JIT Instrumentation - A Novel Approach To Dynamically Instrument Operating Systems

Marek Olszewski †, Keir Mierle †, Adam Czajkowski †, and Angela Demke Brown ‡

† Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
{olszews, keir, aczajkow}@eecg.toronto.edu

‡ Department of Computer Science
University of Toronto
Toronto, Ontario M5S 3G4, Canada
demke@cs.toronto.edu

ABSTRACT

As modern operating systems become more complex, understanding their inner workings is increasingly difficult. Dynamic kernel instrumentation is a well established method of obtaining insight into the workings of an OS, with applications including debugging, profiling and monitoring, and security auditing. To date, all dynamic instrumentation systems for operating systems follow the probe-based instrumentation paradigm. While efficient on fixed-length instruction set architectures, probes are extremely expensive on variable-length ISAs such as the popular Intel x86 and AMD x86-64. We propose using just-in-time (JIT) instrumentation to overcome this problem. While common in user space, JIT instrumentation has not until now been attempted in kernel space. In this work, we show the feasibility and desirability of kernel-based JIT instrumentation for operating systems with our novel prototype, implemented as a Linux kernel module. The prototype is fully SMP capable. We evaluate our prototype against the popular Kprobes Linux instrumentation tool. Our prototype outperforms Kprobes, at both micro and macro levels, by orders of magnitude when applying medium- and fine-grained instrumentation.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—Monitors

General Terms

Performance, Measurement, Experimentation

Keywords

Dynamic Instrumentation, Binary Rewriting, JIT Compiler, Kernel Analysis Tools

EuroSys'07, March 21–23, 2007, Lisboa, Portugal.

1. INTRODUCTION

Modern monolithic operating systems (OSs) are complex evolving constructs spanning tens of millions of lines of code. System administrators and developers frequently need to understand the inner workings of OSs for purposes such as debugging [13], profiling and monitoring [5, 9], and security auditing [10]. Kernel instrumentation is a well established method of obtaining the necessary insight and is often done *dynamically*, because it can be applied without the need for kernel recompilation or rebooting. This is particularly important in production systems where taking a system offline is not a viable option. In addition, dynamic instrumentation eases the analysis of systemic problems or *emergent misbehavior* [17], which only appear after prolonged and continuous system operation.

To date, all dynamic instrumentation systems for operating systems use the probe-based instrumentation paradigm. This approach works by overwriting instructions in the original program with trampolines to instrumentation code. Probes can be implemented efficiently on fixed-length instruction set architectures (ISAs), such as Sun's UltraSparc, by inserting jump instructions to the relevant instrumentation code at each instrumentation point; however, on variable-length ISAs, such as the popular Intel x86 and AMD x86-64, probes have to be implemented with trap instructions [25]. At each instrumentation point, execution of the trap instruction causes an exception handler to be dispatched. The handler must then determine what type of instrumentation (if any) is needed at that point, typically via a hash table lookup. The overhead of these steps makes comprehensive and fine-grained instrumentation unacceptably slow on these architectures. For example, when using the Linux Kprobes framework, we find that recording a list of functions invoked during a system call can slow execution by nearly a factor of fifty.

We propose using just-in-time dynamic instrumentation (or JIT instrumentation for short) instead. In this approach, execution is redirected to a runtime system at the entry of a section of code that is to be instrumented. A JIT compiler creates a duplicate copy of each basic block of the original code immediately before it is executed, embedding calls to instrumentation routines within it, much as if the instrumentation had been added to the source code and the source recompiled. The resulting instrumented basic blocks are stored in a *code cache*, from whence they are dispatched instead of the original code. We provide additional back-

ground on existing operating system instrumentation techniques, and on JIT instrumentation in Section 2.

JIT instrumentation provides both better performance and a better usage model than probe-based techniques for fine-grained instrumentation. The primary performance advantage stems from the fact that instrumentation can be inserted between instructions. This eliminates the need for expensive trap instructions to redirect execution to instrumentation code on variable-length ISAs. In addition, when using a JIT, instrumentation is only inserted into code after it is known that it will execute, thereby eliminating any cost of instrumenting instructions that *might* be executed. Furthermore, if the instrumentation code is small enough, it can be *inlined* directly into the copied basic blocks to eliminate the cost of executing function calls. Further optimizations may also be performed. For example, register and condition code *liveness analysis* can reduce the amount of saved processor state before each instrumentation point in the basic block. We describe our prototype implementation of a JIT instrumentation tool for operating system code which performs these optimizations in Section 3, and show its performance advantages over a probe-based strategy for fine-grained instrumentation in Section 4.

Probe-based instrumentation requires a user to specify the exact locations in the code where instrumentation should be inserted, which can become onerous for fine-grained instrumentation. In contrast, JIT instrumentation requires only entry, and possibly exit, points to be identified. Because instrumentation is added as code blocks are discovered, there is no need for *a priori* identification of possible instrumentation points (either by a user directly, or by a tool). In addition, making a *copy* of instrumented code paths makes it easier to isolate properties of interest. For example, suppose we wanted to know how often `kmalloc()` is invoked, directly or indirectly, as a result of the `execv` system call. With a JIT, the instrumented version of `kmalloc()` is only called as part of `execv`—all other uses invoke the original, uninstrumented code and are therefore automatically excluded from the count. With probe-based instrumentation, it is easy to count entrances to `kmalloc()`, but a significantly more complicated instrumentation routine would be needed to determine whether the call was due to `execv` or not. Worse, all other uses of `kmalloc()` would pay the extra cost of this check. The power and simplicity of the JIT instrumentation user model has proven itself in a number of user space tools such as Pin [15] and Valgrind [18].

We demonstrate the utility of the JIT model for kernel instrumentation by showing several example *plugins* in Section 5. We discuss other approaches for kernel instrumentation, contrasting them with our approach and showing where our work is complementary, in Section 6.

Our framework is the first JIT instrumentation system that can be applied to live operating system kernels. In this work we show that JIT instrumentation can be implemented as a self-contained kernel module that can be loaded and unloaded as needed. We also show that it can deliver substantial performance improvements for fine-grained instrumentation over existing trap-based systems.

Our contributions are:

1. A prototype Linux kernel module implementation that shows that JIT instrumentation is possible and desirable in the kernel.
2. A performance comparison between our prototype and an existing kernel probe implementation, which shows that our approach is faster by up to two orders of magnitude for fine-grained instrumentation.
3. Example plugins that show the power and flexibility afforded by the system, yet remain simple and intuitive to implement.

We end the paper with a discussion of areas for further investigation and conclusions in Sections 7 and 8.

2. INSTRUMENTATION BACKGROUND

In this section we give additional background on existing operating system instrumentation strategies, both static and dynamic, and on general JIT compilation and instrumentation systems.

2.1 Operating System Instrumentation

The most common strategy for instrumenting operating systems involves changing the source code and recompiling the kernel. The Linux Trace Toolkit [26] takes this approach by providing patches for the Linux source code to enable a small number of performance monitoring probe points. If the OS source code is not available, static instrumentation can be added by binary re-writing.

While many binary re-writing tools exist for user space applications (e.g., ATOM [24], Purify [12], EEL [14] and PLTO [23]), there are few designed to modify operating systems. Flower et al. [11] point out some of the challenges they faced in modifying the Spike executable optimizer for kernel binaries, including the kernel's use of self-modifying code and assumptions about the order or location of particular basic blocks, in particular for code that is only used during system boot.

The main advantage of static instrumentation is the ability to apply sophisticated but slow compiler optimizations to the instrumented code. Unfortunately, whether added to kernel source code or a binary, it can only be fully disabled by rebooting to an uninstrumented version. The constant overhead of this "always on" property makes fine-grained static instrumentation unsuitable for production systems. As a result, researchers and system users have turned to dynamic techniques that allow instrumentation to be added and removed as needed. K42, an object oriented research operating system, offers a compromise between the two methods. Statically instrumented modules can be dynamically hot-swapped for their original counterparts without interrupting services [2].

