branch was taken, if the branch was not taken, or both. The first two cases are an example of inserting instrumentation code along control flow graph edges. Branch instructions are also challenging to relocate because they have a PC-relative displacement. To maintain the same if-taken address after relocation, the instruction's displacement must be changed. But because the modified displacement may not fit in the displacement field, branches may need to be relocated as a semantically-equivalent sequence of several instructions, as shown in Figure 4.3.

Code Patch Contents
Instrumentation to be executed whether or not branch is taken, if any, goes here
branch <condCheck><condCode> to "if-taken"
nop // delay slot
if-not-taken:
Instrumentation to be executed only if the branch is not taken, if any, goes here
relocated delay slot instruction, if the branch had a delay slot that is executed on fall-through.
jump to instrumentation point address+8
if-taken:
Instrumentation to be executed only if the branch is taken, if any, goes here
relocated delay slot instruction, if the branch had a delay slot that is executed on if-taken.
jump to destination of the overwritten branch

Figure 4.3: Relocated Conditional Branch Instruction (Most General Case)

Note that the relocated version of the branch enables three possible places to drop in instrumentation: when the branch is taken, when the branch is not taken, or both. The first two places did not exist in the original code. Depending on an "annul" bit in the branch instruction, a delay slot instruction may exist; if so, it can be executed always, only when the branch is taken, or never. The relocated sequence allows each of these possibilities. Finally, note that above sequence can often be optimized, most obviously for an unconditional branch.

4.2.4 Relocating a Delay Slot Instruction

Although in the Solaris kernel, no delay slot instruction is itself a control-transfer instruction (see Section 4.2.1), splicing at a delay slot is problematic. Figure 4.4 illustrates the difficulty, where a splice instruction (ba,a) is placed in the delay slot of a call. Because the ba,a is itself a delayed control transfer instruction (even though it has no delay slot), one instruction of the callee will execute before the ba,a is able to transfer control to the code patch.

Original Code	Code After Splicing	Executed Instructions
...	...	0x1000
0x1000: call bar (at 0x2000)	0x1000: call bar (at 0x2000)	0x1004
0x1004: add %g0, 1, %o0	0x1004: ba,a to 0x3000	0x2000 first instruction of bar
...	...	0x3000 finally, the ba,a to the
		code patch takes effect
	code patch:	
	0x3000:...	

Figure 4.4: Why Splicing at a Delay Slot Cannot Work

The effect of replacing the add instruction at 0x1004, which is the delay slot of the call to bar, with a splice instruction to a code patch, is undesirable. Because the splice instruction (ba,a) is itself a delayed instruction, it will not have a chance to transfer control until exactly one instruction of bar has executed.

Splicing at a delay slot cannot be safety performed, but the desired effect can be achieved by splicing at the delay slot’s control transfer instruction. In each of the control transfer instructions discussed above, there is an opportunity to place instrumentation code before the relocated delay slot instruction. Although this feature is not yet implemented in KernInst, it is not difficult to do. The most challenging case is splicing instrumentation code before a delay slot instruction that is conditionally executed, which occurs when the control transfer instruction is a conditional branch with the annul bit set. Here, the delay slot instruction will only execute if the branch is taken, and as such, maintaining semantics requires instrumentation code that is only executed only when the branch is taken. Because relocation of conditional branches (Section 4.2.3) has if-taken and if-not-taken code portions, the solution is straightforward.

The above technique will not work when the delay slot is also the destination of a branch. The difficulty is in knowing from which control transfer instruction this instruction was reached. There are at least two possibilities: the parent control transfer instruction, and one (or more) branches that reach the delay slot instruction. The path taken to this instruction dictates where control should be resumed at the end of the code patch. If the instruction were reached as a delay slot of its parent control transfer instruction, then it should be handled as described earlier in this section. Otherwise, it should be handled be falling through to the next instruction. Because of this ambiguity, splicing cannot take place at a delay slot instruction that is also the start of a basic block. As mentioned in Section 3.3.2, this pattern is found only four times in the Solaris kernel.

4.3 Splicing: Jumping to the Code Patch

Once a code patch has been emitted, the final step is to overwrite the existing code at the instrumentation point with a jump or branch instruction that reaches the code patch. Similarly, un-splicing takes place by putting the original instruction at the instrumentation point back in place. This step is challenging for several reasons. First, the kernel is running while the splicing is taking place, and cannot be paused. Code may be executing at or near the instrumentation point while splicing occurs, raising the possibility of a dangerous race condition. Second, the Solaris kernel is multi-threaded, increasing the chances of a dangerous race condition. Third, we do not assume that the original code at the instrumentation point has in any way been modified *a priori* to facilitate splicing.

Splicing (and un-splicing) a multi-threaded kernel without pausing is accomplished by ensuring that *only one* instruction is replaced at the instrumentation point. When splicing, the instruction will be a branch to the code patch. When un-splicing, the instruction will be the original, overwritten instruction at the instrumentation point. This section motivates the single-instruction-write rule, notes a technical problem that arises, and discusses two solutions.

4.3.1 Avoiding Hazards During Splicing

For safety, kerninstd always splices by overwriting a single instruction at the instrumentation point with a branch to the code patch. It will take time for the new instruction to take effect; at a minimum, it must make its way from the data cache, to the unified L2 cache, and to the instruction cache, before it can be brought into the pipeline. The delay does not cause any hazardous race conditions; it only means that the precise time when the splice takes effect is unknown. Until it takes effect, any kernel thread executing at the instrumentation point will fetch and safely execute the original pre-splice code. Once the splice takes effect, a kernel thread will fetch and execute the post-splice code sequence. Since it is not possible to execute a mix of the pre-splice and post-splice code sequences, single-instruction splicing is hazard-free.

Atomic splicing is necessary but insufficient to ensure safe execution. For example, even though the SPARC can atomically overwrite two instructions at the instrumentation point, a thread can sometimes execute a mix of pre-splice and post-splice sequences in this case, as shown in Figure 4.5.

Before Splicing	After Splicing	Executed Instructions
Original 1 Original 2 ← PC is here	New 1 New 2 ← PC is (still) here	Original 1, New 2

Figure 4.5: Atomic Splicing is Not Enough

Splicing is hazardous whenever the program counter has executed some, but not all, of the instructions that are overwritten. As soon as execution continues, a mix of some of the pre-splice and some of the post-splice sequences is executed.

The dyninstAPI [13] and Paradyn [43, 44] user-level tools avoid this problem by pausing the program and examining a back-trace of all threads to check whether any are currently within, or will return to, any but the first of the instructions being overwritten. If a hazard is detected, then splicing is deferred and retried later. However, this technique cannot work in a kernel, for several reasons. First, pausing the kernel is not allowed; even if it were, it could disrupt potentially critical background activities. Second, performing the necessary back-trace on all threads would be very expensive. Third, even if pausing were possible and inexpensive, a jump with an unanalyzable destination (such as a longjmp) can find a way to reach the middle of the overwritten instruction sequence, resulting in the same hazard. A fourth problem with multiple-instruction splicing is that it is hard to make it fine-grained. In particular, if the instrumentation point is the last instruction of some basic block B_1, then some of the splice sequence will spill over into the next basic block B_2. If B_2 is reachable from another basic block B_3, then code executing along the path (B_3, B_2), will execute an inconsistent instruction sequence which can crash the kernel.

When splicing, kerninstd overwrites a single instruction by having /dev/kerninst perform an *undoable write*. An undoable write overwrites one instruction in kernel memory. Additionally, it logs information about the change (address and original content), so it can undo the effects of the write in an emergency. In KernInst, such an emergency situation occurs if kerninstd crashes. Without undoable writes, branch instructions used for splicing will be left in place after a kerninstd crash, likely causing a panic when module unloading is re-enabled and the code patch heap is de-allocated. An undoable write is not used when writing to uninitialized allocated memory, or when writing to a location for which undoable write state is already being kept. The latter case occurs when instrumentation at a particular site is changed; the branch to the old code patch is replaced with a branch to the new code patch, and undoable write state is not updated. An undoable write takes about 30 µs when writing to the nucleus, and about 8 µs otherwise. Undoing an undoable write takes about 25 µs when in the nucleus, and about 7 µs otherwise.

4.3.2 The Challenge: Insufficient Displacement to Reach Code Patch

We have seen that safety requires single-instruction splicing, but architectures do not provide an ideal instruction to branch from any instrumentation point to a code patch. An ideal splicing instruction is one which:

- Has enough displacement to reach the code patch from the instrumentation point.
- Does not have a delay slot, which causes the next instruction to be implicitly executed before the code patch is reached.
- Has no side effects other than changing the program counter.
- Jumps to a destination address that is fixed or PC-relative, but not register-relative. A register relative jump instruction can have arbitrary range, but requires the value of a register to be set beforehand, thus leading to unsafe multiple-instruction splicing.
- Is small enough to ensure that only one instruction is overwritten. (Trivially true for any RISC architecture, but a concern for x86.)

Figure 4.6 reviews, for several architectures, the branch and jump instructions that are best suited for single-instruction splicing. Unfortunately, none of the architectures has an instruction that is always suitable. The key limitation is displacement. The patch area heap may be allocated arbitrarily far from the code of most kernel modules. Thus, splicing needs a means for reaching a code patch, no matter the required displacement, while still splicing with a single instruction, for safety. The remainder of this section discusses two solutions to address this reach problem.

4.3.3 Solving the Reach Problem: Springboards

KernInst’s first and most general solution to the displacement problem is called *springboards*. A springboard is a scratch area that is reachable from the instrumentation point by a suitable (single) splicing instruction. The idea is for the

Arch	Instruction	Range	Delay Slot?	Side Effects	Always Overwrites 1 Instruction?
SPARC v9	call	PC ± 2 GB	yes	writes to *%o7*	yes
	ba,a	PC ± 8 MB	no	none	yes
	jump	register ± 16 K	yes	none	yes
PowerPC	b	PC ± 32 MB	no	none	yes
MIPS IV	j	current 256 MB aligned region	yes	none	yes
	b<cond>	PC ± 128K	yes	none	yes
Alpha	branch	PC ± 4 MB	no	none	yes
	jmp	register ± 16K	no	none	yes
x86	jmp	PC ± 2 GB	no	none	no

Figure 4.6: Suitability of Various Instructions for Single-Instruction Splicing.
None of the architectures has an ideal splicing instruction; either displacement is insufficient (RISC architectures), or there is no guarantee that only a single instruction is overwritten when splicing (x86)..

splicing instruction to branch to any unused, reachable springboard, which then performs a long jump to the code patch, using as many instructions as needed. Figure 4.7 diagrams code splicing in the presence of springboards.

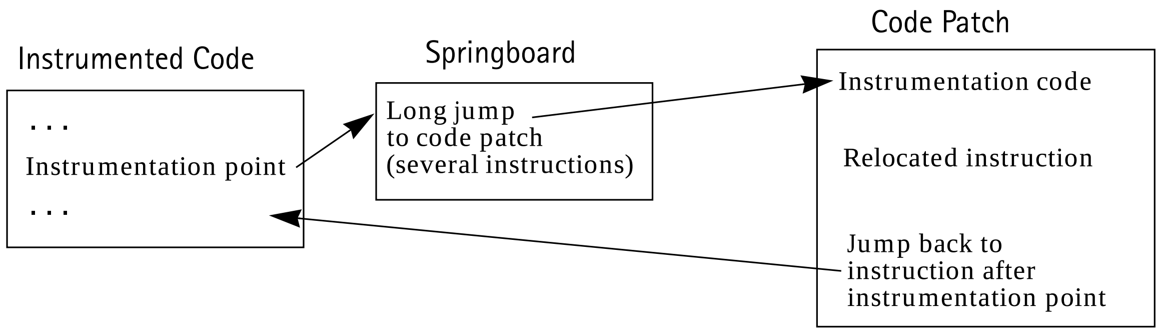


Figure 4.7: Using Springboards to Reach the Code Patch
A level of indirection is used to reach the code patch from the instrumentation point. When finished, the code patch returns directly to the instrumentation point.

Like the code patch, the springboard is written and flushed to the instruction cache before performing the final step of writing the splice instruction at the instrumentation point. The ordering ensures that no kernel thread will begin executing springboard code until the splice has taken effect. Furthermore, since the springboard was originally scratch space, it is safe to assume that no thread was already within the springboard when the splice takes place. Thus, although the springboard contains multiple instructions, there is no hazard similar to that of replacing more than one instruction at the instrumentation point.

The springboard approach requires chunks of scratch space (collectively, the *springboard heap*) to be conveniently sprinkled throughout the kernel, so that every kernel instruction can reach some chunk when using one of the suitable instructions listed in Figure 4.6. Fortunately, UNIX System V-based kernels (such as Solaris), Linux, and Windows NT all have ideally suited space: the initialization and termination routines for dynamically loaded kernel modules.

In a kernel that allows modules to be loaded at run-time (and unloaded, if memory becomes tight), each module has initialization and termination routines that are called just after the module is loaded, and just before the module is unloaded, respectively. In Solaris, these routines are called _init and _fini. In the UNIX SVR4.2 standard, the routines are called <module>_load and <module>_unload [101]. In Linux, they are called init_<module> and cleanup_<module>. In Windows NT, device drivers have a DriverEntry routine which also installs a pointer to a cleanup routine.

Kerninstd can effectively take over the module initialization and termination routines, making them available for adding to the springboard heap, after locking the modules in memory to ensure that these routines are not called. On Solaris, this locking is done by /dev/kerninst with a call to mod_unload_disable. Preventing module unloading and re-loading also obviates the need to redo splicing that would implicitly be undone when and if a module was unloaded and later re-loaded. In practice, no kernel module approaches one megabyte in size (see Figure 3.4), so even a jump instruction with modest displacement, such as the SPARC ba,a, can easily reach the nearest springboard.

In Solaris, a module does not need to supply a _fini routine; such a module will never be unloaded. Furthermore, the core kernel modules unix, krtld, and genunix do not have an _init or a _fini routine. Fortunately, these modules are always loaded contiguously in the kernel and thus can be considered one large module for purposes of having enough displacement to reach a springboard; any solution that works for one of these modules will work for all of them. In this case, KernInst takes over several routines from the unix module that are only used during kernel booting: _start and main. (There are certainly other such routines, but there has not yet been a need for more springboard space than these routines provide.)

General Location	Springboard space (bytes)
Within kernel nucleus	73,920
Outside of kernel nucleus	15,756
Total	89,676

Figure 4.8: Springboard Space in Solaris 7

Since most kernel modules are located in the nucleus, chances are that nucleus springboard space is more useful than non-nucleus springboard space. Nucleus space is gathered in three ways: (1) routines only used during boot-time (_start and main), (2) 65K of space manually compiled into /dev/kerninst on the assumption the driver will be loaded into the nucleus, and (3) the _init and _fini routines of kernel modules in the nucleus. Non-nucleus space is gathered in two ways: (1) 15K of memory allocated with kmem_alloc, and (2) the _init and _fini routines of kernel modules outside the nucleus.

Figure 4.8 summarizes the springboard space that is set aside in the current version of KernInst, broken down into nucleus and outside-of-nucleus components. The nucleus component contains extra springboard space that is manually compiled into the code space of /dev/kerninst, on the assumption that this driver, like most modules, will be loaded into the nucleus.

If a springboard is needed, kerninstd searches for one that is within range, and (if possible) resides in a different I-cache line than both the instrumentation point and the start of the code patch. The allocation takes about 8 µs. Kerninstd then fills the springboards contents with three undoable writes (see Section 4.3.1), which takes about 100 µs if the springboard resided in the nucleus, and 24 µs otherwise.

While the springboard technique may seem ad-hoc, it is general enough to apply to most kernels. Furthermore, with 64-bit operating systems, the code patch heap (allocated in kernel data space) tends to be allocated even farther away from code than with 32-bit systems, exacerbating the reach problem and further motivating the need for a general solution like springboards.

4.3.4 Solving the Reach Problem: In-Nucleus Allocation

A second solution to the reach problem avoids it—by allocating code patches sufficiently close to their respective instrumentation points. Although this technique is not as general as springboards, by using /dev/kerninst to access the kernel's internal memory allocation routines, it is often feasible.

As discussed in Section 3.2, when running on an UltraSPARC processor, the Solaris kernel tries to place all kernel code within a 2 MB area known as the code nucleus. Any kernel module whose code does not completely fit in the nucleus is allocated elsewhere in the data space. Even when the nucleus space has been depleted, in the sense that at least one module's code could not entirely fit within it, there may be leftover space. If large enough, this free space could be used for code patches; if not, it could still be used for springboards. Presently, kerninstd attempts to use leftover nucleus space for the code patch heap. Allocation of one page (8 K) out of the nucleus text, which is performed by /dev/kerninst, takes about 1400 µs. However, there is a much lower amortized cost per instrumentation request, because this allocation is only performed when the code patch heap has run out of space and needs to grow.

4.4 Un-splicing

An instrumentation client that downloads code into the kernel can ask kerninstd to remove it at any time. For kerninstd, there are two cases of un-splicing. The first occurs when no instrumentation remains at this site. Kerninstd un-splices at the instrumentation point by replacing the splicing branch instruction with the original, overwritten instruction. In addition, kerninstd clears the undoable write state for this location (see Section 4.3.1). The operation takes about 65 µs if the instruction was in the nucleus and about 40 µs otherwise. If a springboard was in place, then an additional 80 µs is needed if the springboard resided in the nucleus, and 42 µs if the springboard resided outside of the nucleus. For safety reasons that are detailed shortly, removing the springboard is deferred to give any thread(s) that may be executing within the springboard a chance to leave it before the springboard is freed.

The second case of un-splicing is when some instrumentation remains at the site, for which kerninstd splices an entirely new code patch that contains the remaining instrumentation snippets. No time is spent patching the kernel to un-splice the old code, as in the above paragraph. Because undoable write information is already being kept for this point, this splice is effected by simply writing a new branch instruction at the instrumentation point. If necessary, a new springboard is used as well. The cost is, in fact, slightly less than if splicing at a site that was previously uninstrumented, because no undoable write state needs to be updated. The operation completes in about 43 µs for an instrumentation point in the nucleus, and about 23 µs for an instrumentation point outside of the nucleus. If a springboard is required for the new code patch, it is allocated and initialized with the same costs as splicing a previously uninstrumented site.

In both cases, de-instrumentation leaves behind a code patch, and perhaps a springboard, that must eventually be garbage collected. Safety is a concern, because some kernel thread(s) may presently be executing within, or context switched out while within a springboard or code patch. It may not be safe to return these objects to their respective heaps right away, where they could be re-allocated and (most importantly) re-written before all kernel threads have exited these code structures.

If a springboard or code patch contains no code that blocks, it is unlikely that any thread executing in its midst will remain there long, and simply delaying for a few milliseconds before returning the structure to its free heap may be safe. This is certainly the case for a springboard, which does not contain any instructions that block. The code patch is a greater concern for two reasons. First, instrumentation code may block (if it performs I/O, for example). Or, the relocated instruction in the code patch could block; for example, it may be a call to mutex_enter. The solution presently employed by kerninstd is to wait three seconds before returning the springboard and code patch to their respective heaps, hoping the delay is sufficient to drain any remaining threads from these structures. While it has proven sufficient in practice thus far, the solution is unsatisfying.

A safer approach (that is not yet implemented) is to bracket the entry and exit of the code patch with a thread-safe counter increment and decrement. The counter then reflects the number of kernel threads presently within the code patch; when zero, de-allocation is safe. There are three concerns with this approach, however. First, it adds run-time overhead. Second, the shared counter could cause contention in a multiprocessor. Third, it is not entirely safe. Code that increments the counter cannot take effect until a couple of instructions have executed. Similarly, code that decrements the counter cannot be the last item in the code patch, which is usually a jump back to the instruction following the instrumentation point. Fortunately, it is possible to augment this technique with a delay, to achieve safety. The instructions that may be in execution when exiting a code patch do not block, so a large delay (e.g., one second) should be sufficient to ensure that such code completes. Similarly, the instructions that may be in execution at the start of the code patch, before the counter has been incremented from 0 to 1, also do not block. If the counter is checked twice, with a sufficient delay in between, we can be confident that execution is not within the dangerous instructions.

4.5 Code Replacement

Run-time kernel code replacement is logically distinct from, and complementary to, code splicing. Whereas code splicing inserts code into an otherwise unmodified function, code replacement installs an entirely different version of a function. Note that the replacement function is recognized as any other kernel function by KernInst; the function can be subsequently spliced or replaced. The low-level implementation of code replacement uses many of the same mechanisms used by code splicing, requiring only minor additions to kerninstd to implement.

4.5.1 Installing A Replacement Function

Kerninstd replaces a function by instrumenting the entry point of the original code to unconditionally jump to the new code, as shown in Figure 4.9. No code patch is needed; the jump code resides entirely within the splicing branch instruction if the new version of the function is reachable from the old version with a single branch, or within a springboard otherwise. Code replacements takes about 68 µs if the original function resides in the kernel nucleus, and about 38 µs otherwise. If a springboard is required, then a further 170 µs is required if the springboard resides in the nucleus, and 120 µs for a non-nucleus springboard.

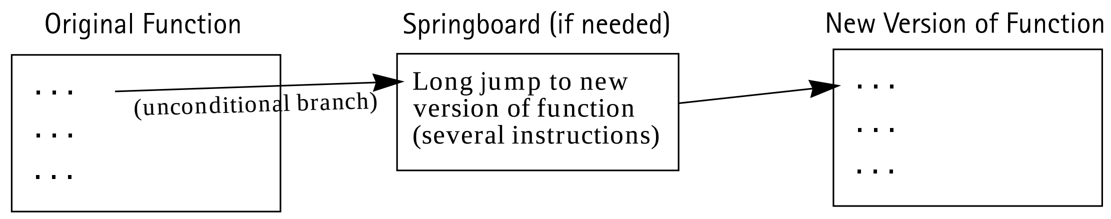


Figure 4.9: Basic Code Replacement
The entry point instruction of the original function is replaced with an unconditional non-delayed branch to the new version of the function. A springboard is used if needed.

The above framework introduces run-time overhead for each call to the function being replaced, which often can be optimized by patching the function’s call sites to directly call the new version of the function. This optimization can be applied for all statically identifiable call sites where the displacement needed to reach the new version of the function is within range of a call instruction (± 2^{31} bytes on the SPARC). Because the optimization cannot be applied to calls with an unanalyzable destination, such as calls through a function pointer, call site optimizations must be applied in addition to, not instead of, the basic framework of Figure 4.9.

The additional cost of optimized code replacement can be quite high, because /dev/kerninst performs an expensive undoable write (see Section 4.3.1) for each call site. Replacing one call site takes about 36 µs if the call site is in the nucleus, and about 18 µs if the call site lies outside of the nucleus. To give a large-scale example, replacing the function kmem_alloc, including patching of its 466 call sites, takes about 14 ms. To help gauge the expected number of call sites, Figure 4.10 summarizes the number of statically identifiable calls to kernel functions that were successfully parsed. Each function in the kernel is called an average of 5.9 times, with a standard deviation of 0.8. The function mutex_exit is called the most: 6200 times.

Module	#Functions Parsed	Average # of calls made to a function in this module	Max # of calls made to a function in this module
genunix	2589	9.2	1438
afs (afs syscall interface)	922	4.1	364
unix	1758	13.2	6200
ufs (filesystem for ufs)	337	2.8	42
nfs (NFS syscall, client, and common)	479	2.4	79
ip (IP Streams module)	373	3.9	86
md (Meta disk base module)	390	3.3	80
tcp (TCP Streams module)	159	2.5	34
procfs (filesystem for proc)	174	2.8	140
sd (SCSI Disk Driver 1.308)	115	2.1	33

Figure 4.10: Number of Statically Identifiable Calls to Kernel Functions

Only functions that are successfully parsed are included. There is wide variance between modules in how often its functions are called. The functions in module unix are called on average 13.2 times, compared to the overall average of 5.9 calls. The standard deviation is 0.8 calls. The function mutex_exit is called most often (6200 times). A few small modules have none of their functions (directly) called.

Module	#Functions Parsed	Average # of calls made to a function in this module	Max # of calls made to a function in this module
rpcmod (RPC syscall)	209	2.4	67
sockfs (filesystem for sockfs)	149	2.7	48
pci (PCI Bus nexus driver)	127	0.6	6
hme (FEPS Ethernet Driver v1.121)	97	3.7	42
se (Siemens SAB 82532 ESCC2 1.93)	69	1.7	18
fd (Floppy Driver v1.102)	54	2.5	22
zs (Z8530 serial driver V4.120)	48	1.6	17
uata (ATA AT-bus attachment disk controller Driver)	127	1.6	18
krtld	127	3.8	48
rpcsec (kernel RPC security module.)	122	1.1	11
ufs_log (Logging UFS Module)	131	1.6	10
xfb (xfb driver 1.2 Sep 7 1999 11:46:39)	99	1.6	11
audiocs (CS4231 audio driver)	83	1.3	14
dad (DAD Disk Driver 1.16)	56	1.1	6
tmpfs (filesystem for tmpfs)	66	1.3	15
ldterm (terminal line discipline)	45	3.1	14
afb (afb driver v1.36 Sep 7 1999 11:47:45)	60	1.2	28
scsi (SCSI Bus Utility Routines)	78	2.9	61
tl (TPI Local Transport Driver - tl)	57	2.7	30
specfs (filesystem for specfs)	48	1.2	9
arp (ARP Streams module)	67	1.5	12
SUNW,UltraSPARC-IIi	60	2.3	40
vol (Volume Management Driver, 1.85)	23	1.9	20
doorfs (doors)	52	1.4	9
su (su driver 1.24)	31	1.2	9
udp (UDP Streams module)	43	1.2	21
timod (transport interface str mod)	36	3.9	41
kerninst (kerninst driver v0.4.1)	116	1.4	9
fifofs (filesystem for fifo)	40	1.0	6
kb (streams module for keyboard)	36	2.2	9
tnf (kernel probes driver 1.47)	55	1.6	20
pm (power manager driver v1.65)	29	1.9	14
TS (time sharing sched class)	36	0.3	6
devinfo (DEVINFO Driver 1.24)	35	2.7	33
ipdcm (IP/Dialup v1.9)	52	0.7	7
ttcompat (alt ioctl calls)	14	1.6	10
diaudio (Generic Audio)	28	2.2	10
elfexec (exec module for elf)	12	2.4	21
shmsys (System V shared memory)	19	1.1	3
ptc (tty pseudo driver control ‘ptc’)	16	0.4	3
tlimod (KTLI misc module)	21	3.0	16
winlock (Winlock Driver v1.39)	39	1.3	8
hwc (streams module for hardware cursor support)	11	0.6	3
ms (streams module for mouse)	17	0.7	6
ptem (pty hardware emulator)	13	3.9	19
simba (SIMBA PCI to PCI bridge nexus driver)	18	0.3	1
seg_drv (Segment Device Driver v1.1)	25	0.3	3

Figure 4.10: Number of Statically Identifiable Calls to Kernel Functions

Only functions that are successfully parsed are included. There is wide variance between modules in how often its functions are called. The functions in module unix are called on average 13.2 times, compared to the overall average of 5.9 calls. The standard deviation is 0.8 calls. The function mutex_exit is called most often (6200 times). A few small modules have none of their functions (directly) called.