Dynamic instrumentation of arbitrary kernel code was first introduced by Tamches and Miller with their work on the KernInst project [25]. KernInst targets the fixed-length instruction set RISC UltraSparc architecture, and is thus able to safely overwrite kernel code with branch instructions that redirect execution to instrumentation routines. These branches can replace a majority of instructions and therefore almost all operating system code can be instrumented dynamically. Since instrumentation is inserted at runtime, it adds no overhead until enabled, which is one of the key advantages of this approach over traditional static instrumentation methods.

Attempts to tailor the KernInst directly-inserted-branch style of dynamic instrumentation to the popular variable-length x86 architecture have proven unsuccessful. For this

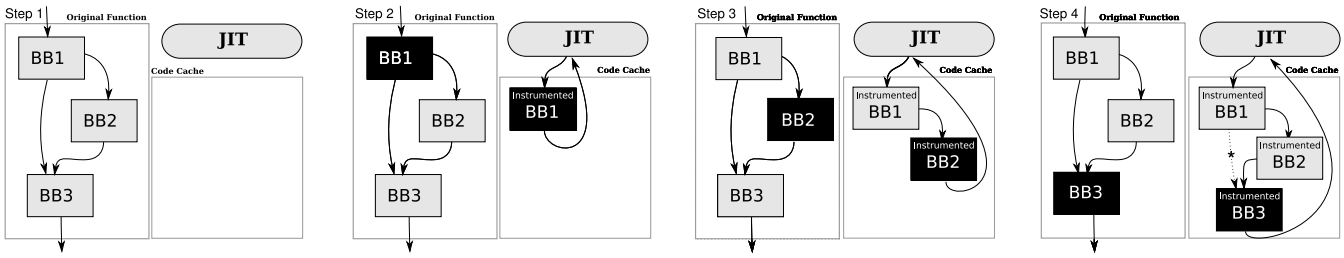

Figure 1: A brief look at the JIT process. Step 1 shows the initial state. Step 2 shows the result of copying the first basic block into the code cache. Step 3 shows how the branch targets of existing cached blocks are updated as destination blocks are brought into the code cache. Step 4 shows the final result, with all three basic blocks now instrumented and executing from the code cache.

ISA, the smallest branch instruction is larger than the smallest instruction, and thus patching operating system code with new branch instructions at runtime is not safe for several reasons. First, on preemptable kernels it is possible that a second thread may be sleeping between two short instructions overwritten by the branch. When the sleeping thread awakens, the processor will try to decode a series of bits that were not intended to be a valid instruction opcode. Second, there is a chance that the inserted branch instruction will overwrite instructions that are not contained within the same basic block. Here, the processor might jump into the middle of the branch instruction with, once again, undesirable results. The GILK [21] project attempts to overcome the second of the two problems by statically analyzing the Linux kernel binary to determine all basic block boundaries. Unfortunately, indirect branch and call instructions limit the accuracy of such analysis as these instructions have branch targets that are not known until runtime.

When targeting the Intel x86 ISA, projects like KernInst, Kprobes [20] and DTrace [7] insert trapping break instructions instead of branches. These instructions are also used for inserting debugger breakpoints, and are intentionally the shortest instructions in the ISA (1 byte). Thus, they can be used to overwrite any instruction, thereby avoiding the two aforementioned problems. By modifying the trap handler, break instructions can be used to execute instrumentation. However, if multiple types of instrumentation exist, a hash table lookup is necessary to determine which instrumentation routine should be called. This lookup is expensive and can seriously degrade performance [6], particularly when the number of instrumented instructions becomes large.

A more detailed overview of related work and the comparison to our approach can be found in Section 6.

2.2 JIT Instrumentation

Traditionally, a trap or jump instruction is inserted at each instrumentation point to redirect control to the appropriate instrumentation. JIT instrumentation takes a different approach: simply rewrite the code to a new location and thus avoid adding jumps or traps. In the process of rewriting the code to a new location, instrumentation is directly inserted into the rewritten code. By rewriting instead of overwriting original code, variable-length instructions are implicitly handled, allowing efficient instrumentation on variable-length ISAs.

At the heart of any JIT instrumentation system is the JIT compiler, or JIT for short. An instrumentation JIT works much like a Java bytecode JIT (e.g., Sun's HotSpot [19] or IBM's Jalapeno [1]), or the JIT in a dynamic binary translator (e.g., HP's Dynamo [3] or Sun's Walkabout [8]). Java JITs translate bytecode to native code to improve performance, and binary translators (e.g., QEMU [4]) typically translate from one ISA to another for a host of reasons, such as running legacy binaries on new hardware. The instrumentation JIT, in contrast, produces code in the same ISA as its input (similar to QEMU accelerator and VMWare virtualization), but with additional instrumentation instructions inserted. A JIT can compile at various granularities; a Java JIT typically compiles at a method granularity, while user space JIT instrumentation tools and binary translators prefer basic block or trace granularities.

Figure 1 shows the JIT instrumentation process for a simple graph of three basic blocks. Initially, in Step 1, no code has been instrumented and the code cache is empty. When execution reaches `BB1`, it is redirected into the JIT. The JIT makes a copy of `BB1`, inserting the required instrumentation and modifying the final instructions to first record the original destination, `BB2`, and to then redirect execution back to the JIT at the end of the block. The new block is then placed into the code cache, as shown in Step 2 of Figure 1. When control returns to the JIT after executing the instrumented `BB1`, the JIT sees that the next block to execute is not in the code cache. Therefore, the JIT copies `BB2` into the code cache, transforming it as with `BB1`. Additionally, the JIT updates `BB1` to branch directly to the instrumented copy of `BB2`, rather than the JIT. The result is shown in Step 3 of Figure 1. This process continues in Step 4 as `BB3` is discovered, instrumented, and placed in the code cache. Note that had control flowed directly from `BB1` into `BB3`, through the dotted branch marked with a * in Step 4, then `BB2` would never enter the code cache at all; this is the nature of JIT instrumentation—only executed code is instrumented.

Since the resulting instrumented code is stored and executed from a new location, the JIT must modify all control instructions to account for this change, as shown in the simple example of Figure 1. Direct control instructions can be updated to point directly to their code cache equivalent targets, if present. Indirect control instructions such as indirect jumps, calls, and return instructions have targets that cannot be resolved at JIT-time, and therefore must be translated at runtime. These instructions are modified to point back to a runtime system, which will perform the translation, invoking the JIT if the target is not in the code cache.

JIT instrumentation can only happen if there is an *entry point* from which control can be redirected to the JIT. After that point, all further execution happens within the JIT engine. User space tools use a number of methods of achieving this. One approach, taken by Pin, is to gain control using

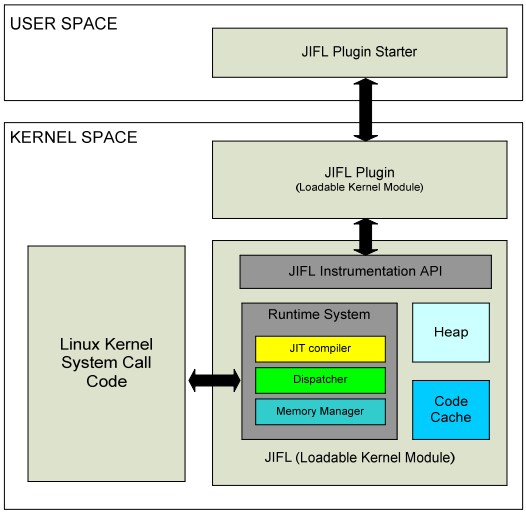

Figure 2: JIFL's Software Architecture

ptrace. Pin injects its JIT into the address space of the application (using the same primitives debuggers use to insert breakpoints), and modifies the application program counter to point to it. A final advantage of JIT instrumentation is that it never modifies the original code. This means that any code that makes assumptions about the original structure of the binary will continue to operate as it did before. It also makes JIT instrumentation very easy to disable, since only the entry point needs to be redirected to the original code, rather than repairing every instrumentation point.

JITs used for instrumentation must interface with a user-supplied plugin that specifies what instrumentation to insert. The JIT applies this instrumentation and optimizes the resulting code, thus specializing the instrumentation for the given context. Register re-allocation, liveness analysis and instrumentation inlining are examples of optimizations that can be performed. The end result is a highly optimized instrumented version of the original code. Since the translation and optimization are one-time costs, the benefits of specialized instrumentation in frequently-executed code makes them worthwhile.

3. DESIGN OF THE JIFL PROTOTYPE

In this section we describe the design of our JIT instrumentation prototype, called JIFL (JIT Instrumentation Framework for Linux). JIFL allows users to instrument system calls on 2.6 Linux uniprocessor and symmetric multiprocessor (SMP) kernels.

3.1 System Overview

JIFL is implemented as a self-contained loadable kernel module. Figure 2 illustrates JIFL's software architecture. At the highest level, JIFL consists of a runtime engine, a private heap, a code cache and an instrumentation API for interfacing with JIFL plugins. JIFL plugins dictate what instrumentation to apply, and can be started and stopped with a user space tool. The runtime engine contains the JIT compiler, dispatcher, and a custom memory allocator.

3.2 JIFL Plugins

JIFL plugins are simply separate kernel modules that depend on the main JIFL module. They can be loaded and unloaded using standard Linux commands. The JIFL starter tool is provided to allow a user to turn off instrumentation before attempting to unload the module. This is important because, otherwise, the plugin may be instrumenting the system calls being used to unload it.

```
#include "jifl.h"

syscall_t syscall;
long long count;

void add_count(long long *counter_ptr, long size);
void bb_inst(bb_t *bb, void *arg);

// Start instrumentation of clone system call
void plugin_start() {
    count = 0;
    syscall_init(&syscall, __NR_clone);
    syscall_add_bb_instrumentation(&syscall,
        bb_inst, NULL);
    syscall_start_instrumenting(&syscall);
}

// Stop instrumentation of clone system call
void plugin_stop() {
    syscall_stop_instrumenting(&syscall);
    printk("Clone system call executed %lld "
           "instructions\n", count);
}

// Called for every newly discovered basic block
void bb_inst(bb_t *bb, void *arg) {
    long bb_size = bb_size(bb);
    bb_insert_call(bb, add_count,
        ARG_VOID_PTR, &count,
        ARG_INT32,    bb_size,
        ARG_END);
}

// Executed for every instrumented basic block
void add_count(long long *counter_ptr, long size) {
    *counter_ptr += size;
}
```

Figure 3: Example plugin used to count the number of instructions executed by the clone system call.

Figure 3 demonstrates the JIFL instrumentation API with a simple instrumentation plugin that counts the number of instructions executed by the `clone` system call. The plugin starts by specifying the `clone` system call with a call to `syscall_init()`. It then adds the `bb_inst()` basic block level callback, which will be called by JIFL every time it discovers a previously un-executed basic block. Finally, it calls `syscall_start_instrumentation()`, which redirects all `clone` system calls to execute through JIFL.

During execution, the JIT calls the `bb_inst()` callback for every basic block it encounters to determine what instrumentation it needs to insert. The callback can use a rich JIFL API to examine the basic block instructions and decide whether to apply any instrumentation; this is discussed in detail in Section 5. In this particular example, all basic blocks are instrumented with a call to `add_count`, with the address of a global counter and the size of the basic block passed as parameters.

When the plugin is stopped, it directs JIFL to stop executing `clone` through the code cache, and simply prints the counter to the console.

3.3 Gaining and Releasing Control

To execute a system call through the JIFL runtime engine, the JIFL dispatcher must gain control before any of the system code is executed. JIFL achieves this by patching

the system call table to redirect execution to itself. Because JIFL has no way of knowing which of the table entries was followed to get to it, JIFL overwrites the table entry of any desired system calls with the address of a dynamically-generated *entry stub*; each stub has the true system call address hard-coded so it can be passed to the dispatcher. The stubs also increment the usage count of the instrumentation plugin module to prevent the plugin and JIFL modules from being unloaded while they are in use. Finally, the stubs call any system-call level instrumentation.

JIFL redirects control out of the code cache back to the original operating system code in two places: at the end of a system call, and at any calls to `schedule()`. The second is required to prevent JIFL from executing code after a context switch. JIFL must ensure that control is returned to it once the current thread is scheduled again. To achieve this, the JIT rewrites all calls to `schedule()`, with jumps to `jifl_schedule()`, which saves the return address in a hash table (which can be looked up by the `current` task pointer) before calling `schedule()`. Upon returning, it finds the return address in the hash table and calls the dispatcher. For system call exits, and calls to `schedule()` that do not return (such as the final one in `sys_exit()`) JIFL inserts code to decrement the plugin module's reference counter.

3.4 The Dispatcher

The dispatcher is responsible for saving and restoring the system call's register and condition code state, as well as locating (via a hash table) and redirecting control to the code cache version of the next basic block to be executed. If the basic block does not exist in the code cache, the JIT compiler is invoked to insert it there. Because some of these tasks are low level, parts of the dispatcher are written in assembly.

The operation of the dispatcher, and its interaction with the JIT compiler, are illustrated in detail in Figure 4. To understand the figure, follow the numbered steps in parts B and C noting that circled steps have corresponding arrows showing either flow of control (solid arrows) static control flow (dashed arrows) or data movement (thick solid arrows). The following section expands on these steps.

3.5 The JIT Compiler

The JIT compiler copies the system call's machine code into the code cache, while instrumenting it as specified by the JIFL plugin. JIT compilation occurs on a basic block level to ensure that only executed code is copied into the code cache, thus keeping the cache as small as possible. This is important because kernel virtual memory is often limited (unlike for user-space JIT systems).

Note that JIFL does not perform static analysis to obtain the boundaries of basic blocks. OS machine code contains many indirect branches, preventing accurate static analysis. Instead, the JIT operates on *dynamic* basic blocks, which are sections of code that contain exactly one control-flow instruction positioned at the end of the section. This is different from the typical definition of a basic block, where branch targets define a new basic block, allowing some basic blocks to end with a non-control-flow instruction. If, while compiling a basic block, JIFL discovers a branch instruction with a target pointing into the middle of an already compiled basic block, a new dynamic basic block is created by copying the targeted instruction and those below it. This approach simplifies analysis and instrumentation at the cost of some duplication in the code cache; user space JIT instrumentation tools use a similar strategy [15].

When compiling a basic block, all but the final control instructions are simply copied into the code cache version. Direct branches, such as the x86 `jcc`, `loopcc`, and `jmp` instructions, are modified so that all control flow is redirected back to the dispatcher. `Call` instructions are converted to `push` and `jmp` instructions. To preserve the contents of the stack, the original non-code cache return address is pushed to the stack. This serves two purposes. First, it enables JIFL to detach itself at any time (in the event of an error) by returning control to the original non-instrumented code. Second, it ensures that any code that depends on the value of this return address will continue to function correctly. For example, call instructions can be used to push the value of the program counter to the stack so that it can be read. Any code depending on this method for retrieving the contents of the program counter will continue to function correctly since the original return address is still pushed to the stack. Indirect `jmp` and `call` instructions are modified in a similar fashion to their direct counterparts, however the address of the next basic block passed to the dispatcher is no longer a constant but rather calculated at runtime. Return instructions are handled like indirect jumps. Their branch targets are also runtime dependent and are obtained by popping the return address off of the stack.

As an optimization, the JIT attempts to link compiled basic blocks directly whenever possible. This is achieved by first checking if the branch or fall-through target of the control instruction is already in the code cache. If so, a `jmp` instruction which jumps to this basic block is inserted into the stub code. This can be seen in Step 13 of Figure 4. The JIT also patches all basic blocks in the code cache that have branch/fall-through targets pointing to the current basic block being compiled so that their subsequent invocations also avoid the dispatcher. Since the branch target of indirect calls, jumps, and returns is not known at JIT compile time, basic blocks ending with such instructions cannot be linked as easily. We apply the same predicated-linking technique described by Luk et al.[15] to link common indirect targets. A chain of comparison stub code is built incrementally at runtime. Each node in the chain checks the target address of the indirect branch instruction to a value seen in the past. If the comparison succeeds, the code jumps to the instrumented version of the target, otherwise, it jumps to the next node. If all comparisons fail, the dispatcher is called to perform a hash table lookup on the address. The dispatcher also inserts a new node in the chain for this address until a maximum chain size is reached.