Module	#Functions Parsed	Average # of calls made to a function in this module	Max # of calls made to a function in this module
sad (Streams Administrative driver'sad')	23	1.7	19
namefs (filesystem for namefs)	32	0.3	3
lockstat (Lock Statistics)	27	0.6	5
ptsl (tty pseudo driver slave 'ptsl')	14	1.5	17
rootnex (sun4u root nexus)	19	0.4	1
dada (ATA Bus Utility Routines)	30	2.3	30
dada_ata (ATA Bus Utility Routines)	30	0.4	4
md5 (MD5 Message-Digest Algorithm)	8	1.5	5
sysmsg (System message redirection (fanout) driver)	15	0.3	1
mm (memory driver)	13	0.2	2
wc (Workstation multiplexer Driver 'wc')	18	0.6	7
ebus (ebus nexus driver)	13	0.3	2
ptm (Master streams driver 'ptm')	12	0.1	1
pts (Slave Stream Pseudo Terminal driver 'pts')	12	0.1	1
RT (realtime scheduling class)	24	0.0	0
iwscn (Workstation Redirection driver 'iwscn')	19	1.0	9
fdfs (filesystem for fd)	17	0.1	2
eide (PC87415 Nexus driver v2.0)	19	0.6	7
conskbd (Console kbd Multiplexer driver 'conskbd')	14	0.1	1
todmostek (tod module for Mostek M48T59)	10	0.0	0
log (streams log driver)	9	0.0	0
sy (Indirect driver for tty 'sy')	12	0.0	0
consms (Mouse Driver for Sun 'consms')	14	0.1	1
kstat (kernel statistics driver)	12	0.2	1
pckt (pckt module)	11	0.2	2
ksyms (kernel symbols driver)	11	0.2	2
inst_sync (instance binding syscall)	11	0.9	2
power (power driver v1.4)	11	0.0	0
cn (Console redirection driver)	13	0.0	0
sysacct (acct(2) syscall)	6	1.0	6
clone (Clone Pseudodriver 'clone')	7	0.0	0
intpexec (exec mod for interp)	5	0.2	1
pseudo (nexus driver for 'pseudo')	10	0.0	0
ipc (common ipc code)	5	1.2	5
pipe (pipe(2) syscall)	4	0.0	0
connld (Streams-based pipes)	6	0.0	0
options (options driver)	7	0.0	0
redirmod (redirection module)	6	0.0	0
TS_DPTBL (Time sharing dispatch table)	5	0.0	0
IA (interactive scheduling class)	3	0.0	0
RT_DPTBL (realtime dispatch table)	3	0.0	0
platmod	5	1.0	1
Kernel-wide:	10637	5.9 (std dev:0.8)	6200

Figure 4.10: Number of Statically Identifiable Calls to Kernel Functions

Only functions that are successfully parsed are included. There is wide variance between modules in how often its functions are called. The functions in module unix *are called on average 13.2 times, compared to the overall average of 5.9 calls. The standard deviation is 0.8 calls. The function* mutex_exit *is called most often (6200 times). A few small modules have none of their functions (directly) called.*

4.5.2 Parsing the New Version of a Function At Run-time

Kerninstd analyzes the replacement (new) version of a function at run-time, creating a control flow graph and calculating a live register analysis in the same manner as kernel code that was recognized at kerninstd startup. This uniformity is important because it allows tools built on top of kerninstd to treat the replacement function as first-class. For example, when kperfmon is informed of a replacement function, it updates its code resource display, and allows the user to measure the replacement function like any other.

4.5.3 Undoing Code Replacement

Dynamic code replacement can be undone by restoring the patched call sites (if any), and then un-instrumenting the jump from the entry of the original function to the new version. This ordering ensures atomicity: once code replacement undoing takes effect, no kernel thread may make a call to the replacement function. Consider calls made both before and after the original function has been uninstrumented:

- **Before the Original Function's Jump is Uninstrumented.** There are two sub-cases. In the first, a call is made through a call site that has not yet been restored, so the replacement function is executed. In the second case, a call site has been restored to its original state; the call will reach the original function, which (because it is still instrumented) proceeds to jump to the replacement function. Thus, before the original function's jump is uninstrumented, code replacement is still in full effect.
- **After the Original Function's Jump is Uninstrumented.** After the original function's jump is uninstrumented, all call sites have already been uninstrumented, so there is no longer any way for a thread to call the replacement function, either directly or indirectly.

Although undoing code replacement takes effect atomically, it is still possible for some thread(s) to still be executing within the replacement function. Therefore, as with code splicing, there is a concern of when it is consequently safe to free the memory used for the replacement version of the function and (if used) the springboard. Code replacement presently uses the same ad-hoc solution as code splicing: wait several seconds, on the assumption that such a delay is long enough for the thread(s) to finish executing the code in question; see Section 4.4.

Basic code replacement, when no call sites were patched, is removed in about 65 µs if the original function lies in the nucleus, and about 40 µs otherwise. If a springboard was used to reach the replacement function, then (after the usual delay to ensure that no code is executing within the springboard) it is removed in a further 85 µs if the springboard resided in the nucleus, and about 40 µs otherwise. Each patched call site is restored in a further 30 µs if it resided in the nucleus, and about 16 µs otherwise.

4.6 Future Work

Future work for code splicing includes porting to other architectures, including those with variable-length instructions and those lacking non-delayed splicing branch instructions. Future work concerning the code replacement primitive is also discussed, including how it can be used to achieve the semantics of splicing with less run-time overhead.

4.6.1 Splicing on Architectures Having Variable Length Instructions

Single-instruction splicing on architectures having variable length instructions, such as x86, is challenging because it is not always possible to overwrite just one instruction. Depending on the existing code at the instrumentation point, an unconditional jump instruction (5 bytes on the x86) may overwrite more than one instruction. If the extra (perhaps partially) overwritten instruction is the destination of a branch, a corrupted instruction stream will be executed. Thus, the safety rules for splicing must be augmented: not only must the splice be performed with a single branch or jump instruction to the code patch and be written atomically, but at most one instruction can be replaced. This requirement can always be satisfied by writing a one-byte trap or illegal instruction, which will transfer control to a trap handler. The handler can be instrumented to look up the address of the offending instruction in a hash table, undo the side effects of the trap, and jump to the appropriate code patch.

Although an x86 port of KernInst is still in progress, the existence of trap-based x86 kernel tracing tools [51, 60] indicate that a similar approach is feasible for splicing.

4.6.2 Splicing on Architectures Having Always-Delayed Branches

Single-instruction splicing is difficult on architectures such as MIPS [86] whose branch instructions always have a delay slot. In such an architecture, a delayed branch instruction must be used for splicing, resulting in an unusual execution sequence when jumping to the code patch. The instruction following the instrumentation point falls into the newly written branch instruction’s delay slot, and thus is implicitly executed before the code patch is reached. In particular, the instruction after the instrumentation point is executed before the instruction originally at the instrumentation point, which has been relocated to the code patch. Thus, the executing ordering of these two instructions is effectively reversed.

If the instrumentation point instruction and its successor are mutually independent, then reversing their execution order is harmless. If they are not independent, but the instrumentation point instruction is independent of its successor and if the successor instruction is idempotent, then both the instrumentation point instruction and its successor can be placed in the code patch. The resulting execution sequence is (1) successor, (2) original instrumentation point instruction, (3) successor; step (1) is equivalent to a no-op. Of course, the independence and idempotency constraints will not always be met, making single-instruction splicing on always-delayed-branch architectures difficult. Another possibility is to splice by replacing the instrumentation point instruction with a trap or illegal instruction, as discussed above for x86 splicing. The present implementation of kerninstd does not require such a solution because the SPARC v9 architecture has a non-delayed unconditional branch instruction that is used for splicing.

4.6.3 Using Code Replacement to Facilitate Splicing

The above sections show that it difficult and/or expensive to splice at certain locations on x86 and MIPS architectures. Code replacement can often help; as presently implemented in the dyninstAPI and Paradyn user-level tools, a new copy of the function with extra `nop` instructions inserted can facilitate splicing. Taking this idea further, instrumentation code can be inlined within the replacement function, without the use of `nop` instructions that are later spliced. In other words, code replacement can achieve the effect of splicing without having to splice at the instrumentation point.

Using code replacement to inline instrumentation code can also be seen as an optimization that eliminates the overhead of the various branches and jumps inherent in splicing (in the worst case, from the instrumentation point, to a springboard, to the code patch, and back to just after the instrumentation point). Therefore, it is possible to achieve fine-grained instrumentation with no branch or jump overheads—heretofore an advantage exclusive to static instrumentation systems [53, 54, 76, 78, 88, 95]. The code replacement itself uses a branch (from the entry of the old function to the replacement function), however that overhead can often be eliminated when code replacement also patches statically identifiable call sites to directly call the new function.

4.6.4 Improvements to Code Replacement

Code replacement approximately doubles the code size of those function(s) presently being replaced. It is conceivable to “recycle” the original function as springboard or code patch space while it is being replaced. However, safety requires first determining that no thread is presently executing within the code being recycled.

A replacement function is first-class within kerninstd and its applications. For example, the function has its control flow graph parsed, can be spliced, and can itself be replaced like any other function. However, kernel tools not based on kerninstd will not recognize replacement functions. Using /dev/kerninst to dynamically add an entry to the kernel’s runtime symbol table structure would allow the replacement function to be recognized by the standard /dev/ksyms driver, perhaps enabling the function to be recognized by the system’s kernel debugger, for example. Further research is needed in this area, as there will certainly be additional issues before replacement code can be recognized as first-class on a system-wide basis.

Code replacement for long-running functions is an unresolved issue. In particular, if a thread was already in the middle of the original function, then code replacement of that function for that specific thread does not take effect until it next calls that function. In long-running functions, this can be a concern. In functions that never return, executed by kernel background threads such as the paging daemon, code replacement as presently implemented will never appear to take effect.

4.7 Conclusion and Research Contributions

This chapter presented the design and implementation fast fine-grained code splicing for a completely unmodified, modern commodity operating system (Solaris 7 running on an UltraSPARC), entirely at run-time. In splicing code patches into the kernel, it is possible to safely instrument a kernel without pausing or synchronization. Springboards provide a general technique for obtaining arbitrary reach regardless of the maximum displacement available in a single branch instruction. In relocating the overwritten instruction to the code patch, this chapter makes two contributions. First, it is always possible to maintain semantics when relocating the overwritten instruction, possibly through a semantically-equivalent sequence of several instructions. Second, in relocating tail calls and conditional branches, code splicing has an additional (and unforeseen) benefit: essentially creating important new instrumentation points where none existed before. The ability to instrument at these new points (before a tail call sequence exits, and in the if-taken and if-not-taken case of a conditional branch) shows that even having an instrumentation point before every machine code instruction is not enough to satisfy all instrumentation requests. And finally, with a few additions, the splicing implementation can easily achieve code replacement, a complementary dynamic instrumentation primitive.

Chapter 5

Kernel Performance Measurement Through Dynamic Instrumentation

This chapter presents the design and implementation of a kernel profiling tool built using dynamic instrumentation. The tool, *kperfmon*, serves as a proof of concept of the utility of dynamic instrumentation. More important, however, kperfmon is a powerful kernel profiler in its own right, providing a flexible and extensible set of performance metrics that can be applied to almost any kernel function or basic block. Kperfmon can create a new wall time metric out of any monotonically increasing software-readable counter. Furthermore, kperfmon can virtualize (i.e., exclude the time spent switched out) any wall time metric by instrumenting the kernel's context switch handlers. This chapter discusses the design and implementation of performance metrics in kperfmon. Chapter 6 presents a case study of using kperfmon to understand and to help optimize the performance of a Web proxy server.

A performance measurement in kperfmon is specified by combining a *metric* with a code *resource*. A metric is any time-varying measurement, such as elapsed cycles, number of data cache misses, or number of database transactions. A code resource is the object being measured by a metric, currently either a function or a basic block. A metric and resource pairing causes kperfmon to generate instrumentation code snippet(s) and the kernel location(s) at which to insert them. Kperfmon then instructs kerninstd to splice the snippets into the kernel. The instrumentation code will update a counter or event accumulator structure. Kerninstd periodically samples this value, which is forwarded to a front-end GUI for consumption.

Because finding bottlenecks is an interactive activity involving successive refinement of the functions or basic blocks to be measured, it is essential for a tool to allow the user to decide what measurements are made at run-time. In kperfmon, instrumentation occurs on request, and can be removed at any time, making kernel performance profiling completely dynamic.

The major features of kperfmon are:

- **Commodity.** Kperfmon measures an unmodified Solaris7/UltraSPARC kernel, requiring instrumentation code that is both thread and multiprocessor safe. Presently, kperfmon has only been implemented on a uniprocessor, though multiprocessor design issues are discussed throughout this chapter.
- **Dynamic.** The overhead of instrumentation is incurred only for what is measured, when it is measured.
- **Fine-grained.** Code resources can be almost any kernel function or basic block.
- **Accurate.** Kperfmon instruments the kernel to directly measure performance, rather than assigning time through periodic sampling of the program counter.
- **Inclusive metrics.** Measurements for a code resource accumulate events whenever code resource is on the call stack, including the time spent in its callees. Inclusive metrics have been shown to aid in the automated search for bottlenecks [14], and are difficult to measure with a sampling-based profiler.

- **Extensible set of metrics.** Any monotonically increasing software-readable counter can serve as a new *event counter* metric (Section 5.1). Any event counter metric can, in turn, be used to construct a new *interval counter* metric (Section 5.2), which measures the number of such events that occur during the execution of a user-specified code resource.
- **Wall and Virtual Time Metrics.** Kperfmon’s basic instrumentation code gathers wall time measurements; each thread’s interval event count includes any events that occur while context switched out between the entry and exit instrumentation points. Wall time metrics are used when measuring the latency of certain code that may block, such as I/O routines and obtaining locks. Virtual time measurements can optionally be achieved by instrumenting the kernel’s context switch handlers to exclude from the accumulated total those events that occur while switched out (Section 5.4). Virtual time metrics are useful for measuring processor events, such as CPU time, branch mispredicts, and cache stall time.

The remainder of this chapter discusses the design and implementation of kperfmon’s various classes of performance metrics.

5.1 Event Counter Metrics

An event counter metric has a counter that is incremented on each occurrence of a particular event. An event conceptually corresponds to the execution of some point in the kernel’s code, perhaps predicated by a logical expression. Some event counters are already kept in hardware, such as UltraSPARC registers for elapsed cycles, number of cycles idled due to I-cache misses, and number of branch mispredictions [96, 97]. Others are calculated in software, such as Solaris’ kstats [18, 91]. Software event counters can also be dynamically created by kperfmon via instrumentation, such as counting the number of entries to a desired function. An event counter serves as a useful performance result in its own right, but is also used by kperfmon as the foundation for interval counter metrics, discussed in Section 5.2.

Event counters created by kperfmon are updated by instrumentation code that increments the counter. The only non-trivial aspects are making the instrumentation thread-safe and multiprocessor-safe. Thread safety requires correctly incrementing the counter when multiple threads may be executing the same instrumentation simultaneously. (Although only one thread at a time can run on a uniprocessor, preemption can occur at any time, which leads to the same requirements.) Kperfmon achieves thread safety, without locks, by updating the counter using the SPARC V9 compare-and-swap instruction [103]. Statistics for kperfmon's event counting instrumentation are shown in Figure 5.1.

Operation	Code size (bytes)	Scratch registers needed	Cost
Event counter increment	40	3	0.32 µs

Figure 5.1: Cost of Incrementing an Event Counter

Cost was measured by using kperfmon to instrument its own instrumentation code. All costs in this chapter are on a 440 Mhz UltraSparc-IIi running Solaris 7, and exclude the overhead for jumping to the code patch and for executing the relocated instruction. Because event counter instrumentation usually occurs at function entry, where few registers are free (Section 3.5), the code size and cost columns describe the worst case, where instrumentation code is bracketed with SPARC save and restore instructions.

Two modifications are required for multiprocessors. First, for correctness on processors with relaxed memory consistency, the code sequence must be annotated with an appropriate memory barrier instruction. Second, for good performance, per-processor event counters should be maintained, to avoid significant cache coherency delays among processors competing for write access.

5.2 Interval Counter Metrics: Overview

An interval counter metric accumulates the number of events (as specified by an underlying event counter metric) that occur when some thread is running within a specified code resource. The general framework for an event accumulating metric was shown in Figure 1.1 (page 11); currval in the figure is implemented with an event counter. At the entry point to the code resource (a function or basic block), instrumentation code to start accumulation is inserted. At the exit point(s) of the resource, instrumentation is inserted to stop accumulation, and add the number of events that occurred since the start point to the accumulator's total.

The framework of Figure 1.1 omits several implementation details:

- **Overlapping Intervals.** Conceptually, each thread accumulates its own interval count between the start and stop interval count instrumentation. When measuring wall time events, these intervals can overlap. For example, thread A can start and stop an interval at (wall) times 100 and 500, respectively. In between those times, thread A could context switch out, and thread B can start and stop the same accumulator at times 200 and 400, respectively. So for 200 time units, there are two threads accumulating their own intervals, which needs to be reflected in the total. Figure 5.2 on page 117 further illustrates wall time accumulation in the presence of overlapping intervals.
- **Virtualization.** The framework of Figure 1.1 places instrumentation that does not detect context switching. It must be augmented to measure virtual time events.
- **Entry and Exit (Start and Stop) Instrumentation Should be Inserted Atomically.** This is necessary to avoid a race condition that can fail to properly stop accumulation of an interval. In particular, if entry point instrumentation is inserted, and some thread executes it and continues execution past the exit point (where instrumentation has not yet been inserted), then an interval's end time

will be incorrect. Kperfmon solves this problem by inserting all stop-accumulation instrumentation first, and by having stop instrumentation perform a no-op for an unstarted accumulator.

- **The “poll-problem” for Wall Time Accumulators.** As mentioned above, when measuring a wall time interval counter metric, each thread conceptually accumulates distinct intervals, which sum to the metric’s grand total. Incorrect measurements can occur when a thread executes exit instrumentation without the corresponding entry instrumentation, which can occur in blocking routines such as poll, which waits on a file descriptor. For example, assume instrumentation is added to poll’s entry and exit points while a thread T_1 is presently blocked in poll. Eventually, T_1 un-blocks and executes the exit instrumentation of poll. A correct implementation will perform a no-op, detecting that T_1 did not execute the corresponding entry instrumentation. A naive implementation may mistakenly stop another thread’s accumulation. Typically, an implementation that uses a shared accumulator structure, instead of a per-thread one, suffers from this problem. However, a solution using per-thread accumulators leads to extra complexity, storage, and run-time cost [104]. As we will see in Section 5.5, an implementation of wall time accumulators that avoids the “poll problem” with reasonable complexity, storage, and cost is challenging but feasible.
- **Dealing With Recursion.** Recursion should be handled by only starting an interval the first time a thread reaches the code’s entry, and only stopping an interval the last time that thread reaches the code’s exit.
- **Multiprocessors.** Fast execution on multiprocessors requires per-processor copies of the event accumulator structure, to avoid cache coherency overhead during writing of a shared structure. Correctness requires handling migration.

5.3 Wall Time Interval Counter Metrics

This section presents the design and implementation of inclusive wall time interval counter metrics. The design, called *cumulative thread events*, is simple and efficient. Kperfmon currently uses this design in its wall time metrics. The design has three limitations: it does not work on multiprocessors, it can give inaccurate results in the presence of recursion, and it suffers from the “poll problem” discussed above. These limitations are addressed by a re-design that adds complexity, as discussed in Section 5.5. When the above limitations are tolerable, the framework described in this section is preferable.

Cumulative thread events instrumentation calculates the area under a curve whose *x*-axis is the underlying event counter (which is typically elapsed cycles) and whose *y*-axis is the number of threads presently accumulating these events. An example is shown in Figure 5.2. The units of the total value are thread-events (e.g., thread-cycles when the underlying curval is elapsed cycles).

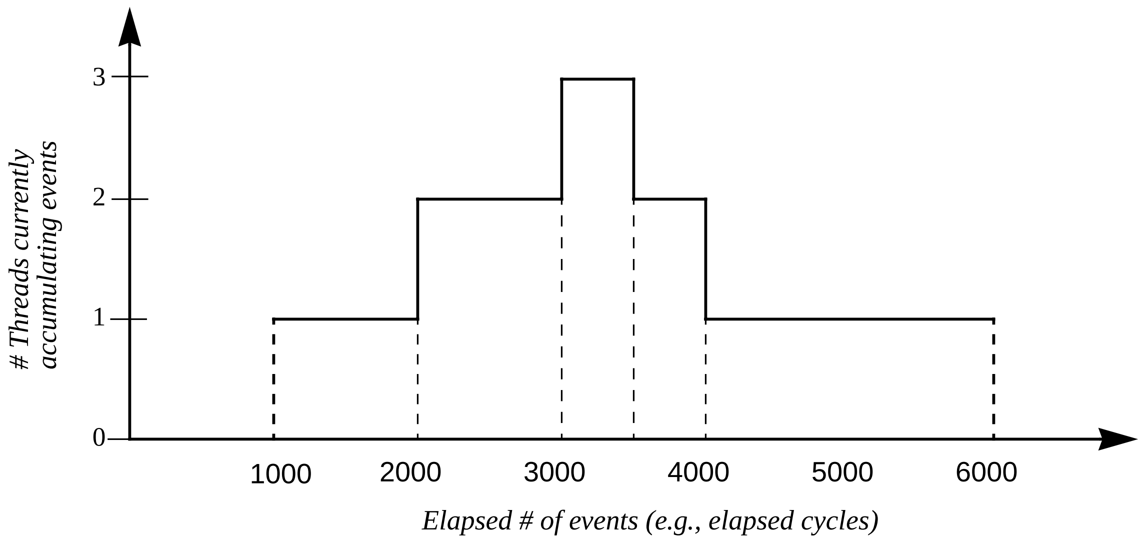


Figure 5.2: Event Interval Accumulation With Multiple Threads
Instrumentation code calculates the area under the curve by adding rectangles when a thread starts or stops accumulation (dashed lines). In this example, start-accumulation occurs at 1000, 2000, and 3000, and stop-accumulation occurs at 3500, 4000, and 6000. The total value for this event accumulator is 7500 thread-events.

The code for entry and exit instrumentation is almost identical, either adding (for entry instrumentation) or subtracting (for exit instrumentation) one from the number of accumulating threads. Before this field is updated, a rectangle is added to the total. The rectangle's width is the elapsed number of events that occurred since the last update of the total field, and its height is the number of threads that were accumulating events during that time.

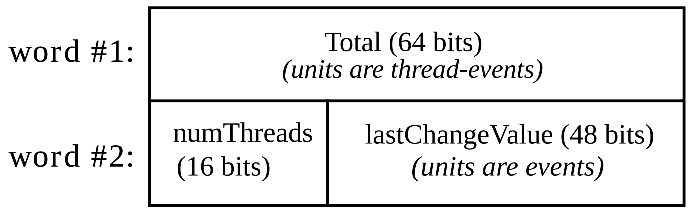


Figure 5.3: Cumulative Thread Events: Accumulator Structure
The accumulator occupies two 64 bit words. The first word contains the total of all accumulated intervals. The second word contains two sub-fields: the number of threads that are presently accumulating an event interval (16 bits), and the value of the underlying event counter at the most recent start or stop operation (48 bits). Putting both the numThreads and lastChangeValue sub-fields into a single 64 bit word allows them to be updated with a single thread-safe compare-and-swap. lastChangeValue stores the least significant 48 bits of the underlying event counter, so the counter's granularity is maintained. 48 bits is usually enough; on a 1 GHz machine, a 48 bit cycle counter can accumulate 195 days worth of cycles before rolling over.