While performing compilation, the JIT also adds any desired instrumentation. Since the compiled dynamic basic block will only be executed from the start, adding instrumentation is essentially as easy as inserting call instructions anywhere in the basic block. However, instrumentation routines may modify the state of the processor, and therefore, instructions that save and restore register and condition code states have to be inserted as well. The JIT performs register and eflags (x86's condition code flags) liveness analysis to reduce the number of these instructions that need to be inserted. Further, if the instrumentation routine is small enough, the JIT will attempt to inline it into the basic block. The following sections describe these optimizations in detail.

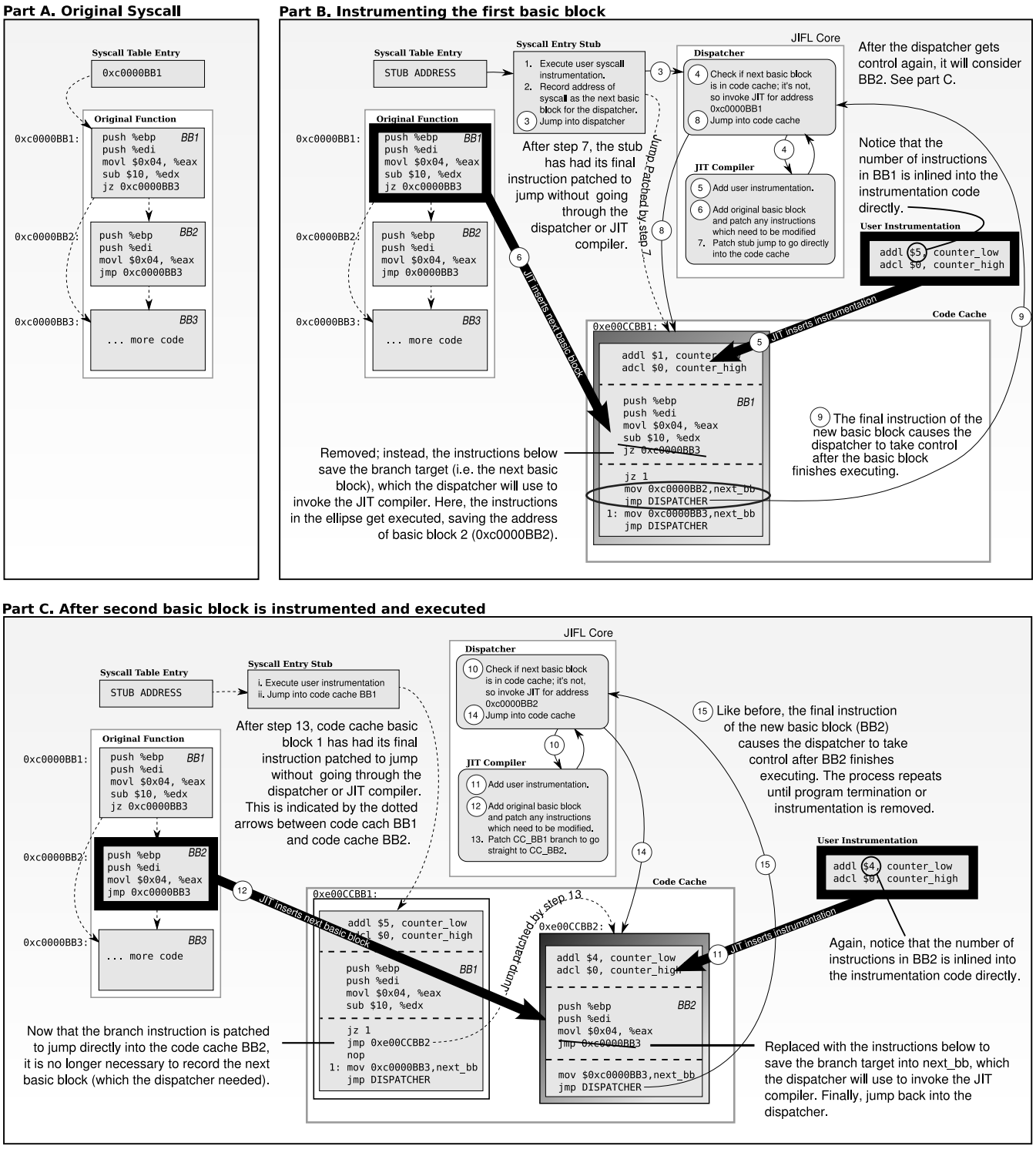

Figure 4: A detailed look at JIFL instrumentation in action. The numbered text describes the sequence of tasks performed by JIFL. Solid arrows depict the flow of control, while dashed lines show static control flow. Thick arrows represent data copying. Steps with circled numbers next to their text also have a matching circled number on the arrow corresponding to the step. The instrumentation inserted is the `add_count` function shown in Figure 3 after inlining.

```
void inc_count(long long *counter_ptr)
{
    (*counter_ptr)++;
}
```

Figure 5: Example 64-bit increment function

```
movl    4(%esp), %eax
addl    $0x01, (%eax)
adcl    $0x00, 4(%eax)
ret
```

Figure 6: gcc-generated assembly code for Fig. 5

3.5.1 Register and Eflags Liveness Analysis

To eliminate redundant state saving instructions, liveness analysis is performed on the instrumentation routine and original OS code to determine the minimum set of condition code flags (eflags) and registers that need to be saved. Liveness analysis proceeds by disassembling instructions following the desired position of instrumentation and checking whether registers or eflags are overwritten (i.e., *killed*) before being used as input. Those that are used before being killed are *live*, and are considered vulnerable if the instrumentation modifies them. If a direct control instruction is encountered, its target basic block is analyzed in the same fashion. Repeated analysis of the same basic block is avoided by entering the address of an already-analyzed basic block in a dedicated hash table. Indirect control instructions are treated conservatively and assumed to lead to a basic block that uses all registers and eflags. The set of vulnerable registers (including the eflags register) must be saved before, and restored after, the instrumentation routine.

When performing instrumentation with basic block granularity, the JIT is free to insert instrumentation anywhere in the basic block. In this case, the JIT will use the liveness analysis results to find the position where the least amount of state needs to be saved.

3.5.2 Instrumentation Inlining

JIFL achieves a large portion of its performance by inlining small instrumentation routines directly into the duplicated operating system code. During inlining, JIFL can also specialize the instrumentation for any parameters that will not change at runtime. For example, when instrumenting the direction of individual branches, the per branch counter address passed to an instrumentation routine (such as the one shown in Figures 5 and 6) will remain constant for all invocations of each branch and can therefore be propagated into the inlined routine.

To achieve these optimizations, standard compiler optimizations such as constant propagation, constant folding, copy propagation and dead-code elimination are also applied. To reduce the complexity of the compiler code, we take the approach of Pin by using architecture-specific optimizations that operate directly on machine code. This is in contrast to the traditional approach of converting all machine code into an architecture-independent intermediate language, optimizing, and converting back. Such an approach is taken by Valgrind; however, we believe that the added complexity and high performance cost is not suitable in kernel space. Furthermore, Pin demonstrates that using their approach, a fair amount of the JIT source code can remain architecture-independent [15].

```
addl    $0x01, 0xCC123400(,0)
adcl    $0x00, 0xCC123404(,0)
```

Figure 7: Fully specialized version of Figure 6 ready for inlining; the address of the counter is now an immediate operand and no longer passed on the stack.

JIFL starts inlining by placing all instrumentation instructions into a linked list so that they can be better manipulated. Next, it generates the static control flow graph; if any indirect jumps are encountered, inlining is aborted since JIFL cannot determine the targets of these jumps. All *move* instructions which read the constant stack parameters are converted into moves that read their immediate operands instead. While the compiler can create many other instructions that access the parameters directly, we found that most parameters have to be dereferenced and therefore need to be moved into a register before use. We are currently working on handling all possible accesses to the parameters. If all accesses to a parameter are removed, the parameter is also removed and all stack accesses are modified to account for the change.

Next, copy propagation is performed to eliminate any needless moves. Dead-code elimination is used to remove the remaining copies if all references to the copied register have been propagated out. Since we currently lack a data-flow solver, these two steps are only performed if the instrumentation routine is composed of a single basic block. Global copy propagation and dead-code elimination are planned for the future.

Finally, the specialized routine is laid out as a continuous sequence of instructions in memory. Since the sizes of basic blocks most likely will have changed, special care must be taken to patch up the relative branch target offsets of control instructions. Return instructions must also be either removed, or converted to relative jumps that point to the end of the inlined code. The resulting code is cached so that it can be reused if subsequent instrumentation inlining needs to be performed for the same instrumentation routine with the same parameter values. Figure 7 shows the final effect of our specialization passes on the code of Figure 6.

3.6 Memory Allocator

JIFL often needs to allocate dynamic memory while performing JIT compilation or analyzing code. Because Linux's memory allocators are not reentrant, JIFL must avoid using them as it might be operating on behalf of a thread currently executing its own memory allocation request. Doing so could result in deadlock or a corrupt system state. Instead, JIFL preallocates and manages its own memory with a custom memory allocator. We found that a simple, slightly optimized, implicit next fit memory allocator was sufficient for our needs. JIFL plugins must also use this memory allocator within their own routines.

If at anytime JIFL becomes low on preallocated memory, it sets a flag to flush its code cache and hash tables to free up more memory the next time the dispatcher is called. In the unlikely event that the heap is filled before that time, JIFL recovers gracefully by restoring the current thread's program counter to the original operating system code and letting it finish the system call without instrumentation. JIFL performs the flush the next time a system call is entered. Once the code cache is flushed, JIFL must

re-JIT and instrument all executed basic blocks again. We have found that a modest amount of memory (5 megabytes per processor) is sufficient to keep the occurrence of flushing low.

3.7 SMP Considerations

JIFL runs efficiently on SMP kernels. Each processor maintains its own private code cache and heap. Despite the additional memory required, private code caches are desirable because they enable the JIT to specialize instrumentation per processor when performing inlining, and almost entirely avoid locks. Each processor also requires a private dispatcher, so that it can save state to global memory without needing to check what processor it is running on. Therefore, the generated system call entry stubs must perform a CPU check to determine which dispatcher to jump to.

Calls to `schedule()` require additional handling on SMP kernels because the process can be migrated to a new processor while sleeping. Therefore, when a system call wakes up and returns to `jifl_schedule()`, JIFL checks which processor it is executing on before calling the appropriate dispatcher. In addition, the hash table with return addresses must be protected by a lock.

4. EVALUATION

This section presents performance results for JIFL, comparing our prototype to Kprobes, the current dynamic instrumentation tool of choice for the Linux kernel. We evaluate both micro and macro performance by using the LMbench [16] benchmark suite and by testing the performance of an Apache 2 web server running with our instrumented kernel. In each experiment, we applied instrumentation to every system call in the kernel.

Our testbed consists of a 4-way Intel Pentium 4 Xeon (2.8 GHz) SMP equipped with 16GB of Memory and L1, L2, and L3 cache sizes of 8KB, 512KB, and 2MB, respectively. The base OS distribution is Debian testing with Linux 2.6.17.13. This version of the kernel contains the most recent version of Kprobes which includes a recent booster patch that improves probe execution time by up to three times on certain instructions[1]. In addition, we increased the number of buckets in the Kprobes probe hash-table from the default of 64 to 1M, to reduce the number of collisions when inserting large numbers of probes. Finally, we compiled the kernel without debugging support to eliminate the additional overhead incurred by the debugger, which shares the breakpoint trap handler with Kprobes.

4.1 Instrumentation

To evaluate the relative performance of JIFL and Kprobes, we use three types of instrumentation, each representing a different instrumentation granularity ranging from coarse to fine-grained. We also show the overhead of executing system calls through the JIFL runtime with no instrumentation.

As an example of coarse-grained instrumentation, we use system call monitoring, in which the goal is to count how many times a system call is invoked (per CPU). For each system call, our JIFL plugin specifies a system call entry callback containing the counter increment code, thus forcing all system calls to run through the code cache. This callback is slightly more expensive than regular instrumentation as we currently do not attempt to perform inlining or liveness analysis on it. For Kprobes, a single probe that increments a counter at the entry of each system call is sufficient.

To evaluate medium-grained instrumentation we use *call tracing*, in which the sequence of function calls executed during a system call for a particular CPU is recorded. We describe our call tracing plugin in Section 5.1. The call trace is stored in a memory buffer and can be printed at the end of the system call. For Kprobes, call tracing is potentially tedious to implement, because there is no obvious automated way to collect the appropriate probe point addresses. We wrote a simple JIFL plugin that stores and prints out the addresses of all functions called after running our benchmark. We use these probe addresses in our Kprobes call tracing implementation.

For fine-grained instrumentation, we use *basic block counting*, in which we count the number of dynamic basic blocks executed by each system call. Using JIFL, we specify a basic block level callback that inserts our instrumentation into each basic block. The instrumentation simply increments a per system call and per CPU counter. As with call tracing, there is no clear way to find the appropriate probe points for counting basic blocks with Kprobes. We employ a similar technique as before, using JIFL to obtain the addresses of all control flow instructions typically executed when running our benchmark. While the applications for basic block counting may be limited, we found that performance is comparable to other more interesting instrumentations (such as the instruction counting example in Figure 3), which are not easily implementable with probes.

Finally, we include the performance of executing system calls through JIFL, with no instrumentation applied. These numbers represent the upfront cost that must be incurred when executing through our runtime system. It is important to remember that this will only be the case for system calls that are actively being instrumented. When no instrumentation is enabled, JIFL introduces no performance penalty.

4.2 Effect on System Call Performance

For our micro benchmark, we ran the LMbench 3.0 benchmark suite with each of our instrumentations. All tests are averaged across 3 runs in addition to LMbench's internal averaging. These tests show JIFL's performance under steady state, i.e. all code is executed from the code cache and the JIT compiler is no longer invoked, since LMbench performs its own warm-up cycles [16].

Figure 8 displays the execution times of a number of key system calls, normalized by the uninstrumented case and displayed on a logarithmic scale. The first bar shows the overhead of running the system call code through JIFL's code cache without any instrumentation. The majority of this overhead can be attributed to the original-to-code-cache address translation that occurs for every indirect call, indirect jump, and return instruction.

The remaining bars compare the performance of JIFL with that of Kprobes for the three types of instrumentation: system call monitoring, function call tracing and basic block counting. Even when simply monitoring system calls, JIFL is faster than Kprobes for three of the microbenchmarks. In other cases, the overhead of a single probe at the entry to the system call is less than the overhead of running from the

[1]Regrettably, we were forced to revert back to Linux 2.6.16.29 for the Kprobes basic block counting runs due to instabilities when inserting large numbers of probe points.

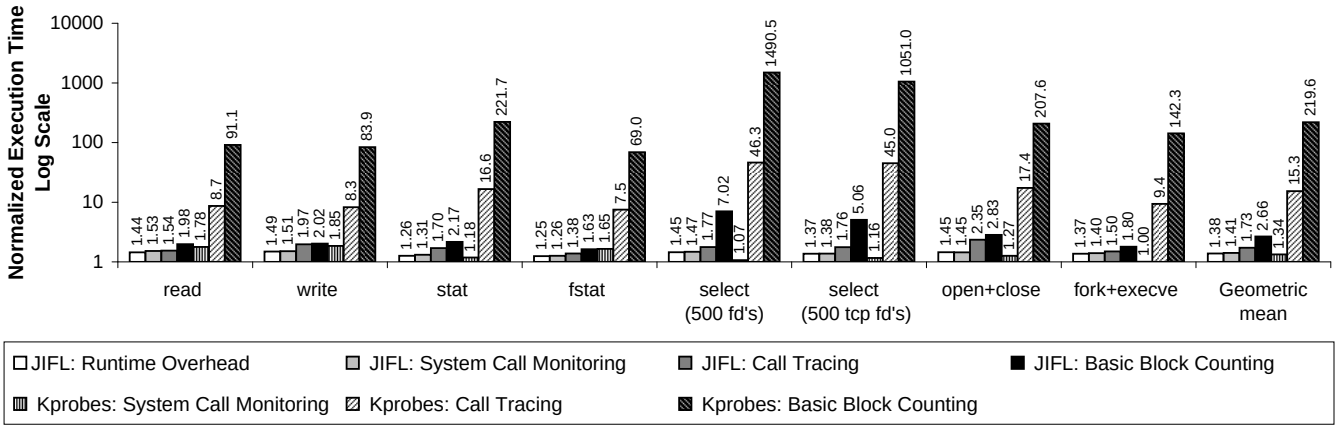

Figure 8: LMbench micro-benchmark performance comparison. Note the logarithmic scale.