The accumulator structure for cumulative thread events is shown in Figure 5.3, and its instrumentation pseudo-code is shown in Figure 5.4. The first 64-bit word holds the accumulated total of all intervals (the area under the curve in Figure 5.2). The second word holds two fields: the number of kernel threads that are presently accumulating events (16 bits), and the value of the underlying event counter at the time of last modification (48 bits). The code first updates the second word. The new value contains the adjusted number of threads and (the least significant 48 bits of) the current value of the underlying event counter. After this completes, the old number of threads is multiplied by the delta in the underlying event counter to obtain a delta to be added to the total field. The lastChangeValue field only stores the least significant 48 bits of the event counter, so 48-bit unsigned arithmetic is used to calculate the change in the event counter. If the new value of the event counter is less than that stored in lastChangeValue, subtraction assumes that rollover has occurred exactly once. To obtain incorrect results, a complete loop of 2^{48} events must occur between starting and stopping accumulation. Such a loop is unlikely; 48 bits is enough to accumulate over 195 days worth of machine cycles on a 1 GHz machine, for example. Both the first and second 64-bit words are updated using separate compare-and-swap operations, for thread safety without the need for locks.

```
PhaseA: // calculate Δtotal; update accum.word2 sub-fields with new values
  oldWord2=timer.word2 // both numThreads and lastChangeValue sub-fields
  currval=current // current is the underlying event counter's present value
  Δvalue=currval-oldval.lastChangeValue
  Δtotal=oldval.numThreads × Δvalue
                  ⎧ oldWord2.numThreads+1           (for a start operation)
  newNumThreads = ⎨
                  ⎩ max(oldWord2.numThreads-1, 0)   (for a stop operation)
  // Change timer.word2 to {newnumThreads, currval} if it still equals oldWord2:
  cas64 (&timer.word2,oldWord2,{newNumThreads,currval})
  if failure goto phaseA  // fails if another thread updated timer.word2
PhaseB: // using Δtotal, update accum.total
  oldTotal=accum.total
  newTotal=oldtotal+Δtotal
  cas64 &accum.total,oldTotal,newTotal
  if failure then goto phaseB // do not repeat the completed phase A
```

Figure 5.4: Cumulative thread events: pseudo-code

Both start and stop accumulation code are nearly identical, differing only on how they calculate newNumThreads. Because compare-and-swap is limited to 64 bits on the UltraSPARC, updates are done in two phases. In phase A, the second word is updated and the delta used in phase B is calculated. In phase B, first word (the total field) is updated using that delta. The compare-and-swap loops are only repeated when there is contention among multiple threads for the same structure. The primitive is non-blocking: an infinite loop, where the compare-and-swap is retried forever with no thread making forward progress does not occur.

Performance numbers for the start and stop primitives are shown in Figure 5.5. As expected given their similarities, the costs are nearly identical.

Operation	Code Size (bytes)	Scratch registers needed	Cost
Start accumulation	128	4	0.29 µs
Stop accumulation	124	4	0.31 µs

Figure 5.5: Cost of Cumulative Thread Events Instrumentation Primitives
These numbers were measured by using kperfmon to instrument its own instrumentation code. In keeping with the measured averages in Section 3.5.5 (page 72), it is assumed that sufficient scratch registers are available at the code resource's exit point but not at the code resource's entry point. Thus, the size and cost for starting accumulation assumes that kerninstd brackets the code with save and restore instructions to free up enough scratch registers.

The cumulative thread events framework has several desirable traits. It is a simple, quick, and thread-safe framework for wall time interval counter metrics. It is thread-safe, correctly handling overlapping accumulation intervals. And finally, it measures inclusive events, known to be useful in automating the search for bottlenecks [14].

However, the design has several limitations. First, the "poll problem" is not solved. The stop primitive checks for underflow, performing a no-op when the old number of threads field was zero. Unfortunately, if this field was greater than zero, due to overlapping interval accumulation by a *different* thread, the no-op will not be performed, and the "poll problem" occurs. Per-thread copies of the accumulator structure would solve this problem, though at the cost of significantly higher storage, complexity, and execution costs [104].

A second limitation occurs in the presence of recursion; a recursive call to instrumented code will incorrectly increment numThreads, because the instrumentation will act as if a different thread is beginning accumulation of an overlapping interval.

A third limitation is that it does not work on multiprocessors. Assuming each processor has its own accumulator structure, the code can be fooled by migration. If a thread starts an interval on one processor and stops on another processor, the stop operation of Figure 5.4 will be applied to the wrong accumulator.

A re-design of cumulative thread events that solves these problems is presented in Section 5.5. However, the re-design is significantly more complex, and has not yet been implemented.

5.4 Virtual Time Interval Counter Metrics

Instrumentation that accumulates virtual time events is more complex than instrumentation code that accumulates wall time events. Simply starting and stopping accumulation at the code resource’s entry and exit points will accumulate wall time events, not virtual time events. With additional kernel instrumentation, interval counters can be adapted to virtual time. This section describes the design and implementation of a virtualized interval counting metric framework called *counted virtual time events*. Like cumulative thread events, the counted virtual time events framework can create a metric out of any underlying event counter. Counted virtual event instrumentation has two parts: start and stop primitives inserted at the code resource’s entry and exit points, and instrumentation placed in the kernel’s context switch routines to stop accumulation when a thread is switched out, and re-start when that thread is resumed.

5.4.1 Start and Stop Primitives

Counted virtual time's accumulator structure is shown in Figure 5.6, and its basic start and stop primitives are shown in Figure 5.7 and Figure 5.8, respectively.

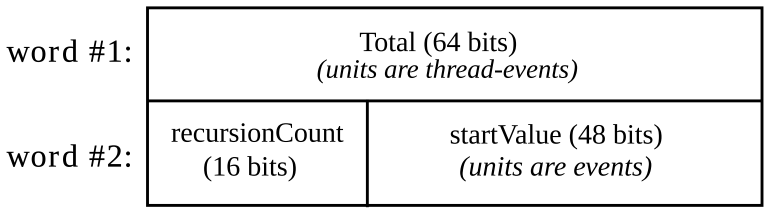


Figure 5.6: Counted Virtual Time Events: Accumulator Structure
The accumulator occupies two 64 bit words. The first word contains the total of all accumulated intervals. The second word contains two sub-fields: the recursion depth of the code resource being measured (16 bits), and the least significant 48 bits of the underlying event counter when accumulation was last started. If the recursion count is 0, then accumulation is presently not occurring, and the startValue field is undefined. Placing both the recursionCount and startValue sub-fields into a single 64 bit word allows them to be updated with a single thread-safe compare-and-swap.

```
Retry:
  oldWord2=accum.word2 // both the recursionCount and startValue sub-fields
  oldRecursionCount=oldWord2.recursionCount
  oldStartValue=oldWord2.startValue
  currval=current // current is the underlying event counter's present value

  newWord2 = { {1,currval}                             (if oldRecursionCount==0)
             { {oldRecursionCount+1,oldStartValue}     (otherwise)

  // Change accum.word2 to newWord2 if it still equals oldWord2:
  cas64 &accum.word2,oldWord2,newWord2
  if failure goto Retry // fails if another thread updated accum.word2
```

Figure 5.7: Counted Virtual Time Events: Start Accumulation
The design closely follows the framework of Figure 1.1 (page 11), with the addition of the recursion count field. The total field is not read or written by the start operation.

Because at most one thread can actively accumulate virtual events (assuming a uniprocessor), the numThreads field of the cumulative thread events accumulator is not needed. In its place is a recursionCount field, which is zero if the currently running thread is not accumulating events, and greater than zero otherwise. If non-zero, the

```
Retry1: // calculate Δtotal; update accum.word2 sub-fields with new values
 oldWord2=accum.word2 // both the recursionCount and startValue sub-fields
 oldRecursionCount=oldWord2.recursionCount
 oldStartValue=oldWord2.startValue
 if oldRecursionCount==0 goto done
 newWord2={oldRecursionCount-1,oldStartValue}
 // Change accum.word2 to newWord2  if it still equals oldWord2
 cas64 &accum.word2,oldWord2,newWord2
 if failure goto Retry1 // fails if another thread updated accum.word2
 if oldRecursionCount>1 goto done
 currval=current // current is the underlying event counter's present value
 Δtotal=currval-oldStartValue
Retry2: // update total if oldRecursionCount is now 0
 oldTotal=accum.total
 newTotal=oldTotal+Δtotal
 // Change accum.total to newTotal  if it still equals oldTotal:
 cas64 &accum.total,oldTotal,newTotal
 if failure goto Retry2 // fails if another thread updated accum.total
done:
```

Figure 5.8: Counted Virtual Time Events: Stop Accumulation

Stopping virtual event accumulation decrements the recursionCount field and (if now 0) adds this interval to the accumulated total field. Because only a single 64-bit word can be updated at a time using compare-and-swap, a two-phase protocol is used. In the first phase, the second word is updated, and a delta for the total field is calculated. The second phase uses the delta to update the total field.

recursion count field gives the number of times the currently running thread has attempted to begin accumulation. The start primitive always increments this field, but only begins accumulation when it changes from 0 to 1. The stop primitive always decrements this field (with the usual proviso of keeping it non-negative), but only stops accumulation when it is changed from 1 to 0. The recursion count field allows the primitives to work correctly under recursion, a feature lacking in cumulative thread events. Figure 5.9 summarizes the cost of the interval counting primitives to start and stop a virtual time interval counter.

Primitive	Code Size (bytes)	Scratch Registers Needed	Cost
Virtual time event interval metric: start	68	4	0.28 µs
Virtual time event interval metric: stop	116	4	0.34 µs

Figure 5.9: Summary of Virtualized Interval Counting Instrumentation Primitives
The numbers include only instrumentation placed in the code resource; instrumentation added to the kernel's context switch code is discussed below. Start instrumentation is assumed to be bracketed with save and restore instructions due to insufficient scratch registers; this cost has been included.

5.4.2 Virtualization: General Algorithm and Additional Structures

KernInst virtualizes event accumulators by dynamically instrumenting the kernel's thread context switch handlers. The implementation currently assumes a uniprocessor. Changes to allow virtualization on multiprocessors are discussed in Section 5.4.4.

Virtualization splices the following code into the kernel's context switch routines:

- **On switch-out:** stop every currently active virtual accumulator that was started by the kernel thread that is presently being switched out.
- **On switch-in:** restart all virtual accumulator(s) that were stopped by the most recent switch-out of the thread that is presently being switched in.

The following invariant aids the implementation of switch-out code:

> Any presently active virtualized accumulator was started exclusively by the currently running thread T_1.

For a proof, assume that any other thread $T_2 \neq T_1$ started this accumulator. T_2 cannot presently be running, because (assuming a uniprocessor) only one thread runs at a time. Therefore, T_2 is presently context switched out. When it was switched out, virtualization instrumentation stopped T_2's accumulation, so we have a contradiction with the assumption that T_2 was presently accumulating events. Because no thread $T_2 \neq T_1$ started accumulation, T_1 must have started accumulation (since some thread started accumulation). Thus, knowing which accumulators to virtualize on context switch-out is trivial—virtualization should stop every presently active accumulator.

The context switch-in instrumentation requires information about which accumulators, if any, were stopped by the most recent switch-out of this thread. This information has been implemented in a hash table, indexed by thread ID, whose entries contain information about what was stopped at the most recent switch-out of this thread. Specifically, this per-thread information contains pointers to all of the virtual accumulator(s) that were stopped, along with a copy of the recursion count field(s) at that time.

Any number of threads may presently be switched out after having started, and before having stopped, the same accumulator. Therefore, multiple hash table per-thread information can contain pointers to the same accumulator structure. Thus, the implementation is a combination of the single accumulator approach (used by cumulative thread events) and per-thread accumulator copies. In particular, there is one accumulator for the actively running thread, plus per-switched-out-thread information about the accumulators that are presently turned off due to virtualization. This hybrid organization compares favorably to one that always allocates a per-thread copy of every accumulator, which has extra complexity as well as additional space and time overhead [104].

Pseudo-code for the context switch-out and switch-in code is shown in Figure 5.10 and Figure 5.11, respectively. Context switch-out instrumentation first allocates a vector from a free list. This vector will gather the set of accumulators that were stopped by virtualization. It then loops through all accumulators (the address of each is kept in a global vector), invoking a metric-specific routine that stops the accumulator if it was started. (Code to stop an active accumulator is specific to the metric's underlying event counter.) When done, the vector holding the set of accumulators that were turned off is added to a hash table indexed by thread ID.

```
assert(%pil >= 10); // This code executes at a high processor interrupt level
vecOfStoppedAccumulators=freelist.get(); // freelist must be pre-allocated
nextStoppedAccumPtr=vecOfStoppedAccumulators; // a pointer
for each virtual accumulators VA do {
  if (VA.recursionCount > 0) then {
     VA.total+=currval-VA.startValue; // currval: underlying event ctr value
     nextStoppedAccumPtr.theAccumulatorPtr=&VA;
     nextStoppedAccumPtr.savedRecursionCount=VA.recursionCount;
     nextStoppedAccumPtr++;
  }
}
nextStoppedAccumulator.theAccumulatorPtr=NULL; // place a sentinel
if any accumulators were stopped then
  hash.set(%g7,vecOfStoppedAccumulators); // %g7: current thread ID
else
  freelist.free(vecOfStoppedAccumulators); // ended up not using it
```

Figure 5.10: Context Switch-Out Instrumentation Pseudo-Code (uniprocessor)
Note that there is no need for synchronization in this instrumentation because it runs at the same interrupt level as the context switch handlers and cannot be preempted except by high-level interrupts (such as an ECC error) or traps (such as register window overflow). The high interrupt level limits the functions that can be called. For example, the vector of stopped accumulators must come from a pre-allocated heap, because calling kmem_alloc *is dangerous from within an interrupt routine (it may block).*

Context switch-in instrumentation code is comparatively simple. It uses the ID of the newly running thread as an index into the global hash table, obtaining a vector of pointers to accumulators that need to be restarted, as well as their saved recursionCount fields at the time of switch-out. Each such accumulator is restarted by invoking a metric-specific routine that depends on the underlying event counter. Finally, the vector of accumulator addresses is returned to the free pool, and the hash table entry for this thread is removed. The size and execution cost of context switch-out and switch-in code are shown in Figure 5.12.

```
  assert(%pil >= 10); // This code executes at a high processor interrupt level
  vecOfStoppedAccumulators=hash.getAndRemove(%g7); // %g7: curr thread ID
  if not found then goto done // nothing needed for this thread
  nextStoppedAccumPtr=vecOfStoppedAccumulators; // a pointer
  while (nextStoppedAccumPtr→theAccumulatorPtr ≠ NULL) {
    // VAptr: pointer to virtual accumulator structure to restart
    VAptr=nextStoppedAccumPtr→theAccumulatorPtr; // accumulator to restart
    VAptr→recursionCount=nextStoppedAccumPtr→savedRecursionCount;
    VAptr→startValue=current; // current value of underlying event counter
    nextStoppedAccumPtr++;
  }
  freelist.free(vecOfStoppedAccumulators);
done:
```

Figure 5.11: Context Switch-In Instrumentation Pseudo-Code (uniprocessor)
As with context switch-out instrumentation, no synchronization is needed because this code is always invoked at a high interrupt priority level, which prevents scheduling.

Operation	Code Size (bytes)	Cost
Context-switch out virtualization code	816	0.65 µs
Context-switch in virtualization code	412	0.58 µs

Figure 5.12: Cost of Context Switch Instrumentation
These numbers were measured by using KernInst to instrument its own instrumentation code. Times shown are on a 440 Mhz UltraSparc-IIi system running Solaris 7.

5.4.3 The Context Switch Instrumentation Points

Virtualization requires identifying all of the kernel’s context switch-out and switch-in sites. The Solaris kernel only schedules kernel threads; all user-level processes, kernel background tasks, and even interrupts are bound to a kernel thread. Solaris has several thread context switch sites, depending on the machine state (particularly the interrupt level) at the time of the context switch. The myriad of context switch points demonstrates that kerninstd is able to instrument low-level kernel routines that run at a high priority level. The variants are:

- Non-interrupt threads, e.g. threads bound to a user process. This is the common case. An optimized variant is used to switch from a zombie thread.
- Interrupt threads whose priority is below that at which the context switch routines themselves execute. Such "low-level" interrupts are bound to a thread, and are allowed to call code that may block.
- Interrupt threads whose priority equals that at which the context switch code executes. An example is the clock interrupt.
- Interrupt threads whose priority is greater than or equal to that at which the context switch routines execute. Such code is not permitted to block. An example is ECC error detection.

As an optimization, the kernel predicts that interrupt handlers complete without blocking. The interrupt is initially bound to a skeletal thread, and the execution context is piggybacked onto the interrupted thread (a process known as pinning). Should the interrupt block or be preempted, the skeletal thread is fully initialized, and the interrupted thread is un-pinned. Interrupts running at a priority level greater than the scheduler must not block, which could cause the context switch code to be preempted and then rescheduled. Similarly, such interrupts can preempt kperfmon's virtualization code, so it is unsafe to perform virtualized measurements of any such high-level interrupt routine, or any routine that it may invoke. Fortunately, because high-priority interrupts may not block, the set of such routines is small.

Figure 5.13 describes most of the context switch points that kperfmon instruments. Virtualization occurs for every context switch except for high-priority interrupts, such as like ECC errors. Any instrumented kernel code that is interrupted

Kernel Location	Switch-Out or Switch-In Point?	Description
swtch and swtch_to, at their respective calls to resume	switch-out	Switches out a non-interrupt thread (the common case)
swtch_from_zombie, at the call to resume_from_zombie	switch-out	Switching from a just-killed thread; doesn't need to save context.
swtch, at the call to resume_from_intr	switch-out	Switches out an interrupt thread.
exit of resume_from_idle (just before the routine lowers the interrupt priority level, however)	switch-in	Called from resume and resume_from_zombie; the common case of switching in a non-interrupt thread.
entry to intr_thread	switch-out	Entry point for low-priority interrupt. Pins the interrupted thread.
Within intr_thread, before it dispatches the interrupt	switch-in	Skeletal interrupt has been created and is ready to run.
Within intr_thread, before it changes %g7 back to the interrupted thread	switch-out	switch-out of a low-priority interrupt thread, when it ran to completion without blocking.
Within intr_thread, just after it changes %g7 back to the interrupted thread	switch-in	switch-in of the thread that had been interrupted, when the interrupt ran to completion without blocking.
exit of resume_from_intr	switch-in	Un-pins the thread that had been interrupted, and resumes it. Used when the interrupt code blocks.

Figure 5.13: Virtualization Instrumentation Points (Abbreviated)

Context switch-out code is spliced at a point where register %g7 contains the ID of the thread that is about to be switched out. Context switch-in splicing occurs at a point where %g7 contains the ID of the thread that is about to be switched in. In addition, splicing occurs at a points where the interrupt priority level is high enough to prevent the scheduler from being preempted (except by high-level interrupts or traps). intr_thread *handles low-priority interrupts;* clk_thread *(excluded from this table for brevity) handles interrupts at the same priority level as the scheduler itself. Higher-level interrupts (handled by* current_thread*) are not instrumented by kperfmon.*

by a high-priority interrupt will not be able to exclude any events that occur in the interrupt. This omission is not a major problem, because high-priority interrupts seldom run for long (they cannot perform any synchronization except for spin locks). Trap handlers, which run at a higher priority than any interrupt, are also not virtualized. However, because such exceptions are usually code that is directly executed by a thread (such as a save instruction leading to a register window overflow trap), it is desirable to include this time in virtualized totals.

Although there are quite a few context switch points, kperfmon is programmed to identify each function that contains a context switch, and where within that function the context switch code is located.

The virtualization code, while simple and inexpensive, can be sub-optimal in a few cases. In particular, a thread being switched out may stop an accumulator, only to have it restarted by the subsequent thread that is switched in. The check for such an optimization requires about as much work as doing it naively, so it was not implemented.

5.4.4 Multiprocessor Issues for Virtualization

A multiprocessor-safe alternative for counted virtual time events has been designed. This design replaces the cumulative thread events design (Section 5.3), which is unsuitable for multiprocessors.

A central assumption of the previous design is that only a single thread can accumulate events at one time. This assumption is not valid on a multiprocessor. The assumption can be restored by using per-CPU accumulators, where a single logical accumulator is represented as *numCPUs* physical accumulators. Start and stop primitives use the current CPU ID as a level of indirection when accessing an accumulator. This change restores the invariant that a presently active accumulator must have been started by the currently running thread. Per-CPU accumulators also ensure that different processors will not actively compete for write access to the same accumulator, causing undue cache coherence overhead.

Per-CPU accumulator totals can be summed after their values have been read, and reported to kperfmon as a single value. Because the total field is the sum of events occurring during possibly overlapping time intervals, a multiprocessor version of virtual time has units of thread-events, not events. Alternatively, per-CPU accumulator totals can be reported individually, allowing users to gauge processor-specific bottlenecks.

With per-CPU versions of a single logical accumulator, the virtualization framework is still a hybrid approach: one accumulator per CPU to represent the actively running thread(s), plus hash table information for the accumulators that were stopped by virtualization code, for each presently switched out thread.

The hash table requires reconfiguration to avoid multiprocessor cache coherence overhead in the presence of competing write access among the various processors. A straightforward solution is to pad all hash table structures to data cache block alignment. On the UltraSPARC-I and II, the L1 data cache has 16 byte sub-blocked 32-byte blocks, and the L2 data cache has 64-byte blocks [96, 97].

A virtualized accumulator that was started on a particular CPU is always stopped on the same CPU. To see why, consider a thread that starts an accumulator, migrates, and then stops that accumulator. On Solaris, migration can only occur for a thread that was already context switched out. At the time of migration, previously-executed context switch-out virtualization code has ensured that the accumulator was stopped while it was still running on the original CPU. Similarly, the context switch-in virtualization code re-starts the accumulator while the thread is running on the post-migration CPU. Stopping a virtualized accumulator on the same CPU on which it was started is important because on-chip registers serving as an event counter (such as elapsed cycles or cache misses) may be out of sync across CPUs. A naive implementation could calculate an interval count by reading event counters from different CPUs, possibly leading to a spurious rollover.

Another invariant of the virtualization framework is that migration of a started virtual accumulator cannot occur. Again, context switch-out instrumentation guarantees this invariant; before any thread migrates, it is first context-switched out, where instrumentation will stop all active accumulators. This invariant ensures that a migrating thread cannot leave one CPU's version of an accumulator started indefinitely. This problem is not easily solved within the cumulative thread events framework of Section 5.3.

Despite the above invariants, it is necessary to prevent migration in the middle of a start or stop primitive. If migration could happen in the middle of a primitive, it would be possible to modify the wrong per-CPU accumulator. At the beginning of a primitive, instrumentation code reads the current CPU, using it to calculate the address of the CPU-specific accumulator structure used in the remainder of the primitive. If migration happens immediately after this calculation, but before the compare-and-swap has finalized changes to the accumulator, then the thread—ignorant that it was just migrated—will update the wrong CPU's accumulator.

Migration can be prevented by locking down a thread for the duration of the primitive, by calling affinity_set and affinity_clear, or by calling splhigh to temporarily set the CPU's interrupt level high enough to prevent scheduling. A more elegant solution makes the primitives sensitive to migration by causing their compare-and-swap to fail. The failure will lead to the entire primitive being retried, this time running on the new CPU for the full duration of the primitive.

5.5 Wall Time Event Accumulating Revisited for Multiprocessors

A multiprocessor version of wall time accumulators that does not share the limitations of cumulative thread events (Section 5.3) has been designed. The cumulative thread events design is fooled by recursion into accumulating too high a result, it suffers from the "poll problem" where one thread can mistakenly stop accumulation begun by another thread, and it cannot handle migration. The solution to these problems for wall time that is discussed in this section builds on the virtualized timer framework of the previous section. Such a design may appear odd, because from a high level, the basic virtualization primitives already calculate wall time, then go to much effort to exclude events occurring while context switched out. However, we have seen that a multiprocessor version of virtualized time interval counters is feasible, so it is a good place to begin. Its hybrid of per-CPU accumulators to represent actively running threads and per-thread accumulators to represent switched-out threads makes wall time event accumulation for multiprocessors feasible.

The key equality that guides the design presented in this section is:

$$\text{wall time} = \text{virtual time} + \text{blocking time}$$

In other words, wall time is the time spent actively running plus the time spent context switched out. Similarly,

$$\text{wall time events} = \text{virtual time events} + \text{blocking time events}$$

Virtual time events can be measured using the design of Section 5.4.4, leaving blocking time events to solve. Blocking time events for each accumulator can be obtained by measuring the events occurring while that accumulator is stopped due to virtualization. Specifically, on context switch-out of an active accumulator, virtualization stops the accumulator. At that time, a different interval measurement is begun, to count the events that occur while that accumulator is stopped due to virtualization. On context switch-in, virtualization re-starts an accumulator if it had been turned off due to context switch-out of this thread; blocking time uses this cue to stop its associated interval measurement. In this way, blocking time events can be measured with one counter per virtualized accumulator. This information can be piggybacked as a third field of a hash table entry, as shown in Figure 5.14.

The solution, though it has some drawbacks (complexity, extra storage in the hash table, and slightly more expensive context switch-out and switch-in code), is desirable for several reasons. First, the runtime instrumentation cost of the basic start and stop primitives (i.e., excluding context switch instrumentation) is only slightly higher than the uniprocessor versions (see Figure 5.9), which in turn was less expensive than the primitives for cumulative thread events (Figure 5.5). The only additional perturbation compared to virtualized accumulators is a bit of extra code in the context switch-out and switch-in instrumentation. An additional benefit of this

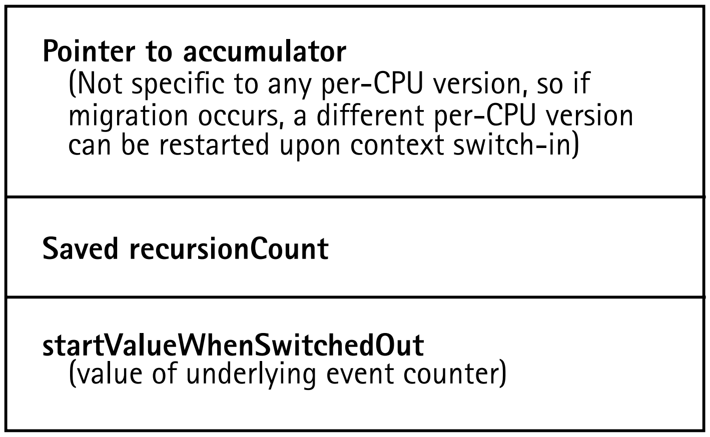

Pointer to accumulator
(Not specific to any per-CPU version, so if migration occurs, a different per-CPU version can be restarted upon context switch-in)
Saved recursionCount
startValueWhenSwitchedOut
(value of underlying event counter)

Figure 5.14: Hash Table Entry for Multiprocessor-Safe Wall Time Measurements
As before, an entry is specific to a particular thread. The startValueWhenSwitchedOut field is new. On context switch-in, the underlying event counter is re-read and subtracted from this field to obtain the number of events that occurred on this accumulator while blocked. This field can be added to the accumulator's total field. Alternatively, a separate total field can be maintained if the user chooses to view virtual time events and blocking time events separately.

design is that the user can measure an entirely new class of metrics, blocking time, to complement virtual time and wall time metrics.

A final benefit of this design is that even if several threads have overlapping wall time accumulation intervals (illustrated in Figure 5.2), they are not afflicted with the so-called "poll problem" (mentioned in Section 5.2) in which one thread can mistakenly stop accumulation on behalf of another thread. The problem is prevented because every wall time accumulator(s) that was started by thread T_1 but was not yet stopped has a recursionCount field that can only be accessed by that thread. In other words, there are thread-specific copies of each accumulator that is presently active. To see why this is true, consider two cases. In the first case, thread T_1 is blocked after having started a particular accumulator. Then a copy of the accumulator's recursionCount field at the time of switch-out is presently saved in the hash table, where no other thread has access to it. (Each hash table entry is indexed by thread ID, making it thread-specific.) In the other case, when T_1 is actively running, it has exclusive use of a recursionCount field for accumulator(s) that it started because there are per-CPU accumulator copies for running threads. Any other thread T_2 that is actively running and that has started the same (logical) accumulator must be on another processor, and is operating on a different per-CPU version of that accumulator, with a different recursionCount field.

5.6 Future Work

The primary limitation of kperfmon as presently implemented is that its instrumentation primitives do not work on multiprocessors. The designs of multiprocessor versions of virtual time metrics (Section 5.4.4) and wall time metrics (Section 5.5) are the most important areas of future work in kperfmon.

The design of metrics in kperfmon is powerful: the set of metrics is easily extended, and any metric can be virtualized. Resources could be made more powerful, however. Presently, the only code resources are kernel functions and basic blocks. It would be useful to add control flow graph edges to the set of possible resources, enabling inclusive time spent in a basic block to be divided into the if-taken and if-not-taken branches, for example. Similarly, call graph edges would allow measurement of the time spent in a function, but only when invoked from a specific call site. These fall under a more general umbrella: it would useful to have paths (at basic block or function granularity, intraprocedural and interprocedural) as resources.

Another useful extension to resources would be an entirely new hierarchy to complement code resources: data resources. This change would allow measurements such as “kernel lock (mutex) blocking time” for a specific mutex object, or “disk seek time” for a specific file.

5.7 Conclusion and Research Contributions

Kperfmon is the first tool that uses dynamic instrumentation to measure an unmodified commodity operating system kernel. It is also the first instrumentation-based tool that can instrument an unmodified kernel at a fine-grain: almost any kernel function or basic block can serve as a code resource, which in turn can be coupled with any kperfmon metric. (Sampling-based profilers can also measure at a fine-grain, but as discussed in Section 2.2.1, they cannot easily measure wall time metrics or inclusive metrics.)

Metrics in kperfmon are extensible; any monotonically increasing event counter can be leveraged in the creation of a new metric. Furthermore, metrics can accumulate either wall time events or virtual time events. Kperfmon is the first kernel profiler that can virtualize any wall time metric. The metrics in kperfmon are inclusive of time spent in callees (unlike sampling-based profilers), making an iterative (and possibly automated [14]) search for bottlenecks straightforward: simply traverse the call graph.

Chapter 6

Using Dynamic Kernel Instrumentation for Kernel and Application Tuning

This chapter presents a case study of using kperfmon to understand and optimize the performance of a Web proxy server. Understanding kernel performance has a two-way benefit, providing information useful for tuning both the kernel and user processes. The ease with which the performance results were obtained also shows the flexibility of a dynamic profiling tool that has full access to detailed performance data.

We first describe the Squid Web Proxy server and the benchmark used to drive it. We then describe the process of identifying the two major bottlenecks, followed by a description of the fixes and the resulting performance gain.

6.1 Introduction

This study used the Squid Web proxy server and the Wisconsin Proxy Benchmark as a workload to drive the kernel.

Web proxies are an effective means for reducing the load on Web servers. Web clients can query a local proxy (instead of the actual server), which adds a security firewall and caches some of the Web server's contents. The proxy retrieves files as needed from the server when it cannot satisfy a client's HTTP request from its cache.

We studied the performance of the Solaris 2.5.1[1] kernel, while running version 1.1.22 of the Squid Web proxy server [63]. Squid uses two levels of HTTP cache: in memory and on disk. Incoming HTTP GET requests are first searched in Squid's memory cache. If the object is not found there, Squid next tries its disk cache. If neither cache contains the desired object, Squid retrieves the object from the Web server. In this study, Squid's disk cache was installed on a local disk running the default Unix File System (UFS).

A heavily loaded proxy server can expect hundreds or thousands of simultaneous TCP connections, which it must multiplex alongside any local disk activity of its own. Squid does not create an independent thread of control to handle each request. Instead, a single thread multiplexes among all active TCP connections and pending file operations, using non-blocking I/O primitives.

Version 1.0 of the Wisconsin Proxy Benchmark [2] was used to drive Squid. Thirty synthetic client processes connect to a Squid proxy. Squid in turn connects to a synthetic Web server process. Three machines are used: client, Squid, and server. All measurements discussed in this chapter were performed on the Squid machine.

Each client process connects to Squid and makes HTTP GET requests with no intervening thinking time. The benchmark runs in two stages. In the first stage, 100

1. When this study was performed, KernInst had only been ported to Solaris 2.5.1.

requests are made for HTTP objects (files). The same file is never requested twice, so there is no locality in this stage. Its purpose is to populate the cache and stress its replacement algorithm. In the second stage, the client again sends 100 requests, but this time with a temporal locality pattern designed to lead to a proxy hit ratio of approximately 50%.

The server process listens on a particular TCP port number for HTTP requests. When one arrives, it parses the URL to determine the appropriate file. If the file has not yet been (synthetically) created, the server creates it. The file's size is uniformly distributed from 3 K to 40 K, 99% of the time. The other 1% of the time, the file size is 1 MB.

6.2 Understanding the Bottlenecks

The search for an application bottleneck (in Squid) led into the kernel, requiring a kernel profiling tool. We use kperfmon to measure the performance of the Solaris 2.5.1 kernel running the above benchmark. This section discusses two bottlenecks that kperfmon helped to identify. Measuring kernel performance yielded two benefits. First, it helped understand kernel bottlenecks, leading to a kernel optimization. Second, it helped understand a bottleneck caused by an application that spends most of its time in the kernel, performing I/O.

6.2.1 The First-Order Bottleneck: File Opens

Because the Wisconsin Proxy Benchmark has a working set size larger than Squid's in-memory cache, a reasonable hypothesis is that Squid might be disk bound[1] due to disk reads (misses in Squid's in-memory cache, but hits in its on-disk cache) or disk writes (misses in the on-disk cache, requiring the file to be brought in from the server). To obtain specific numbers on the time that Squid spends in disk activity, the Quantify profiling tool [77] was run on Squid to determine the source of this first-order bottleneck. Surprisingly, it turned out to be neither disk reads or writes.

Quantify showed that most of Squid's time is spent in the routine storeSwapOutStart, which is called to demote a file from Squid's in-memory cache to its on-disk cache. Interestingly, the time within this routine was spent not in writing to disk, but in its call to open. Squid was spending 76% of its elapsed time (time spent waiting in a select statement) simply opening on-disk cache files for writing. Explaining this result requires investigating Squid's cache organization.

Squid maintains one file per HTTP object that it is caching on disk. Thus, its on-disk cache is not one large fixed-size file, but a collection of all objects being cached, with varying sizes. Although it simplifies Squid's code, this design decision causes severe performance degradation.

Squid organizes its cache files into a three-deep hierarchy, to keep the number of files in any one directory manageable. A hash function maps a file table entry number to a full path name; the same file name can be reused for different cached objects over time. In general, a file used for an object that has been ejected remains on disk, though marked as invalid. When the file name is reused for a new cache object, Squid will overwrite the old invalid file.

1. The constant noise of disk thrashing on the machine running Squid led to the formulation of this hypothesis.

Any bottleneck in the open system call is a serious concern in a single-threaded program that multiplexes between many file descriptors, because UNIX has no interface for a non-blocking open system call. This limitation contrasts with read and write system calls, which can return EWOULDBLOCK if they would block the processing, allowing work to proceed on other, ready file descriptors. With blocking opens, the entire Squid process is blocked whenever an open on any file blocks.

Further performance study requires understanding *why* open was slow. This in turn requires examining the operations that take place in an open system call. User-level performance profilers see system calls as a black box, and can offer little guidance in understanding kernel performance.

6.2.2 Tracing the open Bottleneck to File Creation

With the dynamic nature of KernInst, one can interactively find kernel bottlenecks in the same manner as finding user-level bottlenecks. First, a function that is performing slowly is measured, to determine if it is a bottleneck. If so, its callees are similarly profiled. By using KernInst, this iterative refinement was performed while Squid was running.

The kernel routine copen was measured first. The results of two rate metrics, “calls made to” and “concurrency”, applied to copen, are shown in Figure 6.1. The figure shows that although file opens are occurring only 20-25 times per second, 40% of Squid’s elapsed time is spent in this routine.

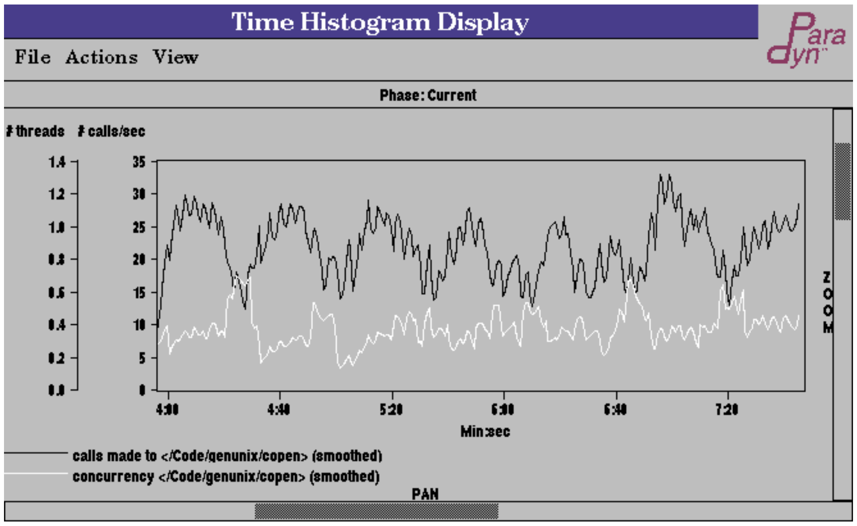


Figure 6.1: copen

Although called only 20-25 times/sec, copen is a clear bottleneck. On average, 0.4 kernel threads are executing in this routine at any given time. This number translates to 40% of Squid's elapsed time, since Squid is a single-threaded program.

Next, the actions performed by copen was examined. copen calls falloc to allocate an available entry in the process file descriptor table, and then calls vn_open to perform the open. Times for these routines are shown in Figure 6.2. falloc takes negligible time, a surprising finding because Squid file table allocation has been reported to be a bottleneck under heavy load in a previous study (performed on Digital UNIX) [8].

Since copen spent most of its time in vn_open, the performance of vn_open was examined next. Here, file creation diverges from opening an existing file; vn_create is called if the O_CREAT flag was passed to the open system call. vn_open is called about 20-25 times per second; of these, about eight go to vn_create. The remaining calls to vn_open are non-creating, indicating hits in Squid's on-disk cache. However, an examination of the latencies of these routines, shown in Figure 6.3, reveals that the

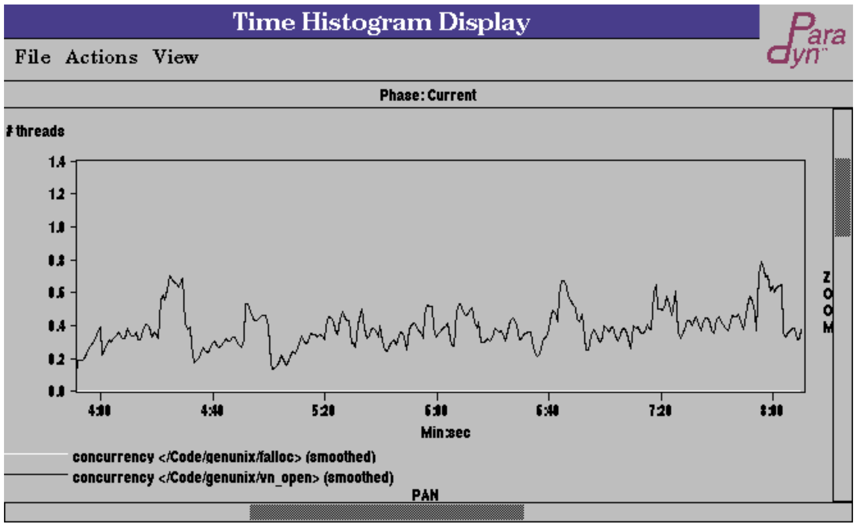


Figure 6.2: The Major Callees of copen: falloc and vn_open
vn_open accounts for almost all of the time spent in copen; negligible time is spent in falloc.

time spent in vn_open was almost entirely consumed by vn_create. Thus, eight file creations per second account for Squid's file-opening bottleneck.

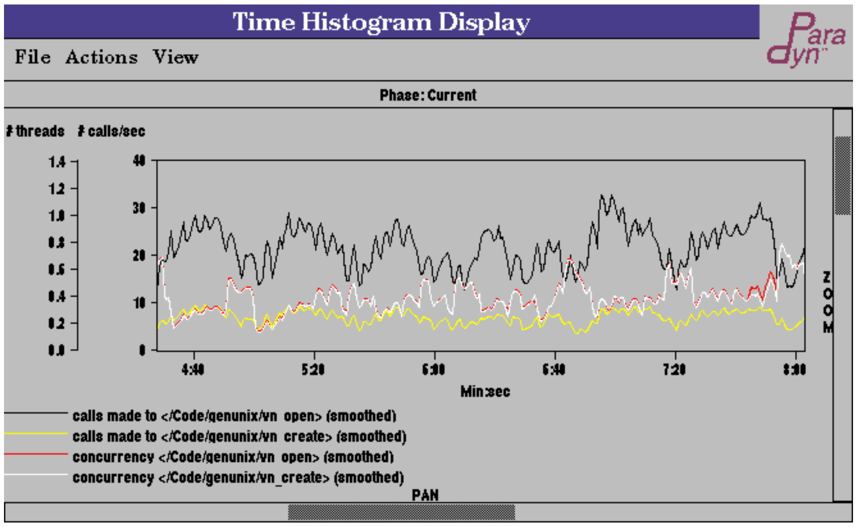


Figure 6.3: vn_open Spends Most of its Time in vn_create
The concurrency curves for vn_open and vn_create overlap (about 0.4 threads in them at any given time).

6.2.3 Understanding the File Creation Bottlenecks

Since file creation is the primary bottleneck, vn_create and its callees were examined. vn_create calls lookuppn (known in older UNIX incarnations as namei) to translate the file path name to a vnode. This vnode is passed to the file system-specific creation routine. The results, shown in Figure 6.4, reveal that file creation has two distinct bottlenecks: path name lookup (lookuppn) and UFS file creation (ufs_create).

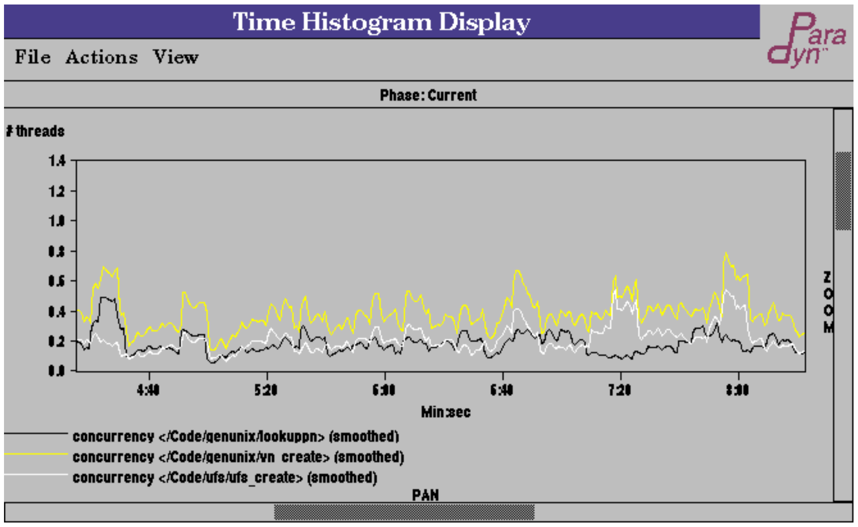


Figure 6.4: vn_create and Its Two Main Callees: lookuppn and ufs_create
Both lookuppn and ufs_create are (distinct) bottlenecks (each consuming about 20% of Squid's elapsed time).

6.2.3.1 The ufs_create Bottleneck

The ufs_create bottleneck was surprising, because UFS file creation performs only local meta-data operations. Yet, as just seen, Squid is spending about 20% of its time there. The time spent in ufs_create was traced to ufs_itrunc, which is invoked when the O_TRUNC flag is passed to the open system call. The two routines have nearly identical performance numbers, as shown in Figure 6.5. Thus, about 20% of Squid's elapsed time is spent truncating existing files to zero size when opening them.

To determine why ufs_itrunc is so slow, its callees were examined; the results are shown in Figure 6.6. Most of ufs_itrunc's time is spent in ufs_iupdat, which synchronously writes to disk any pending update to the file's inode. Thus, truncation

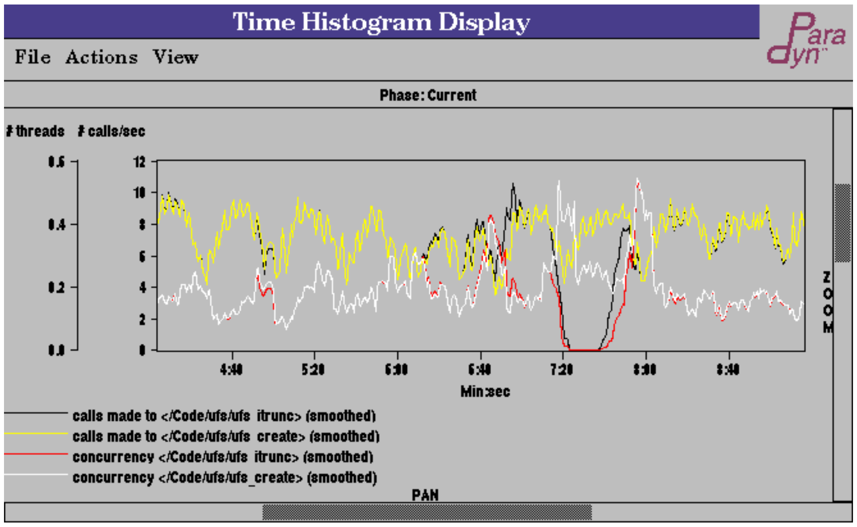


Figure 6.5: ufs_create Time is Spent Primarily in ufs_itrunc
UFS file creation time is dominated by inode truncation; the concurrency curves for ufs_create and ufs_itrunc almost completely overlap.

is slow because inode changes are made synchronously, as dictated by Unix file semantics to ensure file system integrity in case of a crash. Squid's strategy of overwriting existing cache files to avoid meta-data updates required by file deletion is ineffective, since similar meta-data updates are performed by truncation.

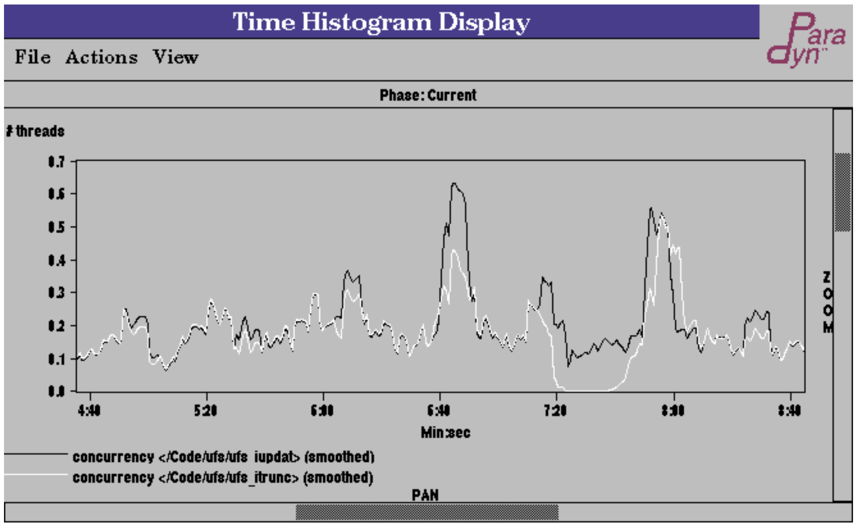
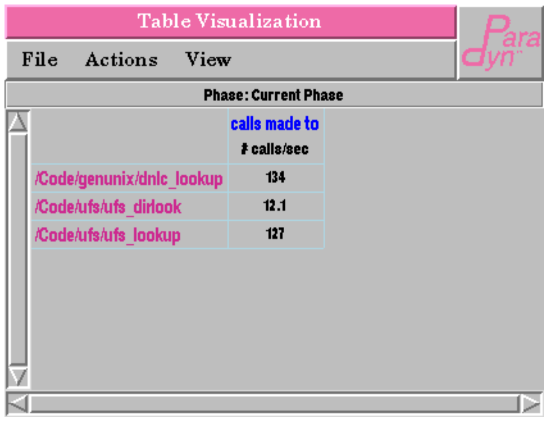


Figure 6.6: Most of ufs_itrunc's Time is Spent in ufs_iupdat
File truncation is slow because UFS meta-data updates are made synchronous by ufs_itrunc.

6.2.3.2 The lookuppn bottleneck

Recall from Figure 6.4 that lookuppn is a second bottleneck. To some extent, this is not surprising, since obtaining a vnode from a path name can require reading a directory file to obtain an inode disk location and reading the inode itself for each component in the path name. Solaris tries to optimize path name lookup through the directory name lookup cache, or DNLC [18, 58]. The DNLC is a hash table indexed by path name component containing a pointer to a cached inode. A hit in the DNLC allows the kernel to bypass disk reads for both the directory file (ufs_dirlook) and the inode (ufs_iget). As shown in Figure 6.7, the DNLC hit rate is about 90%. Unfortunately, the miss penalty (execution of ufs_dirlook) is sufficiently high to account for the ufs_lookup bottleneck, as shown in Figure 6.8.



Phase: Current Phase

	calls made to
	# calls/sec
/Code/genunix/dnlc_lookup	134
/Code/ufs/ufs_dirlook	12.1
/Code/ufs/ufs_lookup	127

Figure 6.7: The DNLC Hit Rate
The miss routine, ufs_dirlook, is only invoked once per 10 calls to ufs_lookup, for a DNLC hit rate of about 90%.

6.3 Optimizations Performed

This section discusses optimizations, one to the kernel and one to Squid, that respectively address the path-name lookup and file truncation bottlenecks.

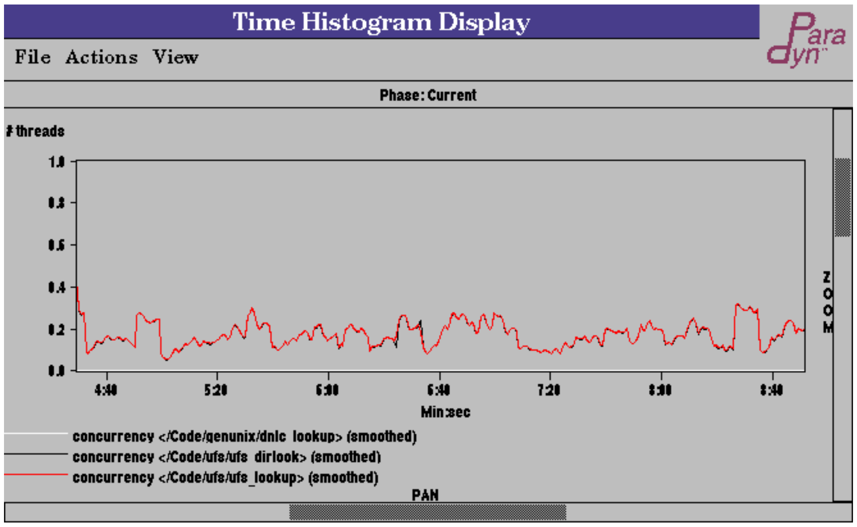

Time Histogram Display
File Actions View
Phase: Current
# threads
1.0
0.8
0.6
0.4
0.2
0.0
4:40
5:20
6:00
6:40
7:20
8:00
8:40
Min:sec
concurrency </Code/genunix/dnlc lookup> (smoothed)
concurrency </Code/ufs/ufs dirlook> (smoothed)
concurrency </Code/ufs/ufs_lookup> (smoothed)
ZOOM
PAN

Figure 6.8: ufs_lookup Spends Most of its Time in ufs_dirlook
Despite a low miss rate, the DNLC miss penalty (ufs_dirlook) is high enough to account for the entire ufs_lookup bottleneck (the ufs_dirlook and ufs_lookup curves almost completely overlap). The dnlc_lookup curve is essentially zero because checking for a DNLC hit or miss is always quick.

6.3.1 Addressing the Pathname Lookup Bottleneck

The DNLC contained a default of 2,181 entries on the test machine. With over 6,000 Squid cache files in the benchmark, plus hundreds of subdirectories in the three-level cache hierarchy, the DNLC was ineffective because Squid's preponderance of small files overwhelmed its capacity. Because Solaris sets the DNLC size based on the kernel variable maxusers, the bottleneck was addressed increasing maxusers to 2,048 in /etc/system (the maximum allowable value [18]) and rebooting. The DNLC size grew to 17,498 after this change.