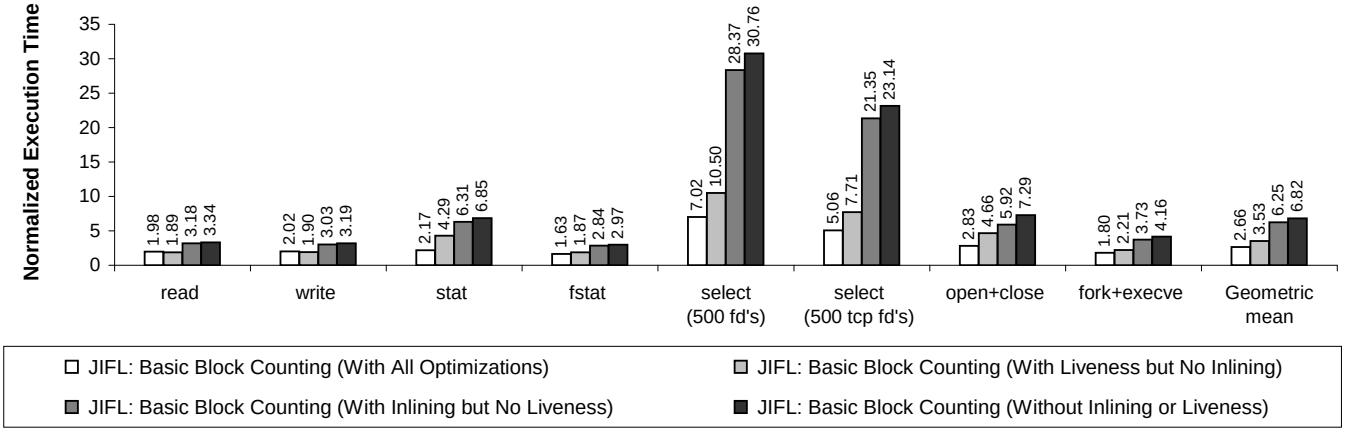

Figure 9: LMbench system call latencies showing the effect of inlining and liveness analysis, when performing fine-grained instrumentation

code cache. On average, the slowdown due to Kprobes, even for very coarse-grained monitoring, is only 0.07 less than the slowdown due to JIFL. When applying finer-grained instrumentation JIFL dramatically outperforms Kprobes. For call tracing, JIFL's slowdown is almost an order of magnitude less than that of Kprobes, on average; the disparity grows to close to two orders of magnitude on average for basic block counting. The high cost of the trap and address lookup for each instrumentation point makes probe-based instrumentation techniques very inefficient on the variable-length x86 ISA.

In Figure 9, we present the effects of liveness analysis and instrumentation inlining on the performance of JIFL. Results are again normalized to the uninstrumented case. We can see that the optimizations yield substantial gains, reducing the slowdown due to instrumentation by more than a factor of 4 in the best case, and a factor of 2.6 on average. It is interesting to note that liveness analysis accounts for most of this improvement, which can be implemented for probe-based instrumentation on fixed-length instruction set architectures. However, the further gains seen by instrumentation inlining (reducing slowdown by nearly a factor of 2 in the best case, and a factor of 1.3 on average) cannot be achieved with probe-based approaches on either fixed or variable length instruction set architectures.

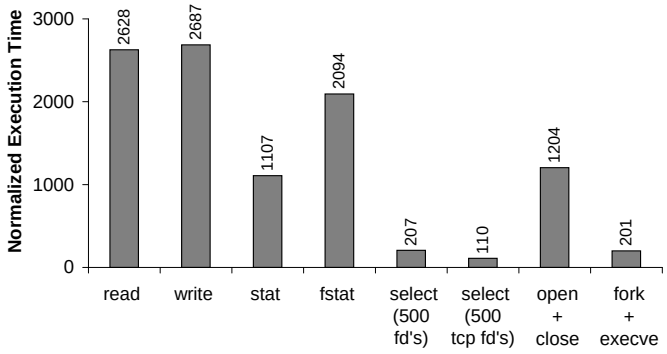

Figure 10: LMbench system call latencies when flushing the code cache after every system call.

Finally, Figure 10 gives a pessimistic estimate of JIFL's JIT cost obtained by flushing the code cache between system call executions. Therefore, it includes the cost of freeing all associated data structures in the heap, as well as the added cost of allocating memory in a fragmented heap on subsequent runs. While the slowdowns are substantial, JIT overhead is typically a one-time cost which can be amortized over a large number of invocations of the instrumented calls. In the following section, we consider the effect of JIFL on

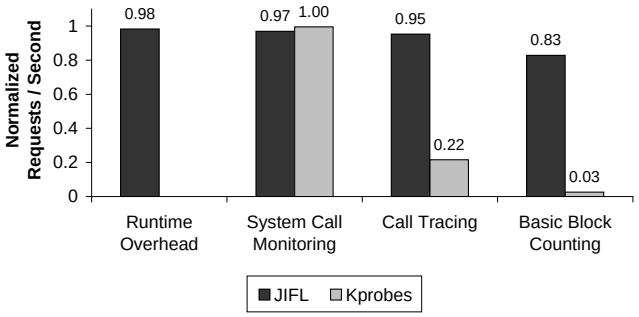

Figure 11: ApacheBench2 throughput comparison. Bigger is better.

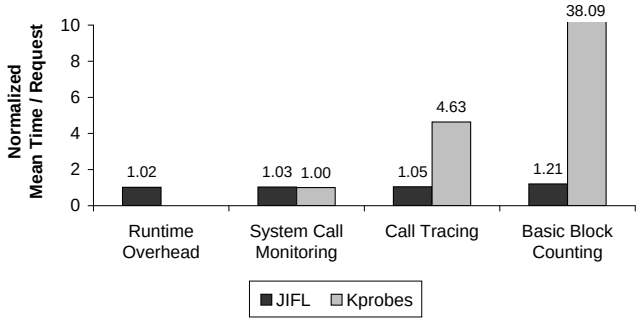

Figure 12: ApacheBench2 request latency comparison. Smaller is better.

a macrobenchmark, which includes JIT compilation and instrumentation overhead in a more realistic setting.

4.3 Effect on Application Performance

We used the Apache 2.0 web server and the ApacheBench2 benchmarking tool to test macro performance. We executed ApacheBench on a single uninstrumented client machine connected to the test machine via a gigabit Ethernet connection, with a concurrency level of 1000. Under these conditions we found that Apache spent roughly 25% of its CPU time in kernel space. We ran the benchmark for 500 thousand requests[2]. This number was intentionally kept low so as to loosely include JIFL's JIT overhead.

Figures 11 and 12 present the Apache throughput and response time results when running with different types of kernel instrumentation. Results are normalized by the performance with an uninstrumented kernel. The trends appear to be similar in both graphs. JIFL incurs a performance cost of approximately 2% when running with no instrumentation due to JIT overhead and cost of executing from the code cache. As noted earlier, this result is included to illustrate the cost of the JIFL framework—if no instrumentation is wanted JIFL would be disabled, incurring no overhead. The additional cost of applying either system call monitoring or call tracing is marginal (5%), while basic block block counting instrumentation adds a 17% penalty. For system call monitoring, Kprobes outperforms JIFL because Kprobes has no constant overhead. However, with call tracing and basic block counting, Kprobes incurs a one order of magnitude degradation in performance. We believe this level of degradation would be unacceptable for most production systems.

Finally, in Figure 13 we show the effect of varying the amount of code cache and heap memory available to JIFL, on Apache's throughput. JIFL performs consistently until it has less than 3 MB per processor. Subsequently, the cost of performing code cache flushes is reasonable until 2 MB.