The effect of increasing the DNLC size is shown in Figure 6.9. Once the benchmark has run long enough to warm up Squid's disk cache (about one minute), the DNLC miss rate, once 10%, drops to 1%. Correspondingly, the total time spent in the miss routine (ufs_dirlook) drops to a negligible fraction of Squid's running time. Best of all, ufs_lookup, and by implication, lookuppn, is no longer a bottleneck.

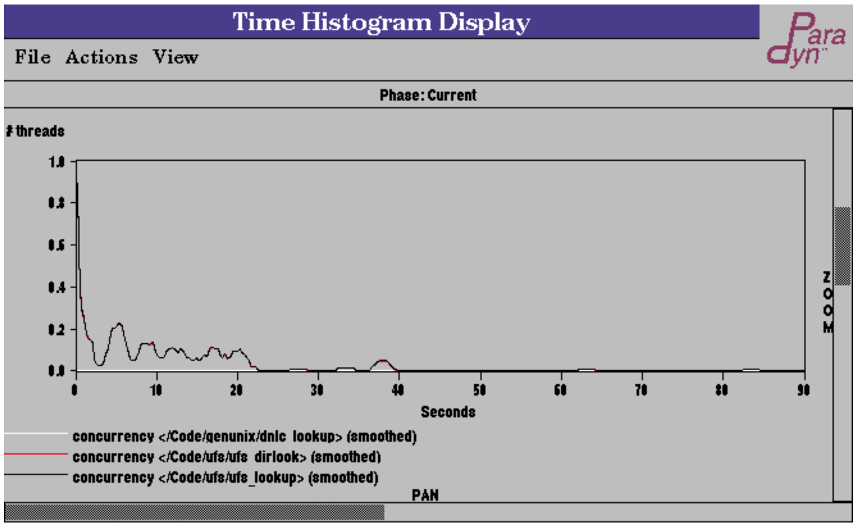


Figure 6.9: The Effect of Increasing DNLC Size on ufs_lookup Latency
For the first twenty seconds of the benchmark, there are enough DNLC misses to account for 10% of Squid's run time. As file names are reused more often, however, the DNLC hits become more frequent, and the ufs_dirlook bottleneck evaporates (compare to Figure 6.8).

6.3.2 Addressing the File Truncation Bottleneck

The second bottleneck, file truncation, was reduced by modifying Squid's source code. Squid has a bottleneck in ufs_create because it prepares to overwrite a file by first truncating its size to zero bytes. This involves updating the file's inode, which is done synchronously, in keeping with UFS semantics. Note that the truncation can be unnecessary, because data blocks will be added as the new version of the file is written. More specifically, if the file's new size is at least the original size, then the truncation was an expensive no-op, because every data block that is freed will be re-allocated. Even if the file is smaller than the one it is replacing, only the now-unused blocks at the end of the file need to be deleted.

Only a few small changes to Squid, totaling 15 lines of source code, were needed to effectively address this bottleneck. First, the O_TRUNC flag is no longer passed to the open system call. The new contents of the file are then written (Squid code that seeks to the end-of-file prior to every write was also removed). When the new file is written, an fcntl call with the parameter F_FREESP is used to truncate the file size at the present location of the file pointer. Thus, if the new file size is smaller than the previous file size, the blocks representing the now-unused end of the file are freed. If the new file size at least as large as the original file size, the fcntl has no effect.

After optimization, kperfmon was re-run to examine file creation performance. As shown in Figure 6.10, performance greatly improved. Less than 20% of Squid's time is now spent creating its disk cache files, compared with 40% before this optimization.

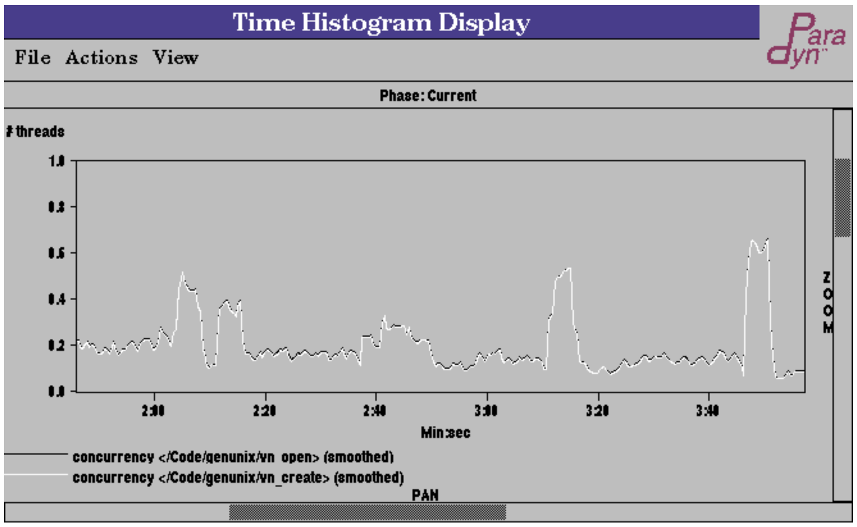


Figure 6.10: File Creation Performance When the Truncation Bottleneck is Addressed
File creation once took 40% of Squid's run time; the inode truncation optimization reduces it to 20%.

File truncation latency in the optimized version of Squid is shown in Figure 6.11. Calls to ufs_itrunc are no longer made by the open system call, because Squid no longer passes the O_TRUNC flag when opening cache files for writing. With the smarter truncation policy, time that Squid spends updating meta-data has reduced from about 20% to about 6%.

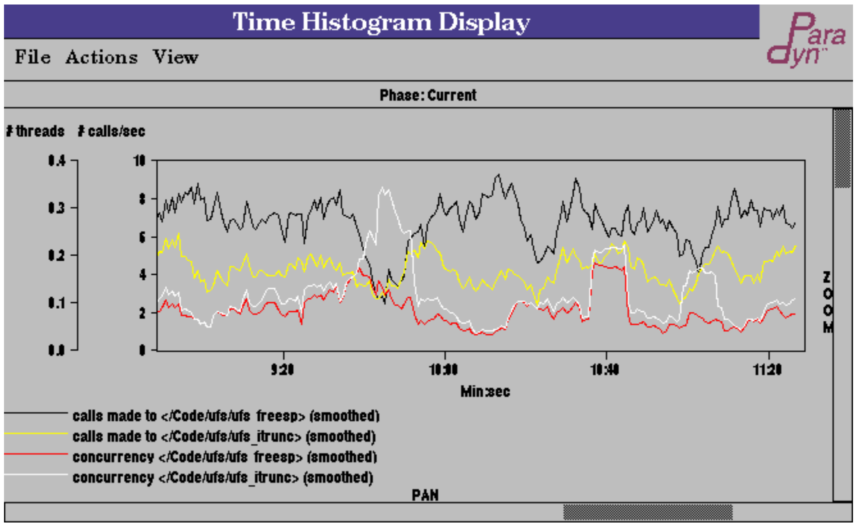

Time Histogram Display
File Actions View
Phase: Current
# threads # calls/sec
0.4
0.3
0.2
0.1
0.0
10
8
6
4
2
0
9:20
10:00
10:40
11:20
Min:sec
calls made to </Code/ufs/ufs_freesp> (smoothed)
calls made to </Code/ufs/ufs_itrunc> (smoothed)
concurrency </Code/ufs/ufs_freesp> (smoothed)
concurrency </Code/ufs/ufs_itrunc> (smoothed)
PAN
ZOOM

Figure 6.11: ufs_itrunc in Optimized Squid

Because Squid no longer uses the O_TRUNC flag opening cache files, truncation is now mostly performed when explicitly requested via fcntl (ufs_freesp).

6.3.3 The Combined Effects of Both Optimizations

As shown in Section 6.3.1, increasing the DNLC size reduced path name lookup time from about 20% of Squid's run-time to about 1%. In Section 6.3.2, avoiding unnecessary file truncating in Squid reduced UFS file creation time from about 20% of Squid's run-time to a negligible value. The combined effects of the two optimizations are shown in Figure 6.12. Total file creation time, which once took about 40% of Squid's run-time, now takes less than 1%. However, to this must be added the time spent explicitly truncating the file through the fcntl (ufs_freesp), which from Figure 6.11 is about 6%. File opens, which once took 40% of Squid's elapsed run-time now take about 7%, a speedup of 57%.

We re-ran Quantify on Squid, and found that only 15% of Squid's elapsed time, excluding idle time in select, was now being spent in storeSwapOutStart. This value is

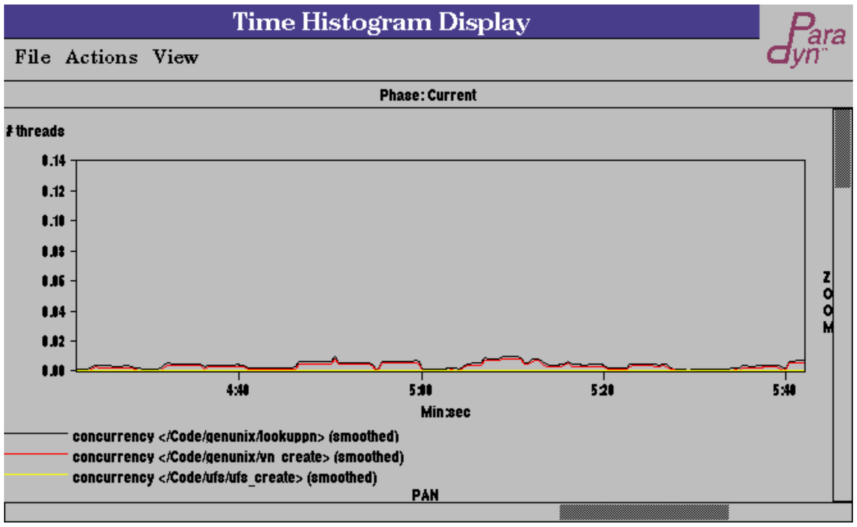


Figure 6.12: vn_create Time With Both the File Truncation and DNLC Optimizations Applied
vn_create once consumed 40% of Squid's run time (Figure 6.4); that number is now less than 1%.

down from 76% before optimizations. The first-order bottleneck in Squid is now write, taking 44% of Squid's non-idle elapsed time.

6.4 Ideas for Future Optimizations

Both of the kernel bottlenecks identified by kperfmon involve meta-data updates when opening a local disk file for writing. Although optimizations that have significantly reduced these bottlenecks have been presented, 7% of Squid's elapsed time is still spent waiting for the open system call to complete. A fundamental redesign of Squid's disk cache would further improve performance. Instead of one disk file per cached HTTP object, Squid should use one huge fixed-size file for its disk cache, managing the contents of this file manually. This design will bypass the UFS file system, with its corresponding synchronous meta-data update overheads.

While any sacrifice of synchronous meta-data updates can cause data loss, and thus is normally unacceptable, there is no need for this feature in a Web proxy server.

File corruption, assuming it can be detected, can easily be tolerated in a proxy server by re-fetching the affected file from the server. Similarly, Squid has no need for the UFS feature of synchronously updating time of last modification when writing files. Measurements described in this chapter suggest that bypassing UFS by managing Squid's disk cache manually would yield major performance improvements, though at the cost of significantly increasing Squid's code complexity.

6.5 Conclusion

This chapter has shown a demonstration of kperfmon to understand two bottlenecks in a heavily loaded Web proxy server. This case study has shown a two-way benefit from kernel measurement, providing information useful for both kernel and application tuning. One of the bottlenecks, poor DNLC performance, was addressed by changing a kernel variable. Another bottleneck, superfluous file truncating, was addressed by changing application code, which shows that kernel profiling is also useful to application developers. In both cases, optimizations were made possible through a detailed understanding of the kernel's inner workings provided by kperfmon. Without information on why the open system call had such high latency, it is unlikely that either optimization would have been found.

Chapter 7

Calculating Control Flow Graph Edge Counts From Block Counts

A simple and effective algorithm is presented for approximating control flow graph edge execution counts from basic block execution counts. Edge counts provide more precise profile information than block counts, and are useful in applications such as the on-line kernel I-cache optimization that is presented in Chapter 8. They may be of interest to other tools, especially sampling-based profilers such as dcpi [3], gprof [40], Morph [107], and VTune [46], which are not able to measure edge counts directly. The approximation of edge counts is effective in practice; measurements for the Solaris kernel in this chapter show that control flow graph edge counts can be calculated from block counts for more than 98% of the edges. Furthermore, counts can be calculated for every edge of 94% of kernel functions. These results show that simple instrumentation to measure block counts can be used in place of technically more difficult instrumentation that measures edge counts.

The algorithm requires a function’s control flow graph and the execution count of each basic block in the graph. It assigns counts to the graph’s edges, precisely calculated when possible, and approximated otherwise. The algorithm is based on a simple graph invariant: that the sum of a basic block’s predecessor edge counts equals the block’s edge count, which also equals the sum of the block’s successor edge counts. The success of the algorithm (how many edge counts can be precisely calculated as opposed to approximated) depends solely on the structure of the control flow graph.

Successfully deriving edge counts tends to contradict the widely held belief that while block counts can be derived from edge counts, the converse does not hold. Although this is certainly true in the general case—it has been proven that edge counts cannot be derived from block counts for arbitrarily structured graphs [71]—the algorithm seems to work well in practice.

7.1 Motivation

There are many uses for edge counts. The run-time kernel I-cache optimization described in Chapter 8 uses edge counts to guide the layout of basic blocks that results in straight-lined execution in the common case. Another use of edge counts is in calculating how often a certain function was executed, which requires edge counting if the function’s entry basic block can be reached not only via a procedure call, but also via a branch from elsewhere in the function. The second use is discussed further in Section 7.4.

Avoiding the need to measure edge counts directly defers the need for an edge splicing primitive within KernInst. Edge splicing is presently un-implemented in kerninstd due to jump tables. If a jump table destination block can also be reached from a branch, then edge instrumentation cannot simply be placed at the entry to that block. A general solution might rewrite the jump table data so that execution of a particular jump table edge jumps to a code patch, which performs desired instrumentation and then jumps to the destination block.

7.2 The Algorithm

To obtain edge counts, two simple formulas are used: the sum of a basic block's predecessor edge counts equals the block's count, which also equals the sum of a basic block's successor edge counts. Therefore, for a block whose count is known, if all but one of its predecessor (successor) edge counts are known, then the unknown edge count can be precisely calculated: the block count minus the sum of the known predecessor (successor) edge counts.

The edge count assignment algorithm (Figure 7.1) is applied to one function at a time. The algorithm is given the function's control flow graph, and counts for each of its basic blocks. A set of basic blocks, containing blocks that might have only one unknown predecessor or successor edge count, is initialized to include all basic blocks in the function. The algorithm repeats the following actions until that set is empty. Some basic block *X* is removed from the set. If only one of its predecessor edges *(W, X)* has an unknown count, then its count is calculated and *W* is added to the set, because with this change, perhaps only one of its successor edge counts remains unknown. Next, a similar action is performed for the successors of block *X*. When the first phase of the algorithm completes, all the edge counts that could be precisely derived from the block counts were so calculated.

```
// Calculate precise edge counts, where possible:
BlockCount(X): the # of executions of basic block X
EdgeCount(X,Y): the # of executions of the control flow edge from block X to block Y
Preds(X): the set of predecessor blocks to basic block X
Succs(X): the set of successor blocks to basic block X

initialize set S to all basic blocks in the CFG
while ($S \neq \emptyset$) {
  remove any entry X from set S
  for all $W \in$ Preds(X) {
    if edge (W,X) is only pred of X with unknown exec count {
      EdgeCount(W,X)=BlockCount(X)- $\sum_{V \in \mathrm{Preds}(X) - \{W\}}$ EdgeCount(V, X)
      $S = S \cup \{W\}$
    }
  }
  for all $Y \in$ Succs(X) {
    if edge (X,Y) is only succ of X with unknown exec count {
      EdgeCount(X,Z)=BlockCount(X)- $\sum_{Z \in \mathrm{Succs}(X) - \{Y\}}$ EdgeCount(X, Z)
      $S = S \cup \{Y\}$
    }
  }
}
// Approximate the remaining edge counts:
KnownSuccs(X): the set of successors to block X connected via an edge whose
               count was precisely calculated in the above phase of the algorithm.
KnownPreds(X): the set of predecessors to block X connected via an edge whose
               count was precisely calculated in the above phase of the algorithm.

for all edges (X,Y) in the CFG with unknown count {
 max1 = BlockCount(X) - $\sum_{Z \in \mathrm{KnownSuccs}(X)}$ EdgeCount(X, Z)

 max2 = BlockCount(Y) - $\sum_{W \in \mathrm{KnownPreds}(Y)}$ EdgeCount(W, Y)

 EdgeCount(X,Y) = min(max1, max2)
}
```

Figure 7.1: Pseudo-Code For Deriving Edge Counts From Block Counts

Based on two graph invariants, that both the sum of a node's predecessor edge counts and the sum of its successor edge counts equals the node's block count, the first phase of this algorithm precisely calculates edge counts, where possible. The second phase approximates the remaining edge counts.

In the second phase of the algorithm, imprecise execution counts are assigned to the remaining edges. Two formulas provide a bound on the count of such an edge *(X, Y)*. Let *KnownSuccs(X)* be the successors blocks of *X* that are connected to *X* by an edge count that was precisely calculated in the first phase. Let *UnknownSuccs(X)* be *Succs(X) - KnownSuccs(X)*. We start with the invariant that the sum of *X*'s successor edge counts equals *X*'s block count, then subtract the sum of *X*'s *known* successor edges from both sides to yield

$$\sum_{Y \in \mathrm{UnknownSuccs}(X)} \mathrm{EdgeCount}(X, Y) = \mathrm{BlockCount}(X) - \sum_{Y \in \mathrm{KnownSuccs}(X)} \mathrm{EdgeCount}(X, Y)$$

Similarly, let *KnownPreds(Y)* be the predecessor edges of *Y* whose counts are known, and let *UnknownPreds(Y)* be *Preds(Y)-KnownPreds(Y)*. The sum of *Y*'s predecessor edge counts equals *Y*'s count. Subtracting the sum of *Y*'s *known* predecessor edges from both sides yields

$$\sum_{X \in \mathrm{UnknownPreds}(Y)} \mathrm{EdgeCount}(X, Y) = \mathrm{BlockCount}(Y) - \sum_{X \in \mathrm{KnownPreds}(Y)} \mathrm{EdgeCount}(X, Y)$$

Each of the two equations give a maximum bound on the execution count of edge *(X, Y)*. The minimum of the maximum values allowed by the two equations provides an upper bound on the count of edge *(X, Y)*. We use this value as an estimate for the count of edge *(X, Y)*. Although this assignment is usually more accurate than assigning the block count of the edge's destination, the value can still be too high. There are alternatives to this policy, such as evenly dividing the maximum allowable value among the unknown edges. However, since it is usually possible in practice to usually derive precise edge counts, approximation is seldom needed, making the issue relatively unimportant.

To give an example of the algorithm at work, Figure 7.2 (from Pettis and Hansen's paper [69]) contains a control flow graph that was used to demonstrate why edge measurements are more useful than node measurements. (Pettis and Hansen did not attempt to calculate edge counts from block counts, noting only that it is not always possible. From this, they assume that tools will resort to using the block count of an edge's destination as a poor approximation for the edge itself.) In this example, edge counts can be derived as follows. First, *B* has only one predecessor edge and only one successor edge, each of which must equal *B*'s count, so edges *(A, B)* and *(B, D)* are assigned a count of 1000. Now, *A* has only one unknown successor edge *(A, C)*, which is given the count of 1 (block *A*'s count, 1001, minus the count of its known successor edge, 1000). Next, C has only one remaining unknown predecessor edge *(C, C)*, which can be assigned the value 2000 (*C*'s block count, 2001, minus the count of the other predecessor edge, 1). Finally, *C* now has one unknown successor *(C, D)*, which can be assigned 1 (*C*'s block count, 2001, minus *C*'s other successor edge, 2000).

7.3 Results

Figure 7.3 contains the results of an experiment to determine how often edge counts can be calculated from block counts. The results are encouraging:

- Across the entire kernel, over 98% of edge counts can be determined from block counts; approximation was needed for less than 2% of edges.
- Over 94% of kernel functions have *all* of their edge counts accurately determined from block counts without the need for approximation.

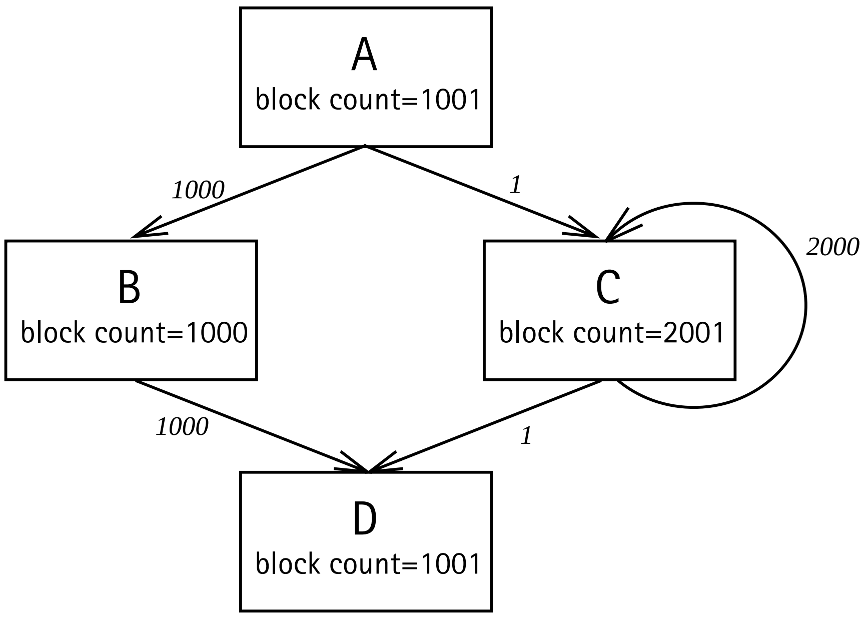


Figure 7.2: An Example Where Edge Counts Can Be Derived From Block Counts
An unknown count for an edge (X, Y) can be calculated if it is the only unknown successor of block X, or the only unknown predecessor of block Y. Repeated application of this rule until convergence can often calculate all edge counts, as in this example. (An augmented version of Figure 3 from [69].)

Module	#Fns parsed	#Edges	Frac. of individual edges whose counts can be calculated	Frac. of fns whose entire edge counts can be calculated
genunix	2589	51225	0.991	0.959
afs (afs syscall interface)	922	27663	0.919	0.678
unix	1758	20288	0.988	0.968
ufs (filesystem for ufs)	337	10840	0.996	0.967
nfs (NFS syscall, client, and common)	479	10279	0.998	0.987
ip (IP Streams module)	373	11437	0.991	0.957
md (Meta disk base module)	390	8812	0.991	0.967
tcp (TCP Streams module)	159	5889	0.997	0.969
procfs (filesystem for proc)	174	5475	0.993	0.948
sd (SCSI Disk Driver 1.308)	115	4737	0.998	0.983
rpcmod (RPC syscall)	209	3972	0.997	0.986
sockfs (filesystem for sockfs)	149	3891	0.998	0.987
pci (PCI Bus nexus driver)	127	2523	0.987	0.937
hme (FEPS Ethernet Driver v1.121)	97	2741	1.000	1.000
se (Siemens SAB 82532 ESCC2 1.93)	69	2731	0.997	0.971
fd (Floppy Driver v1.102)	54	2582	1.000	1.000
zs (Z8530 serial driver V4.120)	48	2147	1.000	1.000
uata (ATA AT-bus attachment disk controller Driver)	127	1936	1.000	1.000
krtld	127	2363	0.984	0.921
rpcsec (kernel RPC security module.)	122	2037	0.996	0.984

Figure 7.3: How Often Edge Counts Can Be Calculated From Block Counts
This table, ordered by module total code size, shows that edge counts can usually be calculated from block counts. The fourth column gives the fraction of individual edges whose count can be determined. The final column gives the fraction of functions for which every edge can be determined. The last row gives kernel-wide totals.