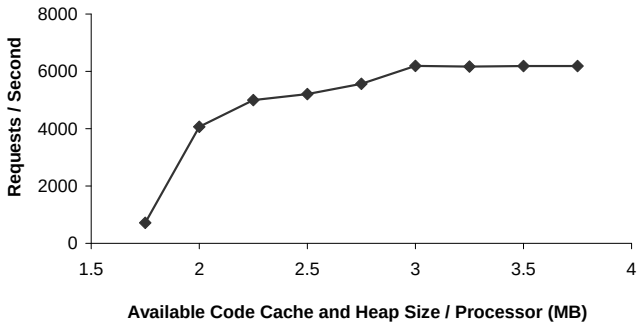

Figure 13: Effects of varying available JIFL memory on ApacheBench2 throughput.

5. EXAMPLE PLUGINS

Instrumenting system calls with JIFL is straightforward, as illustrated by the examples in this section. Users simply create a C source file that implements the `plugin_init`, `plugin_exit`, `plugin_start` and `plugin_stop` functions and relevant callbacks. In `plugin_init`, the plugin specifies callbacks for "interesting" events, such as discovering a new basic block. Typically the callback analyzes the basic block it is passed to determine whether or where to apply instrumentation. By appropriately defining these functions, instrumentation can be placed before or after instructions, basic blocks, routines, or system calls.

In addition, the JIFL build system takes care of compiling the code into a kernel module to simplify development. All code needed to interface with the user space startup tool is included in the `jifl.h` header file, allowing the user to focus on their plugin rather than mundane details.

What follows is a number of examples illustrating how easy it is to use the JIFL API to build plugins. These plugins also show useful applications of fine-grained operating system instrumentation. For clarity, we present the non-SMP versions of the plugins. When writing instrumentation for an SMP kernel, all global data structures must be privatized.

5.1 Call Tracing

Call tracing is a popular method for debugging, profiling and coverage testing. The JIFL API allows construction of a call tracing plugin. Example code to accomplish this is listed in Figure 14. The function `routine_inst()` is invoked when a new routine is encountered by the JIT; here, every routine is unconditionally instrumented with `store_call()` which stores the address of the routine into a queue. At the exit of the system call, the plugin looks up the names of the functions starting at the saved addresses, and prints them.

[2]Unfortunately, we had to reduce this number to 10 thousand when running the Kprobes basic block counting experiment to prevent ApacheBench from timing out.

```
void store_call(jifl_queue_t *queue,
        routine_t* routine) {
    jifl_enqueue(queue, routine);
}

// Called for every newly discovered routine
void routine_inst(routine_t *routine, void *arg) {

    // Current system call number is passed
    // to us by "arg".
    int syscall = (int) arg;

    routine_insert_call(routine, store_call,
            ARG_VOID_PTR, &queues[syscall],
            ARG_VOID_PTR, routine,
            ARG_END);
}

// Invoked upon syscall exit (not performed when
// benchmarking)
void syscall_exit(int return_arg, void* arg) {

    // Current system call number is passed
    // to us by "arg".
    int syscall = (int) arg;
    routine_t* r;

    printk("call trace for process %d\n",
        current->pid);
    while (jifl_dequeue(&queues[syscall], &r)) {
        char name[MAX_NAME_SIZE];
        routine_name(r, name, MAX_NAME_SIZE);
        printk("%s\n", name);
    }
}
```

Figure 14: Plugin code to print out a call trace

5.2 Checking Branch Hints

Branch hint prefixes are commonly used throughout the Linux source code to improve performance when the likely direction of a branch is known to the programmer. In addition, a compiler may try to infer the probable direction of a branch and insert a hint. However, incorrectly hinted branches can lead to performance degradation. Figure 15 presents plugin code that monitors the outcome of hinted branches on a per system call level. The code can be used to monitor the accuracy of branch hints, within the context of each system call. It can be easily extended to monitor individual branches.

In Figure 15, the basic block level callback analyzes each basic block to determine whether it ends with a hinted branch. If so, it inserts two calls to the `increment` instrumentation function specifying the address of two counters based on the branch hint direction. The first call (to `ins_insert_bt_call`) inserts the instrumentation in the branch-taken case. JIFL inserts branch-taken instrumentation by modifying the target of the branch to point to the instrumentation, which it places at the end of the basic block preceding a jump to the actual target. The second call to `ins_insert_bnt_call` simply tells JIFL to insert the instrumentation after the branch.

This type of instrumentation is difficult with probe-based instrumentation because there is no single instruction where a programmer can insert a probe to directly determine the outcome of a desired branch. Instead, he must insert probes at both the branch target and fall-through instructions, which might also be reachable via a different code sequence.

We tested this plugin while running our Apache web server benchmark. The plugin detected 33 system calls which executed hinted branches, and instrumented a total of 186

```
long long correct[NR_syscalls];
long long incorrect[NR_syscalls];

void increment(long long *counter_ptr) {
    (*counter_ptr)++;
}

// Called for every newly discovered BB
void bb_inst(bb_t *bb, void *arg) {

    // Current system call number is passed
    // to us by "arg".
    int syscall = (int) arg;

    // Branches can only be located at
    // the end of a basic block.
    ins_t *last = ins_last(bb);

    if (ins_type(last) != ins_jcc ||
            !ins_branch_hint(last))
        return;

    // The instrumentation depends on the
    // hint direction
    if (ins_bnt_prefix(last)) {
        // Insert branch-taken instrumentation
        ins_insert_bt_call(last, increment,
                ARG_VOID_PTR, &incorrect[syscall],
                ARG_END);

        // Do the same for branch-not-taken
        ins_insert_bnt_call(last, increment,
                ARG_VOID_PTR, &correct[syscall],
                ARG_END);
    } else {
        // Insert branch-taken instrumentation
        ins_insert_bt_call(last, increment,
                ARG_VOID_PTR, &correct[syscall],
                ARG_END);

        // Do the same for branch-not-taken
        ins_insert_bnt_call(last, increment,
                ARG_VOID_PTR, &incorrect[syscall],
                ARG_END);
    }
}
```

Figure 15: Example plugin for verifying branch hint correctness. Note: the code for `plugin_stop` to print out the correct/incorrect hint counts is omitted.

hinted branches. We found that out of the 33 system calls, 10 had misprediction rates that were greater than 50%. Of these, `open`, `close`, `waitpid`, `stat64`, and `lstat64` performed particularly poorly with misprediction rates ranging from 75-99%, while accounting for over 30% of all hinted branches executed.

We used a second plugin to examine individual hinted branches within the context of the five poorly performing system calls. We were able to identify the four branches with the greatest contribution to the high misprediction rate. Using `addr2line`, we were able to map the address of these branches to their locations within the kernel source code. Interestingly, we found that all of these branches were hinted automatically by the compiler, and not by a programmer. We can override gcc's incorrect decision by specifying our own hint. Because JIFL can specify different instrumentation per system call, we were also able to observe the effect of the current system call context on the misprediction rate of individual branches. For example, we found that a branch in the `__link_path_walk()` function, would always be predicted incorrectly when executing the `stat64` system call, but correctly otherwise. Such branches should

```
      mov           counter, %eax
1:    mov           %eax, %edx
      add           $0x1, %edx
      lock cmpxchg  %edx, counter
      jne           1
```

Figure 16: Atomic increment assembly code

```
long long contention_count;

void increment(long long *counter_ptr) {
    (*counter_ptr)++;
}

// Called for every newly discovered BB
void bb_inst(bb_t *bb, void *arg) {
    ins_t *last = ins_last(bb);
    ins_t *second_last = ins_prev(last);

    if (ins_type(last) == ins_jne &&
        ins_type(second_last) == ins_cmpxchg &&
        ins_lock_prefix(second_last))
    {
        // Insert branch-taken instrumentation
        ins_insert_bt_call(last, increment,
                ARG_VOID_PTR, &contention_count,
                ARG_END);
    }
}
```

Figure 17: Example of instrumenting the code around a `cmpxchg` instruction

obviously not be hinted as they would benefit from a global hardware branch predictor.

5.3 Monitoring Lock Contention

Monitoring lock contention is an often useful but difficult task. In this section we present a plugin for monitoring contention of atomic counter increments in the Linux kernel. Figure 16 presents an example of x86 atomic fetch-and-increment assembly code that could be used to implement a ticket lock.[3] The code works by reading the current value of the counter, incrementing it in a register, then attempting an atomic exchange of the new counter value with the old counter value held in memory. If the exchange fails, then the new counter value is obtained from memory (where it must have been modified by another processor) and the process repeats, with the new value incremented and an exchange attempted again.