Module	#Fns parsed	#Edges	Frac. of individual edges whose counts can be calculated	Frac. of fns whose entire edge counts can be calculated
ufs_log (Logging UFS Module)	131	1879	0.991	0.969
xfb (xfb driver 1.2 Sep 7 1999 11:46:39)	99	1855	0.998	0.990
audiocs (CS4231 audio driver)	83	1486	1.000	1.000
dad (DAD Disk Driver 1.16)	56	1558	0.992	0.982
tmpfs (filesystem for tmpfs)	66	1422	0.997	0.985
ldterm (terminal line discipline)	45	1768	0.989	0.911
afb (afb driver v1.36 Sep 7 1999 11:47:45)	60	1023	0.992	0.983
scsi (SCSI Bus Utility Routines)	78	1330	0.994	0.974
tl (TPI Local Transport Driver - tl)	57	1399	1.000	1.000
specfs (filesystem for specfs)	48	1030	0.988	0.938
arp (ARP Streams module)	67	1397	1.000	1.000
SUNW,UltraSPARC-IIi	60	910	1.000	1.000
vol (Volume Management Driver, 1.85)	23	1184	0.997	0.957
doorfs (doors)	52	1113	1.000	1.000
su (su driver 1.24)	31	922	0.991	0.935
udp (UDP Streams module)	43	1131	1.000	1.000
timod (transport interface str mod)	36	1054	0.989	0.917
kerninst (kerninst driver v0.4.1)	116	1005	0.996	0.991
fifofs (filesystem for fifo)	40	920	0.996	0.975
kb (streams module for keyboard)	36	984	0.996	0.972
tnf (kernel probes driver 1.47)	55	780	0.959	0.873
pm (power manager driver v1.65)	29	806	1.000	1.000
TS (time sharing sched class)	36	787	1.000	1.000
devinfo (DEVINFO Driver 1.24)	35	668	0.982	0.914
ipdcm (IP/Dialup v1.9)	52	816	0.995	0.981
ttcompat (alt ioctl calls)	14	692	1.000	1.000
diaudio (Generic Audio)	28	772	0.995	0.964
elfexec (exec module for elf)	12	453	1.000	1.000
shmsys (System V shared memory)	19	515	1.000	1.000
ptc (tty pseudo driver control ‘ptc’)	16	638	1.000	1.000
tlimod (KTLI misc module)	21	557	1.000	1.000
winlock (Winlock Driver v1.39)	39	448	0.991	0.974
hwc (streams module for hardware cursor support)	11	382	1.000	1.000
ms (streams module for mouse)	17	468	1.000	1.000
ptem (pty hardware emulator)	13	459	1.000	1.000
simba (SIMBA PCI to PCI bridge nexus driver)	18	274	1.000	1.000
seg_drv (Segment Device Driver v1.1)	25	339	1.000	1.000
sad (Streams Administrative driver’sad’)	23	371	1.000	1.000
namefs (filesystem for namefs)	32	271	0.985	0.969
lockstat (Lock Statistics)	27	327	1.000	1.000
ptsl (tty pseudo driver slave ‘ptsl’)	14	335	1.000	1.000
rootnex (sun4u root nexus)	19	361	1.000	1.000
dada (ATA Bus Utility Routines)	30	339	1.000	1.000
dada_ata (ATA Bus Utility Routines)	30	330	1.000	1.000
md5 (MD5 Message-Digest Algorithm)	8	46	1.000	1.000
sysmsg (System message redirection (fanout) driver)	15	297	1.000	1.000
mm (memory driver)	13	272	1.000	1.000
wc (Workstation multiplexer Driver ‘wc’)	18	292	1.000	1.000
ebus (ebus nexus driver)	13	213	1.000	1.000

Figure 7.3: How Often Edge Counts Can Be Calculated From Block Counts

This table, ordered by module total code size, shows that edge counts can usually be calculated from block counts. The fourth column gives the fraction of individual edges whose count can be determined. The final column gives the fraction of functions for which every edge can be determined. The last row gives kernel-wide totals.

Module	#Fns parsed	#Edges	Frac. of individual edges whose counts can be calculated	Frac. of fns whose entire edge counts can be calculated
ptm (Master streams driver ‘ptm’)	12	234	1.000	1.000
pts (Slave Stream Pseudo Terminal driver ‘pts’)	12	229	1.000	1.000
RT (realtime scheduling class)	24	225	1.000	1.000
iwscn (Workstation Redirection driver ‘iwscn’)	19	199	0.980	0.947
fdfs (filesystem for fd)	17	195	1.000	1.000
eide (PC87415 Nexus driver v2.0)	19	180	1.000	1.000
conskbd (Console kbd Multiplexer driver ‘conskbd’)	14	189	0.979	0.929
todmostek (tod module for Mostek M48T59)	10	56	1.000	1.000
log (streams log driver)	9	134	0.970	0.889
sy (Indirect driver for tty ‘sy’)	12	123	1.000	1.000
consms (Mouse Driver for Sun ‘consms’)	14	166	0.976	0.929
kstat (kernel statistics driver)	12	131	1.000	1.000
pckt (pckt module)	11	153	1.000	1.000
ksyms (kernel symbols driver)	11	97	1.000	1.000
inst_sync (instance binding syscall)	11	95	1.000	1.000
power (power driver v1.4)	11	104	1.000	1.000
cn (Console redirection driver)	13	99	1.000	1.000
sysacct (acct(2) syscall)	6	74	0.946	0.833
clone (Clone Pseudodriver ‘clone’)	7	82	1.000	1.000
intpexec (exec mod for interp)	5	85	0.953	0.800
pseudo (nexus driver for ‘pseudo’)	10	74	0.946	0.900
ipc (common ipc code)	5	61	1.000	1.000
pipe (pipe(2) syscall)	4	21	1.000	1.000
connld (Streams-based pipes)	6	28	1.000	1.000
options (options driver)	7	39	1.000	1.000
redirmod (redirection module)	6	21	1.000	1.000
TS_DPTBL (Time sharing dispatch table)	5	10	1.000	1.000
IA (interactive scheduling class)	3	9	1.000	1.000
RT_DPTBL (realtime dispatch table)	3	6	1.000	1.000
platmod	5	10	1.000	1.000
Kernel-wide:	10637	225375	0.984 (std dev:0.001)	0.944 (std dev:0.004)

Figure 7.3: How Often Edge Counts Can Be Calculated From Block Counts

This table, ordered by module total code size, shows that edge counts can usually be calculated from block counts. The fourth column gives the fraction of individual edges whose count can be determined. The final column gives the fraction of functions for which every edge can be determined. The last row gives kernel-wide totals.

Kerninstd currently calculates edge counts for every (parsed) kernel function, using approximation where necessary, in about 2.4 seconds.

7.4 Obtaining Number of Calls to a Function From Its Block Counts

In this section it is shown how to use edge counts to accurately obtain the number of times a function is called. Perhaps surprisingly, the number of calls to a function is not always simply the number of times that its entry basic block is executed, because the entry block may also be the destination of a branch from within the function. A function beginning with a while loop may have such a structure. Kperfmon calculates an accurate function call count by subtracting the execution count of the function's entry block predecessors from the execution count of the entry basic block itself.

Of the kernel functions whose control flow graphs were successfully parsed, 70 functions have an entry block with at least one predecessor edge, complicating the calculation of number of times these functions are called. Of these 70 functions, only three (blkleft, copyout_blkleft, and copyin_blkleft) have an entry block whose predecessor edge counts could not all be derived from the block counts.

7.5 Future Work

The vast majority of kernel functions can accurately derive their edge execution counts from their block execution counts. It would be useful to know if, in turn, the derived edge counts can be used to approximate (intra-procedural) path execution counts. A paper by Ball, Mataga, and Sagiv contains theory and results for determining when edge profiles are accurate predictors of path profiles [7].

It would be useful to employ previous research into block counting with lower perturbation. Knuth and Stevenson present an algorithm for finding the minimum number of block counters that are necessary and sufficient for block execution counting [49].

Ball and Larus show that it is less expensive to determine block counts via edge profiling, in terms of minimizing the total number of instrumentation points [6]. This would tend to argue that once edge splicing is implemented in kerninstd, it should discard the mechanisms of this chapter in favor of measuring edge counts directly. However, basic block execution count measurements can benefit from an optimization: instrumentation code can be placed anywhere in the basic block. Among the choices, there may be one with enough available scratch registers to execute the instrumentation code without spilling register(s), something which is not considered in Ball and Larus' work. Furthermore, as demonstrated in Section 4.2.3, dynamic instrumentation at conditional branches—which would be frequently used in edge profiling—is one of the more expensive places to splice, due to the expense of relocating the overwritten instruction.

A paper by Probert [71] contains proofs that basic block counts are insufficient to determine edge counts not only for arbitrary control-flow graphs, but also for reducible graphs [50] (ones whose loops have only a single entry point). However, Probert shows that for "well-delimited" programs, whose control flow constructs (`if`, `while`, etc.) all have corresponding delimiters (`end if`, `end while`, etc.), a source code transformation can insert basic block counts which are sufficient to derive edge counts. Furthermore, it is shown how the set of basic block counters that are necessary for deriving edge counts can be minimized. When viewed at the machine code level, Probert's algorithm only works on functions whose control-flow graphs match that of such "well-delimited" functions. A non-matching function would require the addition and instrumentation of dummy basic blocks that are placed at edges in the original graph. The set of functions for which edge counts can precisely derived from block counts may be isomorphic to the set of functions to which

Probert's algorithm applies. If this holds, then Probert's work could be leveraged, reducing the number of basic blocks that need instrumenting to a minimum.

The techniques of this chapter may be usefully applied to inter-procedural edge counting; in other words, to obtain call graph edge counts. Call instructions are rarely conditional, so most call graph edge counts are simply the count of the basic block containing the calling instruction. An exception occurs for calls made via inter-procedural branches. If such a branch is conditional, then an edge count, the number of executions of the if-taken case, determines the call graph edge count. Calls whose destination cannot be statically determined, such as a call through a function pointer, will be the most difficult to measure, because the call instruction can have any number of callees, each of which are a control flow graph edge.

7.6 Conclusion and Research Contributions

An algorithm has been presented for determining edge counts from block counts, and applied to the Solaris 7 kernel. The results show that 98.4% of edges kernel can be precisely determined, and that 94.4% of kernel functions can have *every* one its edges so determined. However, the positive results are not meant to imply that edge splicing is not useful, for if it were available, then all edge counts could be calculated.

As a means of calculating edge counts that is independent of the mechanism used to measure block counts, the work shown in this chapter is especially beneficial to tools that cannot directly measure edge counts, particularly sampling-based profilers.

Chapter 8

Dynamic Kernel Optimization and Evolving Kernels

This chapter presents mechanisms for turning an unmodified commodity kernel into an *evolving* one. An evolving system is one that changes its code dynamically to adapt to the run-time environment. Such changes can include, but are not limited to, dynamic optimizations, security patches, and bug fixes. The ideal evolving system allows every step of the development cycle to be performed on-line. As a proof of concept, an evolving kernel algorithm has been implemented within kperfmon. This algorithm is a dynamic kernel version of Pettis and Hansen's code positioning I-cache optimizations [69]. There are two contributions from this implementation. First, it is the first general-purpose infrastructure for dynamic kernel optimization, which provides evidence that an unmodified commodity kernel can be made into an evolving one. And second, it is the first on-line kernel version of this optimization. As an initial case study, dynamic code positioning reduced by 17.6% the execution time of the kernel's TCP read-side routine (tcp_rput_data) while running a Web mirroring benchmark.

8.1 Evolving Kernels and Run-time Kernel Code Positioning

Performance tuning and debugging are performed in a cycle of several steps. Some of the steps, such as performance measurement or bug finding, can take place at run-time, but program modifications are typically installed off-line. For an operating system kernel, installing off-line changes requires a reboot, which can be prohibitive. To address this problem, a framework for *evolving kernels* has been designed and implemented. The framework allows a kernel to dynamically change its code to adapt to the run-time environment. Although the framework applies equally well to evolving user-level code, the current implementation is done within KernInst.

An evolving kernel algorithm is driven by a policy that will make use of the following three mechanisms:

- **Measurement.** Data is gathered that identifies a problem, such as a bug or a performance bottleneck. Kperfmon is used for this step.
- **Change the code.** Kernel code is modified to try to solve the problem; the mechanism depends on the evolving algorithm. For bug fixing, a programmer typically manually changes the source code. Certain performance bottlenecks may be solved with automated transformations, such as that provided by a feedback-directed compiler [9, 17, 19, 69, 89]. Kerninstd does not implement specific policies. Instead, it accepts and installs externally generated code changes. This is in keeping with kerninstd’s policy of providing the mechanisms for inserting code into a running kernel, while allowing the code to be generated by any outside source. This chapter describes a specific mechanism for dynamic code positioning to the Solaris 7 kernel.
- **Install the new code.** The third step of an evolving algorithm installs the modified code into a running system. Kerninstd’s code replacement

primitive (Section 4.5) is used for this purpose, which allows an outside source to change the code of an entire function dynamically.

An evolving algorithm (whether in the kernel or in a user-level program) is more general than *dynamic feedback* [32], which chooses between several distinct policies at run-time. A dynamic feedback system hard-codes all of its components; the logic for driving the adaptive algorithm, the measurement code, all policies, and the logic for switching between the policies at run-time are compiled into the system *a priori*. An evolving system does not hard-wire these components.

As a demonstration of the mechanisms necessary to support an evolving kernel, we have implemented a run-time prototype of three I-cache optimizations due to Pettis and Hansen [69]:

- **Procedure splitting.** Also called *outlining* in Scout [62], this optimization moves seldom-executed “cold” basic blocks away from “hot” ones, to reduce the I-cache pollution of rarely executed code. Such code is prevalent in the kernel, due to extensive error checking.
- **Basic block positioning.** A function’s blocks are reordered to facilitate straight-lined execution in the common case. Advantages include increasing the code executed between taken conditional branches, decreasing I-cache internal fragmentation due to un-executed code that shares a line with common code, and a reduction of unconditional branches. In Pettis and Hansen’s work, block positioning yielded the greatest benefit.
- **Procedure positioning.** This optimization places the code of functions that exhibit temporal locality adjacent in memory, to reduce the chances of I-cache conflict misses.

The three optimizations are collectively called *code positioning*. Pettis and Hansen implemented them in a feedback-directed customized compiler for user code, which applies the optimizations off-line and across an entire program. In contrast, our implementation is performed on kernel code and entirely at run-time. Also, only a selected set of functions is optimized, not the entire kernel.

Code positioning is performed using the following steps:

- A function to optimize is chosen (this is the only step currently requiring user involvement).
- Kperfmon determines if there is an I-cache bottleneck at that function. If so, block counts are gathered for this function and those of its descendants that it calls frequently. From these counts, a determination is made on which functions to optimize (the chosen function and a subset of its descendants). (Section 8.2)
- Kerninstd parses a relocatable representation of these functions. (Section 8.3)
- An optimized re-ordering is chosen and installed into a running kernel. (Section 8.4)

Interestingly, once the optimized code has been installed, the entire code positioning optimization must be repeated (once)—optimizing the optimized code— for reasons that are discussed in Section 8.2.3.

8.2 Kernel Code Positioning: Measurement

Our kernel code positioning optimization starts with the measurement phase. The user specifies a function that is to be evaluated for optimization. The first measurement step, described in Section 8.2.1, determines whether an I-cache bottleneck is present. If so, then code positioning is performed.

Because a function’s callees affects its I-cache performance, code positioning is performed not only on the user-specified function, but also on some of its call graph descendants. This collective set of functions is called a *function group*; the user-specified function is the group’s *root function*. The second measurement step, described in Section 8.2.2, collects block execution counts for the root function and a subset of its call graph descendants.

The intuitive basis for the function group is to gain control over I-cache behavior while the root function is executing: once the group is entered via a call to the root function, control will probably stay within the group until the root function returns. This gives us knowledge about the likely flow of control that can be leveraged to improve I-cache performance during this time.

8.2.1 Is There An I-Cache Bottleneck?

The first step of measurement checks whether code positioning is needed. Kperfmon applies the metric I-cache vstall time/vtime to the group’s root function, which yield the fraction of that function’s virtual time in which the CPU is stalled due to an I-cache miss. The denominator is virtual time, an inclusive, virtualized interval count metric (see Section 5.4) whose underlying event counter is the on-chip elapsed cycle counter. The numerator is similar, with the underlying event counter of the on-chip I-cache stall cycle counter [96, 97]. (Because an I-cache miss does not cause a context switch, I-cache miss cycles are included in a function’s virtual time.) If the ratio is above a threshold (which is arbitrarily set to 10% by default and can be changed by the user), then the algorithm continues, on the assumption that I-cache performance might benefit from code positioning.

The UltraSPARC processor, like most modern processors, decouples instruction fetching from instruction execution by executing instructions out of an instruction buffer, which is filled via prefetching out of the I-cache. This decoupling can tolerate some I-cache miss time by executing other instructions in the instruction buffer. Accordingly, the UltraSPARC performance register for I-cache stall cycles measures the number of cycles that the CPU spends idling due to an empty instruction buffer; the cause will be an I-cache miss whose miss handling latency could not be fully tolerated.

Like all of kperfmon’s interval measurements, both the number of virtualized I-cache stall cycles and the number of virtualized cycles are inclusive measurements. Therefore, any time that the root function spends in its descendant functions (and any I-cache misses that occur during such times) while the root function is on the call stack are included in the measurements. This inclusion is desirable because it takes into account the descendants’ effect on the root function’s I-cache behavior.

8.2.2 Collecting Block Execution Counts for Functions

The second phase of measurement performs a breadth-first traversal of the call graph, collecting basic block execution counts of any function that is called at least once while the root function is on the stack.

The traversal begins by instrumenting the root function to collect its basic block execution counts. After a user-defined delay to let a benchmark run (the default is 20 seconds), the instrumentation is removed and the block counts are examined. Among the blocks that were executed at least once, statically identifiable callee functions have their block counts measured in the same way.

Pruning is applied to the call graph traversal in two cases. First, a function that has already had its block execution counts collected is not re-measured. Second, a function that is only called from within a basic block whose execution count is zero is not measured.

Because indirect function calls (such as through a function pointer) do not appear in the call graph, a function reached only via such calls will not have its block counts measured. Such functions will not have a chance be included in the optimized group.

8.2.3 Measuring Block Execution Counts Only When Called from the Root Function

Block execution count instrumentation should include only those executions when the root function is on the call stack, to only consider the extent to which a descendant function contributes to the root function's I-cache behavior. In particular, a descendant may be called not only when the root function is on the call stack, but also from other kernel functions having nothing to do with the root function. Block executions in the latter case should not be counted.

This more selective block counting is achieved by performing code positioning twice—re-optimizing the optimized code. (The first time, the group is generated using block counts that were probably too high.) Code replacement is performed solely on the root function, so the non-root group functions only are invoked while optimized root function is on the call stack. This technique ensures that block execution counts during re-optimization include only executions when the root function is on the call stack, as desired.

Collecting block counts in a single pass, without re-optimization, could require predicating block counting instrumentation code with a test for whether the code was called (directly or indirectly) from root function. Paradyn uses such “constraint” predicates in user-level instrumentation by keeping instrumentation that increments a counter when the root function is entered, and decrements that counter when the root function exits [43, 44]. The counter acts as a flag that can be tested by block counting instrumentation. (The counter is non-zero when the root function is on the call stack.) However, beyond the extra run-time cost, thread-safety requires per-thread flags, with corresponding extra complexity [104].

8.3 Obtaining A Relocatable Representation of the Group’s Code

After the measurement phase of the optimization, kerninstd parses each of the group functions into a relocatable representation. This representation allows an optimized version of a function to be re-emitted with arbitrary basic block ordering, even to the point of interleaving the blocks from different functions, as required by procedure splitting. In general, basic blocks can be reordered while maintaining semantics by adjusting branch displacements, adding unconditional branches, and rewriting jump tables.

Kerninstd obtains a relocatable representation of a function by parsing its machine code into a control flow graph of basic blocks, much like what is done on kerninstd startup. No source code is required for this step. The difference is what is stored for a basic block. A relocatable basic block contains a sequence of *code objects*. There are two categories of code objects: those which execute a PC-independent sequence of non-control-flow instructions (the simple case), and those which represent a control transfer. In both cases, a code object is represented in a location-independent manner, so that a basic block can be re-emitted elsewhere in kernel memory with equivalent semantics. To achieve this, a code object's control transfer information avoids keeping any absolute addresses.

An intra-procedural conditional branch code object contains the condition to test, a delay slot instruction, and the ID (not the address) of the if-taken and if-not-taken basic blocks. Inter-procedural branch code objects store the name of the if-taken function instead of a block ID. A procedure call code object stores the name of the callee, not its address, because if the callee ends up as part of the optimized group, its address will have changed.

Code objects for jump instructions that implement jump tables (as in a C switch statement) are the most complex. They store a list of basic block IDs representing the individual destinations, so that when the function is emitted with differing ordering, the jump table data can be altered appropriately. A jump table object also contains the block ID and offset of the instructions that set a register to the fixed address of the jump table, so these instructions may be backpatched in the new version of the function, which has its own jump table data. As with other code objects, no significant kerninstd infrastructure additions were needed for parsing code objects, because the necessary structural information was already being calculated as a matter of course during control flow graph parsing (see Section 3.3).

The simplest code object is one that represents a sequence of PC-independent instructions that do not alter control flow such as adds, loads, and stores. However, control flow information may still need to be stored. If the code object ends its basic block, which occurs when a basic block falls through to another one, then the PC-independent code object stores the basic block ID of the fall-through block.

In general, for any code object that stores a fall-through block that happens to be at the start of another function, then the name of the succeeding function is stored instead of a basic block ID. When the code object is emitted in a new location, extra code is emitted to jump to the destination function in the (likely) case that the succeeding function no longer resides immediately after this basic block.

With only one exception, kerninstd is able to parse code objects for any function that it was able to parse into a control flow graph during startup. The exception is a non-loading jump table, which only occurs in two kernel functions (see Section 3.3.2). In this case, the jump semantics are not amenable to destination block reordering, because the jump's destination is a non-loaded register offset from the jump instruction. This kind of code object can be implemented in the future by emitting a more traditional jump table sequence that matches the original semantics. A traditional jump table provides the necessary level of indirection to account for the possible reordering of the destination blocks. Any function whose code objects are not successfully parsed will be excluded from the optimized group. Calls to such a function from within the group will be directed to the original version of that function.

8.4 Code Positioning: Choosing the Block Ordering

There are three steps taken in choosing the ordering of basic blocks within the optimized group. First, the set of functions to include in the optimized group is determined. Second, procedure splitting is applied to each such function, segregating the group-wide hot blocks from the cold ones. Third, basic block ordering is applied within the distinct hot and cold sections of each function. These steps determine the ordering of basic blocks within the group, which are emitted contiguously in virtual memory, implicitly performing procedure placement. A sample group layout is shown in Figure 8.1.

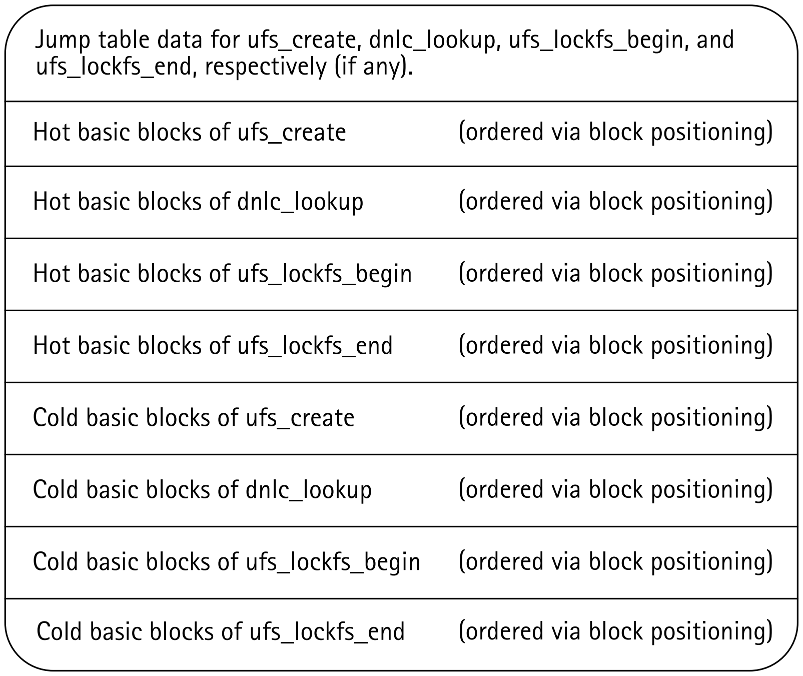

Jump table data for ufs_create, dnlc_lookup, ufs_lockfs_begin, and ufs_lockfs_end, respectively (if any).
Hot basic blocks of ufs_create
(ordered via block positioning)
Hot basic blocks of dnlc_lookup
(ordered via block positioning)
Hot basic blocks of ufs_lockfs_begin
(ordered via block positioning)
Hot basic blocks of ufs_lockfs_end
(ordered via block positioning)
Cold basic blocks of ufs_create
(ordered via block positioning)
Cold basic blocks of dnlc_lookup
(ordered via block positioning)
Cold basic blocks of ufs_lockfs_begin
(ordered via block positioning)
Cold basic blocks of ufs_lockfs_end
(ordered via block positioning)

Figure 8.1: Sample Layout of an Optimized Function Group

In this example, the function group consists of the root function, ufs_create, three of its descendants: dnlc_lookup, ufs_lockfs_begin, and ufs_lockfs_end. The chunks are not shown to scale; the cold chunks will typically be larger than the hot ones. Code in the group may call other functions within the group.

8.4.1 Which Functions to Include in the Group?

Among the functions that had their basic block execution counts measured, the optimized group will include those having at least one *hot block*. A hot basic block is one whose measured execution frequency, when the root function is on the call stack, is greater than 5% of the frequency that the root function is called. (The threshold is user-adjustable.) The technique presented in Section 7.4 is used to accurately determine how often the root function is called.

8.4.2 Procedure Splitting

Procedure splitting is the first of the code positioning optimizations that is applied. For each function in the group, a *chunk* (a contiguous layout of basic blocks) containing its hot basic blocks is segregated from the chunk containing the function's cold basic blocks, if any. The test for a hot block is the same as described in Section 8.4.1, with the exception that a function's entry block is always placed in the hot chunk, and always at the beginning of that chunk, for simplicity.

Pettis and Hansen consider any block that is executed at least once to be hot. Kperfmon can mimic this behavior by setting the user-defined hot block threshold to 0%, since an execution count of zero is always considered cold.

To help the optimization, not only are the hot and cold blocks of a single function segregated, but all of the group-wide hot blocks are segregated from the group-wide cold blocks, as shown in Figure 8.1. In other words, procedure splitting is applied on a group-wide basis.

8.4.3 Block Positioning

Block positioning uses edge execution counts to choose a layout for basic blocks that is optimized to execute straight-line code in the common case. Kperfmon applies block positioning to the hot subset of basic blocks, ordering one function's hot blocks chunk. Block positioning is also applied to the function's cold blocks chunk, although this is relatively unimportant, assuming that cold blocks are seldom executed. The remainder of this section discusses the positioning of hot basic blocks.

The algorithm that we use for block positioning is a variant of that described by Pettis and Hansen. Given a function's control flow graph, the set of hot basic blocks and their corresponding block execution counts, edge execution counts are derived, using the algorithm of Chapter 7. The hot blocks are each placed in a *chain*, a sequence of blocks that will be emitted as straight-lined code. Initially, each block is in its own chain. The graph's edges are then visited, ordered by their execution frequency (highest first). If an edge connects a block that is the tail block in a chain to a block that is the head of a different chain, then the two chains are merged. The process repeats until all edges are traversed.

The motivation behind the chains strategy is to place the more frequently taken successor block immediately after the block containing a branch. In this way, chains can eliminate some unconditional branches. For a conditional branch, placing the likeliest of the two successors immediately after the branch allows the fall-through case to be the more commonly executed path (after reversing the condition being tested by the branch instruction, if appropriate).

In general, the number of basic blocks (or instructions) in a chain gives the expected distance between taken branches, assuming that the edge counts are an accurate approximation of path counts [7]. The higher the number of instructions between taken branches, the better the I-cache utilization and the lower the mispredicted branch penalty.

If there is more than one chain among the hot blocks, then a decision needs to be made on their respective ordering. The control flow graph edges connecting blocks contained in two chains *(inter-chain edges)* are used to guide relative chain ordering. Intuitively, an inter-chain edge *(A,B)* connects two basic blocks *A* and *B* that were not laid out contiguously because a higher-weighted edge led to a presumably better choice for a successor block to *A*, or another edge led to a presumably better predecessor choice for *B*. The algorithm for chain placements starts by placing the chain containing the function’s entry block, then chooses the next chain based on the most frequently executed inter-chain edge, which is essentially a weighted depth-first search of the chain graph. Pettis and Hansen add an extra layer of logic to this step, attempting to order chains so that inter-chain edges are forward branches whenever possible. This decision matches the assumption made by early HP-RISC processors that forward branches are predicted not taken. Because modern processors dynamically update their prediction logic in hardware, we did not implement this step.

8.5 Emitting and Installing the Optimized Code

Once each function has segregated its basic blocks into hot and cold chunks (through procedure splitting) and chosen an ordering of basic blocks within those chunks (through block positioning), the optimized group code is generated, installed into the kernel, and then has its functions analyzed like any other kernel function.

8.5.1 Code Generation

Like emitting instrumentation code into a code patch, emitting functions of the outlined group is done in a relocatable form, because the group's memory location is presently unknown. An example of a relocatable element is an inter-chunk branch, whose displacement is unknown until the distance between chunks are defined. Call instructions to non-group functions specify the address of the callee; the instruction will later be patched to contain the proper PC-relative offset. Calls to group functions are specified by callee name, since the callee's address is presently unknown. Jump table data is another relocatable element. In the example of an offset jump table (Section 3.3.2), the entries depend on the displacement between the jump instruction and the destination basic block, which is represented in the relocatable code as the difference between two labels.

Branch instructions are emitted to take advantage of block positioning. If an unconditional branch's destination block now resides immediately after the block containing the branch, then the branch is optimized away, leaving only its delay slot instruction, if any. An unconditional inter-procedural branch requires extra attention. If the destination function lies within the group, then it is re-emitted as a single branch instruction. Otherwise, because we are no longer sure that the branch instruction’s displacement is sufficient, it is emitted with a long jump sequence to a constant address. (As with emitting long jumps in code patches (Section 4.2), kerninstd’s live register analysis helps, by identifying an available scratch register to use, when possible.)

A conditional branch is emitted by branching to a label that represents the beginning of the if-taken block, after which an unconditional branch (with no delay slot) is emitted to reach the former fall-through block. If the former fall-through block still resides after the branch, then the unconditional branch can be optimized away, yielding a code sequence that, aside from the possibly updated offset to reach the if-taken block, is unchanged.

The more challenging case of intra-procedural conditional branches occurs when, due to code positioning, the former if-taken block now resides immediately after the block containing the branch instruction. In this case, the condition being tested by the branch is reversed, the destination of the branch is set to the former fall-through block, and the branch’s predict bit is set to false. If the conditional branch had a delay slot instruction that was only executed when the branch was taken (this depends on its annul bit), then the reversal of the condition test means that the delay slot instruction should now be executed only if the branch is *not* taken. There is no such setting for SPARC branch instructions, so in this case, the delay slot is filled with a nop instruction, and the former delay slot instruction is emitted immediately after the nop, which makes it immediately precede the fall-through basic block, achieving the desired effect.

Inter-procedural conditional branches are the most complex relocatable element to emit. If the destination function lies within the optimized group, then kperfmon assumes that it can (still) be reached with a branch instruction, and a conditional branch to a label representing the callee is emitted. After the branch, code is emitted to reach the former fall-through block. As with intra-procedural conditional branches, there are three cases. First, if the fall-through block is intra-procedural and still resides immediately after the block containing the branch, then no extra code is emitted. Second, if the fall-through block is intra-procedural but has been moved, then an unconditional branch (with no delay slot) to a label representing that block is emitted. In the final case, if the fall-through block resides in another function (a case occurring 116 times in optimized assembly code; see Section 3.3.1), then a jump to that function is emitted: an unconditional branch instruction if that function resides in the group (and thus is guaranteed to be nearby), or a long jump sequence otherwise.

8.5.2 Installing the Optimized Group

Once the relocatable structures are emitted, they are sent to kerninstd with a request to download the code, with the specific chunk ordering shown in Figure 8.1, into a contiguous area of kernel nucleus memory. At this time, kerninstd will resolve the inter-chunk branches, call instructions, jump table data, and other such relocatable elements, much like a linker does.

The contiguous group layout has two consequences. First, it implicitly performs procedure placement. Second, it ensures that both the ±512 KB and the ±8 MB displacement provided by the two classes of SPARC branch instructions is enough to transfer control between any two chunks.

Pettis and Hansen’s method of emitting branches between hot and cold basic blocks differs from KernInst’s. In their system, any such branch is redirected to a nearby stub, which performs a long jump. Although these stubs are infrequently executed (because transfers between hot and cold blocks seldom occur), they increase total hot code size. For each branch from a hot to a cold block within a function, a stub is placed at the end of that function’s hot blocks. This layout ensures that hot blocks of multiple functions cannot be contiguously laid out for minimal I-cache footprint, because the stubs, which are effectively small but cold basic blocks, reside between the hot chunks.

8.5.3 Parsing the Group Functions

After group installation, kerninstd analyzes the group’s functions in the same way that kernel functions were analyzed on kerninstd startup: control flow graphs are constructed, register analysis is performed, and the call graph is updated. This first-class treatment of runtime-generated code allows the new functions to be instrumented (so the speedup achieved by the optimization can be measured, for example) and even re-optimized (a requirement, as mentioned in Section 8.2.3).

Procedure splitting and the consequent interleaving of functions within the optimized group required improving kerninstd’s control flow graph parsing algorithm. First, a function can now contain several disjoint “chunks”. The chunk bounds must be provided, so that branches can properly be recognized as intra-procedural or inter-procedural, and so basic blocks that fall through to another function can be identified. Fortunately, it is easy to pass the chunk bounds of each group function to the parsing code; that information follows immediately from the procedure splitting phase (Section 8.4.2).

The group's parsed functions appear in kperfmon's code resource hierarchy. Group functions are assigned to a dummy module whose name is *groupX*, where *X* is a unique group identifier. Within this module, each function's name is a colon-separated concatenation of the original function's module and function names. For example, a group function containing the function tcp_rput_data from the tcp module is assigned the function name tcp:tcp_rput_data. (Concatenation avoids any chance of a name conflict within the group functions. Two kernel functions may have the same name as long as they reside in different modules. Now that kerninstd is placing them in the same "module", unique names are required for the group functions.)

8.6 Kernel Code Positioning: Case Study with TCP running FTP

As a concrete demonstration of the efficacy of run-time kernel code positioning, this section presents initial results in optimizing the I-cache performance of a Web retrieval benchmark. We study the performance of tcp_rput_data (and its callees), the major TCP function that processes incoming network data. tcp_rput_data is called thousands of times per second in the benchmark, and has poor I-cache performance—about 36% of tcp_rput_data's execution time is due to delays caused by I-cache misses. Using our prototype implementation of code positioning, this percentage was reduced to about 28.5%. The optimization is presently limited by the inability to include within the group routines that are called via function pointers. Nevertheless, code positioning reduces the time per invocation of tcp_rput_data from 6.6 µs to 5.44 µs, a decrease in execution time of 17.6%.

8.6.1 Benchmark

We used the GNU wget tool [38] to (repeatedly) fetch the Paradyn research papers Web page [67], and all of the files that are linked to it. In all, 34 files are fetched, totaling about 28 MB of data, largely comprised of postscript, compressed postscript, and PDF files. The benchmark contained ten simultaneous connections, each running the wget program as just described.

The benchmark spends a large amount of time in TCP code. In particular, the read-side of a TCP connection is stressed, especially the routine tcp_rput_data, which processes data that has been received over an Ethernet connection and recognized as an IP packet. We chose to perform code positioning on tcp_rput_data because of its size (about 12K bytes of code across 681 basic blocks), which suggests there is room for I-cache improvement in this function.

8.6.2 The Performance of tcp_rput_data Before Code Positioning

The above benchmark completes in 36.0 seconds before code positioning is applied. To investigate where the time was being spent, we used kperfmon to measure the performance of each invocation of tcp_rput_data.

We concentrated on optimizing the time spent *per invocation* of tcp_rput_data to identify a performance improvement that will scale with the execution frequency of tcp_rput_data. The execution frequency is a function of processor and network speed, and the network load of the benchmark.

To determine whether tcp_rput_data is likely to benefit from code positioning, we measured the amount of inclusive virtual execution time that tcp_rput_data spends in I-cache misses. The result is surprisingly high; each invocation of tcp_rput_data takes about 6.6 µs, of which about 2.4 µs is idled waiting for I-cache misses. In other words, tcp_rput_data spends about 36% of its execution time in I-cache miss processing.

The basic block execution counts of tcp_rput_data and its descendants that were gathered in Section 8.2.2 give an estimate on the basic blocks that are executed while the benchmark is running. The measured execution counts are an approximation, both because code reached via an indirect function call is not measured, and because the measurement includes executions of a basic block without regard to whether the group's root function is on the call stack. These approximate block counts were used to estimate the likely I-cache layout of the subset of these blocks that are hot, based on kperfmon's default interpretation that the hot blocks are those which are executed over 5% as frequently as tcp_rput_data is called. The result is shown in Figure 8.2.

Because tcp_rput_data is called frequently, it is important that the function executes well out of the I-cache. Two conclusions about I-cache performance can be drawn from Figure 8.2. First, having greater than 2-way set associativity in the I-cache would have helped performance. The hot subset of tcp_rput_data and its descendants cannot execute without I-cache conflict misses. Second, even if the I-cache were fully associative, it may be too small to effectively run the benchmark. The bottom of Figure 8.2 estimates that 244 I-cache blocks (about 7.8K) are needed to

0	0	0	0	1	1	1	0	1	1	0	0	0	1	2	2
1	1	2	2	2	2	2	2	2	1	1	1	1	1	1	0
1	1	1	2	2	2	1	0	0	0	0	0	1	2	2	1
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
0	0	1	1	2	0	1	1	3	2	2	1	1	1	1	1
2	2	0	0	0	0	0	1	1	1	1	2	2	2	1	2
2	1	1	2	2	2	0	1	1	3	3	3	3	1	2	3
1	3	2	1	1	0	1	1	1	0	0	0	0	0	0	0
0	0	0	2	1	1	1	1	1	1	1	1	1	0	0	1
2	2	3	2	2	1	2	2	1	0	0	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	0	1	0	0	1	0	1	1
1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	2	1	1	1	1	2	2	1	2	1	2
3	3	4	4	4	3	4	4	4	3	2	1	0	2	2	1
0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0
Total # of cache blocks: 244 (47.7% of the I-Cache size)															

Figure 8.2: I-cache Layout of the Hot Blocks of tcp_rput_data and its Descendants (Pre-optimization)
Each cell represents a 32-byte I-cache block; the number within a cell is how many hot basic blocks, with distinct I-cache tags, fall on that block. This figure shows 256 cache blocks, totalling 8K. The UltraSparc I-cache is 16K 2-way set associative, so two addresses can map onto the same block in this figure without causing a conflict miss. Cells that are highlighted have more than two addresses mapping to that I-cache block, indicating a conflict.

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total # of cache blocks: 132 (25.8% of I-Cache size)															

Figure 8.3: The I-cache Layout of the Optimized tcp_rput_data Group
There are no I-cache conflicts among the hot basic blocks. Compare to Figure 8.2.

hold the hot basic blocks of tcp_rput_data and its descendants, which is about half of the total I-cache size. Because other code, particularly Ethernet and IP processing code that invokes tcp_rput_data, is also executed thousands of times per second, the total set of hot basic blocks likely exceeds the capacity of the I-cache.

8.6.3 The Performance of tcp_rput_data After Code Positioning

Code positioning was performed to improve the inclusive I-cache performance of tcp_rput_data. Figure 8.3 presents the I-cache layout of the optimized code, estimated in the same way as the data in Figure 8.2. There are no I-cache conflicts among the group's hot basic blocks, which could have fit comfortably within the confines of an 8K direct-mapped I-cache.

Figure 8.4 shows the functions in the optimized group along with the relative sizes of the hot and cold chunks. The fourth column of the figure shows how many chains were needed to cover the hot chunk. One is ideal, indicating a likelihood that all of the hot code is covered by a single path that is contiguously laid out in memory.

Code positioning reduced the benchmark's end-to-end run-time by about 7%, from 36.0 seconds to 33.6 seconds. To explain the speedup, we used kperfmon to measure the performance improvement in each invocation of tcp_rput_data. Code positioning reduced the I-cache stall time per invocation of tcp_rput_data by about 35%, the branch mispredict stall time by about 47%, and the overall virtual execution time by about 18%. In addition, the IPC (instructions per cycle) increased by about 36%. Pre- and post-optimization numbers are shown in Figure 8.5.

8.6.4 Analysis of Code Positioning Limitations

Code positioning performs well unless there are indirect function calls among the hot basic blocks of the group. This section analyzes the limitations that indirect calls placed on the optimization of tcp_rput_data (and System V streams code in general), and present measurements on the frequency of indirect function calls throughout the

Function	Jump Table Data	Hot Chunk Size (bytes)	Number of Chains in Hot Chunk (1 is best)	Cold Chunk Size (bytes)
group1/tcp:tcp_rput_data	56	1980	10	11152
group1/unix:mutex_enter	0	44	1	0
group1/unix:putnext	0	160	1	132
group1/unix:lock_set_spl_spin	0	32	1	276
group1/genunix:canputnext	0	60	1	96
group1/genunix:strwakeq	0	108	1	296
group1/genunix:isuioq	0	40	1	36
group1/ip:mi_timer	0	156	1	168
group1/ip:ip_cksum	0	200	1	840
group1/tcp:tcp_ack_mp	0	248	1	444
group1/genunix:pollwakeup	0	156	1	152
group1/genunix:timeout	0	40	1	0
group1/genunix:.div	0	28	1	0
group1/unix:ip_ocsum	0	372	4	80
group1/genunix:allocb	0	132	1	44
group1/unix:mutex_tryenter	0	24	1	20
group1/genunix:cv_signal	0	36	1	104
group1/genunix:pollnotify	0	64	1	0
group1/genunix:timeout_common	0	204	1	52
group1/genunix:kmem_cache_alloc	0	112	1	700
group1/unix:disp_lock_enter	0	28	1	12
group1/unix:disp_lock_exit	0	36	1	20
Totals	56	4260	34	14624

Figure 8.4: The Size of Optimized Functions in the tcp_rput_data Group

The group contains a new version of tcp_rput_data, and the hot subset of its statically identifiable call graph descendants, with code positioning applied. This figure shows the effects of procedure splitting, in which all hot chunks are moved away from all cold chunks. The fourth column contains the number of chains in the hot chunk. One chain covering the entire hot chunk is ideal, indicating a likelihood that a single hot path, laid out contiguously, covers all of a function's hot blocks.

Measurement	Original	Optimized	Change
Total virtual execution time per invocation	6.6 µs	5.44 µs	-1.16 µs (-17.6%)
I-cache stall time per invocation	2.4 µs	1.55 µs	-0.85 µs (-35.4%)
Branch mispredict stall time per invocation	0.38 µs	0.20 µs	-0.18 µs (-47.4%)
IPC (instructions per cycle)	0.28	0.38	+0.10 (+35.7%)

Figure 8.5: Measured Performance Improvements in tcp_rput_data After Code Positioning

The performance of tcp_rput_data has improved by 17.6%, mostly due to fewer I-cache stalls and fewer branch mispredict stalls.

kernel, to give a quantitative idea of how the present inability to optimize across indirect function calls constrains code positioning. We then discuss a unique technical limitation that prevents the dynamic optimization of the Solaris function mutex_exit.

The System V streams code has enough indirect calls to effectively limit what can presently be optimized to a single streams module (TCP, IP, or Ethernet). Among the measured hot code of tcp_rput_data and its descendants, there are two frequently-executed indirect function calls. Both calls are made from putnext, a stub routine that forwards data to the next upstream queue by indirectly calling the next module's stream "put" procedure. This call is made when TCP has completed its data processing (verifying check sums and stripping off the TCP header from the data block), and is ready to forward the processed data upstream. Because callees reached by hot indirect function calls cannot currently be optimized, we miss the opportunity to include the remaining upstream processing code in the group. At the other end of the System V stream, by using TCP's data processing function as the root of the optimized group, we missed the opportunity to include downstream data processing code performed by the Ethernet and IP protocol processing.

To give a quantitative indication of how the inability to optimize indirect calls limits code positioning, Figure 8.6 contains the number of indirect calls made by each kernel module. The figure shows that direct calls (or direct inter-procedural branches) greatly outnumber indirect calls, and some modules make no indirect calls. However, because indirect calls exist in the unix and genunix modules, which contain many utility routines that are invoked throughout the kernel, any large group will likely contain at least one indirect function call. For example, although the figure shows that TCP makes no indirect function calls, we have seen that the unix module's putnext function, which performs an indirect call, is pulled into the group.

Module	#Functions Parsed	Average # of direct calls (or inter-procedural branches) by a function in this module	Average # of indirect calls made by a function in this module
genunix	2589	5.3	0.2
afs (afs syscall interface)	922	8.2	0.2
unix	1758	2.8	0.1
ufs (filesystem for ufs)	337	8.4	0.7
nfs (NFS syscall, client, and common)	479	7.3	0.2
ip (IP Streams module)	373	6.4	0.1
md (Meta disk base module)	390	6.3	0.6
tcp (TCP Streams module)	159	9.1	0.0
procfs (filesystem for proc)	174	8.8	0.4
sd (SCSI Disk Driver 1.308)	115	14.4	0.0
rpcmod (RPC syscall)	209	5.7	0.5
sockfs (filesystem for sockfs)	149	8.8	0.1
pci (PCI Bus nexus driver)	127	5.7	1.0
hme (FEPS Ethernet Driver v1.121)	97	11.0	0.1
se (Siemens SAB 82532 ESCC2 1.93)	69	14.3	0.2
fd (Floppy Driver v1.102)	54	15.0	0.0
zs (Z8530 serial driver V4.120)	48	14.0	0.3
uata (ATA AT-bus attachment disk controller Driver)	127	5.7	0.1
krtld	127	3.7	0.8
rpcsec (kernel RPC security module.)	122	5.1	0.3
ufs_log (Logging UFS Module)	131	4.9	0.0
xfb (xfb driver 1.2 Sep 7 1999 11:46:39)	99	3.6	0.8
audiocs (CS4231 audio driver)	83	7.5	0.3
dad (DAD Disk Driver 1.16)	56	11.1	0.0
tmpfs (filesystem for tmpfs)	66	7.1	0.1
ldterm (terminal line discipline)	45	9.6	0.0
afb (afb driver v1.36 Sep 7 1999 11:47:45)	60	5.0	0.8
scsi (SCSI Bus Utility Routines)	78	4.9	0.4
tl (TPI Local Transport Driver - tl)	57	7.7	0.0
specfs (filesystem for specfs)	48	6.5	0.4
arp (ARP Streams module)	67	5.6	0.0
SUNW,UltraSPARC-IIi	60	3.2	0.0
vol (Volume Management Driver, 1.85)	23	21.0	0.0
doorfs (doors)	52	7.9	0.2
su (su driver 1.24)	31	14.5	0.0
udp (UDP Streams module)	43	6.6	0.1
timod (transport interface str mod)	36	10.6	0.0
kerninst (kerninst driver v0.4.1)	116	2.9	0.0
fifofs (filesystem for fifo)	40	6.7	0.2
kb (streams module for keyboard)	36	5.8	0.0
tnf (kernel probes driver 1.47)	55	3.4	0.3
pm (power manager driver v1.65)	29	11.4	0.1
TS (time sharing sched class)	36	4.6	0.1
devinfo (DEVINFO Driver 1.24)	35	7.4	0.1
ipdcm (IP/Dialup v1.9)	52	4.8	0.1
ttcompat (alt ioctl calls)	14	5.4	0.0
diaudio (Generic Audio)	28	5.1	0.8
elfexec (exec module for elf)	12	12.5	0.2
shmsys (System V shared memory)	19	10.0	0.0
ptc (tty pseudo driver control ‘ptc’)	16	11.0	0.0
tlimod (KTLI misc module)	21	6.9	0.0

Figure 8.6: Average Number of Direct and Indirect Calls Made by Kernel Functions

Module	#Functions Parsed	Average # of direct calls (or inter-procedural branches) by a function in this module	Average # of indirect calls made by a function in this module
winlock (Winlock Driver v1.39)	39	3.8	0.0
hwc (streams module for hardware cursor support)	11	12.9	0.0
ms (streams module for mouse)	17	6.5	0.0
ptem (pty hardware emulator)	13	12.4	0.0
simba (SIMBA PCI to PCI bridge nexus driver)	18	8.7	0.0
seg_drv (Segment Device Driver v1.1)	25	3.1	0.5
sad (Streams Administrative driver'sad')	23	4.7	0.0
namefs (filesystem for namefs)	32	3.4	0.3
lockstat (Lock Statistics)	27	2.4	0.0
ptsl (tty pseudo driver slave 'ptsl')	14	9.9	0.0
rootnex (sun4u root nexus)	19	3.9	0.0
dada (ATA Bus Utility Routines)	30	2.8	0.3
dada_ata (ATA Bus Utility Routines)	30	2.8	0.3
md5 (MD5 Message-Digest Algorithm)	8	1.6	0.0
sysmsg (System message redirection (fanout) driver)	15	4.5	0.5
mm (memory driver)	13	4.8	0.0
wc (Workstation multiplexer Driver 'wc')	18	5.1	0.0
ebus (ebus nexus driver)	13	6.7	0.0
ptm (Master streams driver 'ptm')	12	10.2	0.0
pts (Slave Stream Pseudo Terminal driver 'pts')	12	10.0	0.0
RT (realtime scheduling class)	24	2.4	0.0
iwscn (Workstation Redirection driver 'iwscn')	19	4.5	0.3
fdfs (filesystem for fd)	17	2.4	0.0
eide (PC87415 Nexus driver v2.0)	19	4.7	0.1
conskbd (Console kbd Multiplexer driver 'conskbd')	14	4.1	0.0
todmostek (tod module for Mostek M48T59)	10	2.7	0.0
log (streams log driver)	9	5.4	0.0
sy (Indirect driver for tty 'sy')	12	5.5	0.0
consms (Mouse Driver for Sun 'consms')	14	3.8	0.0
kstat (kernel statistics driver)	12	4.2	0.3
pckt (pckt module)	11	4.5	0.0
ksyms (kernel symbols driver)	11	4.4	0.0
inst_sync (instance binding syscall)	11	3.5	0.4
power (power driver v1.4)	11	4.5	0.0
cn (Console redirection driver)	13	1.9	0.2
sysacct (acct(2) syscall)	6	4.5	0.8
clone (Clone Pseudodriver 'clone')	7	3.4	0.3
intpexec (exec mod for interp)	5	3.6	0.0
pseudo (nexus driver for 'pseudo')	10	2.8	0.0
ipc (common ipc code)	5	1.2	0.0
pipe (pipe(2) syscall)	4	6.2	0.2
connld (Streams-based pipes)	6	1.7	0.0
options (options driver)	7	0.6	0.0
redirmod (redirection module)	6	1.5	0.0
TS_DPTBL (Time sharing dispatch table)	5	0.4	0.0
IA (interactive scheduling class)	3	0.7	0.0
RT_DPTBL (realtime dispatch table)	3	0.7	0.0
platmod	5	0.0	0.0
Kernel-wide:	10637	6.0	0.2

Figure 8.6: Average Number of Direct and Indirect Calls Made by Kernel Functions

8.7 Future Work

Future work that can improve the effectiveness of runtime kernel code positioning includes better handling of function pointers, automated selection of the optimized group’s root function, block ordering across procedure boundaries, and additional inline expansions.

Calls via function pointers are never included in an optimized group, because they are not recognized in the call graph traversal. This can limit the effectiveness of the code positioning optimization. As mentioned in Section 3.3.2, such calls occur about 2,400 times in the kernel. It would be useful to employ an instrumentation technique, similar to what is done in Paradyn [14], to dynamically update the contents of the call graph when indirect calls are made.

Once indirect callees are measured, they can be found to be hot enough to deserve inclusion in the group. Ensuring that such optimized callees are actually invoked requires informing the indirect call instruction of the callee’s new location. The desired effect can be achieved by emitting a few instructions immediately before the indirect call. These instructions change the value of the register containing the callee’s address to the address of the group’s version of the callee, if the previous value of that register was the original location of the callee. (The test is needed because there can be many callees of any one indirect call instruction.)

Another candidate for future work is to remove user involvement in the first code positioning step (choosing the group’s root function), thus allowing all steps to be performed automatically. Paradyn’s Performance Consultant [14, 42] has already shown that bottlenecks can be automatically located for non-threaded user programs, through a call graph traversal. It should be possible to develop a multi-thread aware Performance Consultant, and then adapt it to kperfmon, which uses the same underlying measurement concepts of metrics and resources. Completely automating the optimization allows for a truly evolving kernel, where, in the background, kperfmon periodically finds and attempts to optimize I-cache bottlenecks.

Although it is not explicitly stated in Pettis and Hansen's paper, basic block ordering uses edge counts merely as an approximation of path counts in choosing chains. Toward that end, it would be useful for kperfmon to be able to collect path profiles of group functions.

Other than emitting all hot sections before any cold sections, the relative placement of functions within the group is arbitrary. With future work, this situation can be improved by performing basic block positioning across procedure call bounds, allowing chains to contain basic blocks from different functions. The benefit would be execution of even longer sequences of straight-lined code in the common case. This change would not necessarily blur the bounds between group functions or otherwise make it impossible to parse their control flow graphs. The only major complexity would be functions whose code is spread out in more than the three chunks that are presently supported (jump table data, hot basic blocks, and cold basic blocks). Note that this implementation would not by itself cause any increase in the optimized group size.

A further optimization is to expand several hot paths, duplicating any basic block(s) that reside in multiple hot paths. Path expansion can increase the length of straight-lined code execution. Pettis and Hansen essentially try to order code for one such hot path (using edge counts as an approximation of path counts), though like kperfmon, it currently does not allow a path to extend beyond a function call boundary. Removing this limitation enables inter-procedural paths. Duplicating hot basic blocks (in effect, inlining just the hot portions of callees) will place several optimized paths in the group. This optimization is performed by the Dynamo user-level run-time optimization system [5], which has found path expansion generally beneficial, though it can backfire on some occasions due to code explosion when the code size exceeds the size of the I-cache. Dynamo runs on an HP-PA processor, which has the luxury of an unusually large (1 MB) dedicated I-cache. It remains to be seen whether a different processor would be less tolerant of code explosion. For example, the UltraSPARC-I and II processors have only a 16 KB I-cache.

Because non-root group functions are always invoked while the root function is on the call stack, certain invariants may hold that enable improved optimizations. For example, a variable may always be constant, allowing constant propagation and dead code elimination. Other optimizations include inlining, specialization, path expansion, and super-blocks. Such optimizations are presently unimplemented, demonstrating the need for a general-purpose back-end machine code optimizer. Unfortunately, such optimizations are typically performed by compilers, on intermediate code, not by assemblers. However, the full suite of optimizations implemented by Dynamo shows that (with much effort) many optimizations can be achieved by operating directly on machine code.

8.8 Conclusion and Research Contributions

This chapter has introduced the notion of evolving kernels, which are able to change their code in response to runtime circumstance. As a proof of concept, we have implemented one kind of evolving kernel algorithm, a run-time version of Pettis and Hansen’s code positioning optimizations. Our implementation of code positioning is the first on-line kernel version of this optimization. Aside from adaptive algorithms and tunable variables built into a kernel’s source code (such as adaptive mutex locks in Solaris), this implementation is also the first on-line evolving algorithm of a kernel. The implementation provides evidence that an unmodified commodity operating system kernel can be made into an evolving one; there is no need to limit research into evolving systems to custom kernels.

Chapter 9

Conclusion

Run-time changing of kernel code has many uses, including performance profiling, kernel debugging, tracing, testing via code coverage, inserting security auditing checks at runtime, dynamic optimizations, process-specific resource management extensibility, transparent data modification, and adapting to security attacks. This thesis has investigated mechanisms for fine-grained dynamic kernel instrumentation, allowing the code of a commodity operating system to change at runtime. We have shown its application to performance profiling and dynamic optimizations. This chapter summarizes the research contributions of this dissertation, and discusses areas of future work.

9.1 Summary and Research Contributions

We have designed and implemented a technology, fine-grained dynamic instrumentation of a commodity operating system kernel, and investigated two of its applications: kernel performance measurement and evolving kernels.

The primary question raised by this line of research,

> Is fine-grained dynamic instrumentation of a completely unmodified, already-running modern commodity operating system kernel possible?

has been answered in the affirmative. The following technical challenges were overcome, which serve as further contributions:

- **Bootstrapping.** It is possible to attach to and instrument a running kernel.
- **Structural Analysis.** The kernel's control flow graphs, call graph, and live register analysis information were calculated without kernel source code. The control flow graphs are important because they identify allowable instrumentation points, and help to understand complex instruction sequences such as tail calls. Live register usage allows instrumentation code to often avoid spilling registers to the stack.
- **Hazard-Free Splicing.** Single-instruction splicing avoids a race condition where some thread(s) may be executing near or within the instrumentation point. The kernel cannot be paused to check for this condition, so it was necessary to avoid, rather than detect, the hazard.
- **The Reach Problem.** Single-instruction splicing is difficult because no single branch or jump instruction can always reach a code patch. Two novel solutions were designed: springboards and in-nucleus code patch allocation.
- **Instruction Relocation.** Some instructions cannot be trivially relocated to the code patch: control transfer instructions having delay slots, instructions that are pc-dependent, tail calls, and otherwise trivial-to-relocate instructions that reside in a delay slot. A relocation mechanism was designed for all but one intractable scenario: delay slot instructions that are also the destination of another control flow instruction.

  An unexpected result was that it was important (and possible) to splice at a finer granularity than instruction level in two cases: conditional branches (both the if-taken and fall-through points) and tail calls (the point where, logically, the call has returned, but the caller itself has not yet returned).

Kperfmon demonstrates the utility of fine-grained dynamic kernel instrumentation, but more importantly, it is a powerful performance tool in its own right. The major research contributions of kperfmon are:

- **Measure Almost Anywhere.** Kperfmon can measure almost any kernel function or basic block.
- **Dynamic.** Unlike a static instrumentation system, dynamic instrumentation allows the user to instrument only what is of interest, when it is of interest.
- **Extensible Metrics.** Kperfmon can create new interval counter metrics out of any monotonically increasing event counter.
- **Classes of Metrics.** Wall time metrics and inclusive metrics are difficult to achieve with sampling-based profilers.
- **Virtualization.** Any wall time metric can be virtualized via dynamic instrumentation of the kernel’s context switch code. Virtual time metrics are normally easy to achieve with sampling-based profiler (though not combined with the inclusive property) but are difficult to achieve via instrumentation.

The final component of this dissertation is a framework for evolving kernels. An evolving kernel (1) performs measurements to identify a problem, (2) generates improved code to insert into the kernel, (3) installs this code into a running kernel, and (4) repeats the process. The first step is well-suited to kperfmon. The third step is performed through code replacement, an instrumentation primitive that complements code splicing. The following contributions have been made in the area of evolving kernels:

- **General Framework for Dynamically Changing a Commodity Kernel’s Code.** Code replacement allows almost any kernel function to have its implementation changed at run-time.

- **Prototype Dynamic Kernel Optimizer.** Our implementation of code positioning is the first run-time optimization of an unmodified modern commodity operating system kernel that does not leverage any built-in kernel tuning facilities.
- **An Unmodified Commodity Kernel Can Be Made Into an Evolving One.** Research into runtime kernel code modification need not be limited to custom kernels or modified versions of otherwise commodity kernels.

9.2 Future Work

9.2.1 Dynamic Kernel Instrumentation

When splicing at a non-nucleus instrumentation point, there is presently no guarantee that a code patch can be allocated nearby. Instrumentation may have to rely on outside-of-nucleus springboard space, which can be scarce; unlike the nucleus, where modules are allocated close to each other and can reach each other's _init and _fini springboard space, no such guarantee applies to outside-of-nucleus modules. A solution resides in directed virtual memory allocation, allocating kernel virtual memory pages from the kernel's internal structures, with care to pick a page within the reach of a single branch instructions.

Code removal can be unsafe, possibly freeing code patches or springboards while while some kernel thread(s) are still executing within them. This issue, and a potential solution, are discussed in Section 4.4.

A KernInst port to the x86 architecture would show that fine-grained splicing of an unmodified kernel can be made ubiquitous. Creating control flow graphs and performing a live register analysis is no more difficult on variable-length instruction architectures than on RISC[1]; the major x86 challenge is achieving single-instruction splicing, given variable-length instructions, as discussed in Section 4.6.1.

It would be convenient to accept instrumentation code directly out of ELF relocatable files. However, because kerninstd may bracket instrumentation code with SPARC save and restore instructions, clients currently generate two versions of instrumentation code, one using scratch registers as determined by a live register analysis, and other assuming the code will be bracketed with save and restore. A better form for instrumentation code would use virtual registers, which are assigned to actual registers by kerninstd. Unfortunately, there is no ELF standard for such a format.

9.2.2 Kernel Performance Profiling

Kperfmon's instrumentation code requires numerous changes when run on a multiprocessor, as detailed in Sections 5.4.4 and 5.5. Fortunately, the changes are all straightforward to implement.

A fruitful area of future work would combine the kernel instrumentation features of KernInst with the user-level instrumentation features of Paradyn [43, 44], allowing bottleneck searches to continue through system calls.

Paradyn's Performance Consultant [14, 42] automatically searches for bottlenecks. Nowhere would the seamless transition between user-level and kernel dynamic instrumentation be more immediately useful than in a logical extension of the Performance Consultant to kernel code.

1. Vic Zandy has completed a prototype implementation of kerninstd's structural analysis steps on x86/Linux.

Kperfmon would be more powerful with high-level metrics, such as “number of TCP re-transmissions”, “number of ethernet collisions”, “elapsed time spent performing disk seeks”, “average number of threads blocked of a mutex lock”.

Measurements predicated on the caller’s identify would be useful, allowing queries such as “How many threads are presently blocked on a mutex lock while running in procedure P?”. More generally, such predicates could be extended to arbitrary desired chains of basic blocks, to provide kernel path profiling.

As discussed in Section 5.6, it would be useful to extend kperfmon’s resource hierarchy to include data objects, such as specific mutex objects, files, TCP connections, IP ports, and physical disks. This extension would provide end users with a higher-level view of performance data that is easier to interpret.

Kernel tools that access symbols via /dev/ksyms would benefit if kerninstd updated the kernel’s internal information upon code replacement. This change would allow replacement code to be recognized as first-class in applications that do not use kerninstd to obtain their structural analysis information.

9.2.3 Dynamic Kernel Optimization and Evolving Kernels

The implementation of the code positioning optimization within KernInst was time-consuming. It would have been useful to leverage an existing tool-kit that provided a number of runtime kernel optimizations (operating directly on binary code), to avoid re-implementing optimizations that are already implemented in compilers.

Among the unanswered questions in the evolving kernel framework are whether all traditional compiler optimizations are feasible at runtime, especially without access to the kernel’s source code.

And finally, a vision for KernInst is to enable fully evolving systems, where all operations such as kernel compilation, installation, linking, debugging and applying patches take place at run-time. It remains to be seen whether this is feasible and practical. Among the questions to be raised are whether such a system be easily debugged.

9.3 Final Thoughts

In this dissertation, we have tried to make two key points:

- That fine-grained dynamic kernel instrumentation, if possible, would have many applications.
- That fine-grained dynamic instrumentation is possible on a completely unmodified, already-running commodity operating system kernel.

This research has been performed at runtime, on a production commodity kernel. We believe that this was important to show the feasibility of kernel code modification.

We hope that this thesis research is but the tip of an iceberg that eventually culminates in fully evolving kernels, where an operating system is never taken off-line—whether for performance measurement, to apply patches, to debug, or even when making major developmental changes. By its very nature, an operating system is ideally never taken off-line or even momentarily paused (especially when running mission-critical server software), and we believe that this ideal can some day be made a reality.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. **Compilers: Principles, Techniques, and Tools**. Addison-Wesley, Reading, MA, 1986.

[2] J. Almeida and P. Cao. *Wisconsin Proxy Benchmark 1.0.* Available at `http://www.cs.wisc.edu/~cao/wpb1.0.html`.

[3] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T. A. Leung, R.L. Sites, M.T. Vandervoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *16th ACM Symposium on Operating System Principles (SOSP)*, Saint-Malo, France, October 1997.

[4] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, Effective Dynamic Compilation. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, May 1996.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, BC, June 2000.

[6] T. Ball and J.R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems* **16**(4), July 1994.

[7] T. Ball, P. Mataga, and M. Sagiv. Edge Profiling versus Path Profiling: The Showdown. *25th Annual ACM Symposium on Principles of Programming Languages (POPL)*. San Diego, CA, January 1998.

[8] G. Banga and J.C. Mogul. Scalable Kernel Performance for Internet Servers Under Realistic Loads. *1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[9] R. Barnes, R. Chaiken, and D. Gillies. Feedback-Directed Data Cache Optimizations for the x86. *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*, Haifa, Israel, November 1999.

[10] B.N. Bershad, C. Chambers, S.J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E.G. Sirer. SPIN – An Extensible Microkernel for Application-

Specific Operating System Services. *6th ACM SIGOPS European Workshop*, Dagstuhl Castle, Germany, September 1994.

[11] B.N. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiucynski, D. Becker, C. Chambers, and S.N. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. *15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[12] M. Bishop. Profile Under UNIX by Patching. *Software—Practice & Experience* **17**, 10 (October 1987).

[13] B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. Available at `http://www.cs.umd.edu/projects/dyninstAPI`.

[14] H. Cain, B.P. Miller, and B.J.N. Wylie. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. E*uropean Conference on Parallel Computing (Euro-Par)*, Munich, Germany, August 2000.

[15] J.P. Casmira, D.P. Hunter, and D.R. Kaeli. Tracing and Characterization of Windows NT-based System Workloads. *Compaq Digital Technical Journal*, 10(1), January 1999. Available at `http://www.digital.com/DTJT00/index.html`.

[16] W. Chen, S. Lerner, R. Chaiken, D.M. Gillies. Mojo: A Dynamic Optimization System. *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*. Monterey, California, December 2000.

[17] K. Chow and Y. Wu. Feedback-Directed Selection and Characterization of Compiler Optimizations. *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*, Haifa, Israel, November 1999.

[18] A. Cockroft and R. Pettit. **Sun Performance And Tuning: Java and the Internet**. Sun Microsystems Press (Prentice-Hall), Mountain View, CA, 1998.

[19] R. Cohn and G. Lowney. Feedback Directed Optimization in Compaq's Compilation Tools for Alpha. *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*, Haifa, Israel, November 1999.

[20] A. Cox and R. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. *20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[21] D.R. Cheriton and K.J. Duda. A Caching Model of Operating System Kernel Functionality. *1st USENIX Symposium on Operating System Design and Implementation (OSDI)*, Monterey, CA, November 1994.

[22] Compaq Corporation. *Using DCPI/ProfileMe for Performance Tuning on Alpha 21264a (EV67)*. Presentation at Compaq, Nov 1999. Available at `http://www.tru64unix.compaq.com/dcpi/documentation/PM_Tutorial[1].ppt`.

[23] C. Cowan, T. Audrey, C. Krasic, C. Pu, and J. Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. *International Conference on Configurable Distributed Systems (ICCDS)*, Annapolis, MD, May 1996.

[24] C. Cowan, A. Black, C. Krasic, C. Pu, J. Walpole, C. Consel, and E. Volanschi. Specialization Classes: An Object Framework for Specialization. *5th International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Seattle, WA, October 1996.

[25] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos. *ProfileMe*: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. *30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-30)*, Research Park Triangle, NC, December 1997.

[26] D. Deaver, R. Gorton, N. Rubin. Wiggins/Redstone: An On-line Program Specializer. *Hot Chips 11*. Stanford, CA, August 1999. Presentation available at `ftp://www.hotchips.org/pub/hotc7to11cd/hc99/hc11_pdf/hc99.s6.2.Deaver.pdf`.

[27] L.P. Deutsch and B.W. Lampson. *DDT Time Sharing Debugging System Reference Manual*. Document #30.40.10 (rev.), University of California, Berkeley, May 1965.

[28] P. Deutsch and C.A. Grant. A Flexible Measurement Tool for Software Systems. *Proceedings of Information Processing (IFIP)*. Ljubljana, Yugoslavia, 1971.

[29] P. Deutsch. Personal e-mail communication, February 1999.

[30] Digital Equipment Corporation. *DDT – Dynamic Debugging Technique*. Technical Report DEC-10-UDDTA-A-D, Maynard, MA, 1974.

[31] Digital Equipment Corporation. **Alpha 21164 Microprocessor Hardware Reference Manual**. Maynard, MA, 1995.

[32] P. Diniz and M. Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997.

[33] D. Engler. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, May 1996.

[34] D.R. Engler and M.F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing Using Dynamic Code Generation. *ACM Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM)*, Stanford CA, August 1996.

[35] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole, Jr. The Operating System Kernel as a Secure Programmable Machine. *6th ACM SIGOPS European Workshop*, Dagstuhl Castle, Germany, September 1994.

[36] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[37] M. Fiuczynski, B. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. *1996 Winter USENIX Conference*, San Diego, CA, January 1996.

[38] Free Software Foundation. *GNU wget: The Non-Interactive Downloading Utility*. Available at `http://www.gnu.org/manual/wget-1.5.3/html_mono/wget.html`.

[39] D. Ghormley, S. Rodrigues, D. Petrou, and T. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. *1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[40] S.L. Graham, P.B. Kessler, and M.K. McKusick. gprof: a Call Graph Execution Profiler. *SIGPLAN 1982 Symposium on Compiler Construction*, Boston, MA, June 1982.

[41] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Joust: A Platform for Liquid Software. *IEEE Network* (Special Edition on Active and Programmable Networks), July 1998.

[42] J.K. Hollingsworth and B.P. Miller. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. *Seventh ACM International Conference on Supercomputing (ICS)*, Tokyo, July 1993.

[43] J.K. Hollingsworth, B.P. Miller, and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools, *Scalable High Performance Computing Conference (SHPCC),* Knoxville, TN, May 1994.

[44] J.K. Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naim, Z. Xu and L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. *International Conference on Parallel Architectures and Compilation Techniques (PACT),* San Francisco, CA, November 1997.

[45] Intel Corporation. **Pentium(R) Pro Processor Developer's Manual**. McGraw-Hill, June 1997.

[46] Intel Corporation. VTune Performance Analyzer 4.5. `http://developer.intel.com/vtune/analyzer/index.htm`.

[47] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application Performance and Flexibility on Exokernel Systems. *16th ACM Symposium on Operating System Principles (SOSP),* Saint-Malo, France, October 1997.

[48] S.R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Summer 1986 USENIX Technical Conference,* Atlanta, GA, June 1986.

[49] D.E. Knuth and F.R. Stevenson. Optimal Measurement Points for Program Frequency Counts. *BIT* **13** (1973).

[50] D. E. Knuth. Structured Programming with GOTO Statements. *ACM Computing Surveys* **6**, 4 (1974).

[51] G.H. Kuenning. Precise Interactive Measurement of Operating Systems Kernels. *Software—Practice & Experience* **25**, 1 (January 1995).

[52] B.W. Lampson. Interactive Machine Language Programming. *AFIPS Joint Computer Conference* **27**, 1 (Fall 1965).

[53] J.R. Larus and T. Ball. Rewriting Executable Files to Measure Program Behavior. *Software—Practice & Experience* **24**, 1 (February 1994).

[54] J.R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI),* La Jolla, CA, June 1995.

[55] P. Lee and M. Leone. Optimizing ML with Run-Time Code Generation. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, May 1996.

[56] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel. *12th ACM Symposium on Operating System Principles (SOSP)*, Litchfield Park, AZ, December 1989.

[57] J. Mauro. *Kernel Synchronization Primitives*. In SunWorld Online, September 1999. Available at `http://sunsite.uakom.sk/sunworldonline/swol-09-1999/swol-09-insidesolaris.html`.

[58] J. Mauro and R. McDougall. **Solaris Internals: Core Kernel Components**. Sun Microsystems Press (Prentice-Hall), Palo Alto, CA, 2001.

[59] A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, T.A. Proebsting. Scout: A Communications-Oriented Operating System. *5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[60] R.J. Moore. Dynamic Probes and Generised Kernel Hooks Interface for Linux. *USENIX Association Proceedings of the 4th Annual Linux Showcase & Conference*, Atlanta, October 2000.

[61] D. Mosberger, L.L. Peterson, P.G. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. *ACM Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM)*, Stanford, CA, August 1996.

[62] D. Mosberger and L.L. Peterson. Making Paths Explicit in the Scout Operating System. *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, October 1996.

[63] National Laboratory for Applied Network Research. *Squid Web Proxy Server*. Available at `http://squid.nlanr.net/squid`.

[64] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. *Second USENIX Symposium on Operating System Design and Implementation (OSDI)*. Seattle, WA, October 1996.

[65] G. Necula. Proof-Carrying Code. *24th Annual ACM Symposium on Principles of Programming Languages (POPL)*, Paris, France, January 1997.

[66] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI),* Montreal, Canada, June 1998.

[67] Paradyn Project Technical Papers Web Site. `http://www.cs.wisc.edu/paradyn/papers/index.html`.

[68] P. Pardyak, B. Bershad. Dynamic Binding for an Extensible System. *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, October 1996.

[69] K. Pettis and R.C. Hansen. Profile Guided Code Positioning. *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI),* White Plains, NY, June 1990.

[70] C. Ponder and R. Fateman. Inaccuracies in Program Profilers. *Software—Practice & Experience* **18**, 5 (May 1988).

[71] R.L. Probert. Optimal Insertion of Software Probes in Well-Delimited Programs. *IEEE Transactions on Software Engineering* **8**, 1 (January 1982).

[72] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis Kernel. *Computing Systems* **1**, 1 (Winter 1988).

[73] C. Pu and J. Walpole. A Study of Dynamic Optimization Techniques: Lessons and Directions in Kernel Design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science & Technology, 1993.

[74] C. Pu, T. Audrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. *15th ACM Symposium on Operating System Principles (SOSP),* Copper Mountain, CO, December 1995.

[75] C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. Microlanguages for Operating System Specialization. *SIGPLAN Workshop on Domain-Specific Languages (DSL)*, Paris, France, January 1997.

[76] Rational Software Web Page. *Purify: Fast Detection of Memory Leaks and Access Errors*. Available at `http://www.rational.com/products/whitepapers/319.jsp`.

[77] Rational Software Web Page. *Quantify Web site*. Available at `http://www.rational.com/products/quantify`.

[78] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. *USENIX Windows NT Workshop*, Seattle, WA, August 1997.

[79] D.H. Rosenthal. Evolving the Vnode Interface. *1990 Summer USENIX Technical Conference,* Anaheim, CA, June 1990.

[80] R. Saavedra and D. Park. Improving the Effectiveness of Software Prefetching With Adaptive Execution. *1996 Conference on Parallel Algorithms and Compilation Techniques (PACT),* Boston, MA, October 1996.

[81] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI),* Seattle, WA, October 1996.

[82] M.I. Seltzer and C. Small. Self-monitoring and Self-adapting Operating Systems. *6th Workshop on Hot Topics in Operating Systems (HotOS-VI),* Rio Rico, AZ, March 1997.

[83] M.I. Seltzer, C. Small, M.D. Smith. Symbiotic Systems Software. *First Annual Workshop on Compiler Support for Systems Software (WCSSS)*, Tucson, AZ, February 1996.

[84] E.G. Sirer, M. Fiuczynski, P. Pardyak, B. Bershad. Safe Dynamic Linking in an Extensible Operating System. *First Annual Workshop on Compiler Support for Systems Software (WCSSS),* Tucson, AZ, February 1996.

[85] R. Sites. Personal communication, October 1998.

[86] Silicon Graphics. *R10000 Microprocessor User's Manual-Version 2.0.* Available at `http://www.sgi.com/processors/r10k/manual.html`.

[87] C. Small. A Tool for Constructing Safe Extensible C++ Systems. *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, NM, April 1998.

[88] M.D. Smith. Tracing With Pixie. Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, Stanford, CA, November 1991.

[89] M.D. Smith. Overcoming the Challenges to Feedback-Directed Optimization. *ACM SIGPLAN Workshop on dynamic and Adaptive Compilation and Optimization (Dynamo '00),* Boston, MA, January, 2000.

[90] Solaris ABI tools: appcert. Available at `http://www.sun.com/developers/tools/appcert`.

[91] Solaris manual page. kstat (3k).

[92] Solaris manual page. lockstat (1m).

[93] Solaris manual page. prof(1).

[94] Solaris manual page. tracing (3x).

[95] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI),* Orlando, FL, June 1994.

[96] Sun Microsystems. *UltraSPARC User's Manual*. Microelectronics division. Available at `www.sun.com/microelectronics/manuals`.

[97] Sun Microsystems. *UltraSPARC-IIi User's Manual*. Microelectronics division. Available at `www.sun.com/microelectronics/manuals`.

[98] A. Tamches and B.P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[99] A. Tamches and B.P. Miller. Using Dynamic Kernel Instrumentation for Kernel and Application Tuning. *The International Journal of High Performance Computing Applications* **13**, 3 (Fall 1999).

[100] United States Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria,* TCSEC DOD 5200.28-STD, December 1985.

[101] U. Vahalia. **UNIX Internals: The New Frontiers**. Prentice Hall, Upper Saddle River, NJ, 1996.

[102] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. *14th ACM Symposium on Operating System Principles (SOSP)*, Asheville, NC, December 1993.

[103] D.L. Weaver and T. Germond, eds. **The SPARC Architecture Manual**. SPARC International (Prentice Hall), Menlo Park, CA, 1994.

[104] Z. Xu, B.P. Miller and O. Naim. Dynamic Instrumentation of Threaded Applications. *7th SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), Atlanta, GA, May 1999.

[105] Z. Xu, B. Miller and T. Reps. Safety Checking of Machine Code. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, BC, June 2000.

[106] Z. Xu, T. Reps, and B. Miller. Typestate Checking of Machine Code. *European Symposium On Programming (ESOP)*, Genova, Italy, April 2001.

[107] X. Zhang, Z. Wang, N. Gloy, J.B. Chen, and M.D. Smith. System Support for Automatic Profiling and Optimization. *16th ACM Symposium on Operating System Principles (SOSP)*, Saint-Malo, France, October 1997.