Counting the number of times the atomic exchange fails can give a good indication of contention. In this case, we simply need to count the number of times that the jne branch jumps backwards. Figure 17 presents a plugin that searches for the atomic increment code and counts the number of times that this branch is taken.

The resulting code after instrumentation is presented in Figure 18. Note that the JIT compiler inlined the instrumentation function and that the `contention_count` address has been propagated into the code. The resulting low overhead instrumentation is especially important for contention monitoring where observation can often alter the original behavior. This is likely not possible with Kprobes due to its high monitoring overhead.

[3]This type of code sequence occurs in the Linux kernel.

```
      mov           counter, %eax
1:    mov           %eax, %edx
      add           $0x1, %edx
      lock cmpxchg  %edx, counter
      jne           2
      jmp           exit
2:    add           $1, 0(contention_count)
      adc           $0, 4(contention_count)
      jmp           1
exit:
```

Figure 18: Result of instrumentation

6. RELATED WORK

6.1 Pin

Pin [15] is a popular user space JIT instrumentation tool from Intel. Pin injects itself into the address space of an executing application, enabling it to take control. Thereafter, Pin JIT compiles and applies instrumentation to the application. Much of JIFL's internals and its plugin API are modelled after Pin.

Running in user space allows Pin to optimize aggressively. For example, Pin instruments at a trace-level granularity to improve opportunities for register re-allocation and reduce JIT overhead, and uses function cloning to reduce the cost of return instructions. Unfortunately, these optimizations require too much memory for JIFL to use in the kernel.

6.2 KernInst

KernInst [25] is a dynamic instrumentation framework designed for debugging, profiling, and application tuning. KernInst was the first to implement probe-based dynamic instrumentation in the kernel. Because it targeted the UltraSparc RISC architecture, KernInst was able to safely implement probes with branch instructions. Although their paper only evaluates the UltraSparc implementation, Tamches and Miller proposed trap instructions for redirecting control on x86. The current code release includes an x86 implementation which uses this trap-based strategy.

Like JIFL, KernInst applied register (though not condition code) liveness analysis to reduce the number of state saving instructions. However, unlike JIFL, KernInst cannot insert instructions in existing code and therefore cannot inline instrumentation to reduce overhead.

6.3 Kprobes

Kprobes [20] also uses the trap instruction methodology, suggested by KernInst, for implementing probes on the x86 architecture. The project has been embraced by the open source community and is now present in the main Linux source tree.

Because execution is redirected to instrumentation routines by means of a trap and hash table lookup, instrumentation is heavyweight. To alleviate this, a patch called Djprobes is currently under development, which allows overwriting some addresses with a 5-byte jump instruction, enabling direct jumps to instrumentation code. However, there are several complications surrounding preemptive kernels, SMP support, and safety, which at the time of writing have prevented Djprobes from being merged upstream. Kprobes cannot use register liveness analysis (unlike KernInst and JIFL) to reduce the cost of saving and restoring processor state since the trap instruction has only one target, although this limitation will not apply to Djprobes.

6.4 GILK

The GILK project [21] attempts to use static analysis of OS code to determine locations that could be safely probed with branch instructions on the x86 architecture. While this approach is often accurate, it is never safe in the presence of indirect instructions. Further, it cannot be used to overcome the race conditions that arise from using branch instructions on a variable-length ISA in a preemptable kernel.

6.5 DTrace

DTrace [7] is an instrumentation framework for the Solaris operating system, designed for use with production systems. DTrace makes it easy to monitor system resources, allowing system administrators to quickly identify the causes of system sluggishness, or to examine the otherwise unattainable system resources used by software (e.g., the number of I/O requests per second). DTrace is also able to dynamically instrument both user-level and kernel-level code.

DTrace instrumentation works by inserting jump-based trampolines on fixed-length RISC architectures, but uses the same trap mechanism as KernInst or Kprobes on variable-length ISAs. Anecdotally, DTrace runs quite fast on Sparc architectures, however, we expect it would suffer similar overheads as KProbes on x86. The jump-based implementation prevents instrumentation inlining, which can offer significant speed advantages as demonstrated by JIFL.

Because DTrace is intended for use in production systems, it guarantees that user instrumentation cannot cause additional system failures. User-supplied instrumentation code is expressed in a C-like high-level control language which enforces safety. While JIFL plugins are currently written in C, it would be straightforward to add a scripting language allowing verification in the same manner as DTrace.

6.6 SystemTap

The SystemTap project [22] is a joint effort by Red Hat, IBM, Intel, and Hitachi to add an easy to use front end to Kprobes with functionality similar to DTrace, including the use of a scripting language. Instrumentation scripts can make symbolic references to the kernel, user programs, or included libraries (called "tapsets"). Scripts are compiled into a kernel module and loaded to start the probes and handlers. Although a stable version of SystemTap is not yet released, some early adopters have found it useful. SystemTap currently uses Kprobes for low-level instrumentation.

6.7 K42

K42, an object oriented research operating system [2], offers two instrumentation strategies. The first strategy works by modifying the target address of object invocations, allowing coarse-grained instrumentation at method invocations. The second strategy allows fine-grained instrumentation via K42's hot-swapping capability. With this approach, the user directly modifies the relevant source code (for example, by adding instrumentation), recompiles the module, and hot-swaps it in place of the old module with no service disruption. K42's instrumentation facilities offer high performance, as instrumentation can be compiled into the code. However, JIFL is more flexible by allowing decisions about where to instrument to happen at run-time, and furthermore, allows the user to instrument *cross-cutting* concerns where the item of interest occurs in many places (for example all cmpxchg instructions); finding and manually instrumenting all such code in the kernel would be prohibitively time consuming.

6.8 QEMU

QEMU[4] is a multi-architecture full system emulator. It uses dynamic binary translation, and is capable of translating from one architecture to another. When the host system is the same architecture as the guest system (the machine being emulated), QEMU uses a virtualization strategy which allows most emulated code to run directly on the hardware in a manner extremely similar to JIFL, with a demand populated code cache, basic block linking, and more. QEMU is a JIT for efficient emulation of a contained guest operating system, while JIFL is a JIT for efficient instrumentation of the host operating system. While it is entirely possible that instrumentation capabilities could be added to QEMU, they are not currently available, and furthermore, are not intended for the host operating system. Thus, while QEMU and JIFL are quite similar, they fulfill different needs.

7. FUTURE WORK

There are two avenues of pursuit for improving JIFL. First, the implementation discussed in this paper is not preemption-safe. Preemption poses a number of concerns when running on SMP kernels as threads executing within a private code cache can be migrated to a different processor without switching code caches. This presents a problem for both the private dispatchers and for any per processor inlined user instrumentation. It is possible to correct the issue by modifying the dispatcher, and by forcing users to write preemption safe instrumentation; however, doing so may inconvenience the user and may potentially degrade performance. Another option would be to disable preemption while executing within the code cache, thus maintaining current performance and usability, while still allowing JIFL to be attached to a live preemptable kernel. Unfortunately, JIFL would not be able to observe the true behaviour of the uninstrumented kernel. Second, JIFL can be extended to instrument kernel threads, which do not make use of system calls. Gaining control of such threads can be achieved by modifying their program counter while they sleep, or with a one-time executed dynamic instrumentation probe.

Finally, we would like to quantify the performance benefits that can be obtained by using the results gathered with our various plugins. For example, it would be interesting to see the performance impact of fixing the incorrect branch hints detected by our example instrumentation.

8. CONCLUSION

JIT instrumentation provides an efficient, fine-grained kernel instrumentation framework for the Intel x86 architecture. We have demonstrated the viability of this approach with a prototype for the Linux kernel, called JIFL. Experimental results show JIFL outperforms the leading alternative framework for Linux (Kprobes) by nearly a factor of 50 when applied to the medium-grained task of extracting commonly used debugging information (call traces). For fine-grained tasks, JIFL outperforms Kprobes by one to two orders of magnitude. The simplicity and versatility of JIFL, as illustrated by our example plugins, make it a powerful tool for kernel analysis.

9. ACKNOWLEDGMENTS

We would like to thank the National Science and Research Council of Canada (NSERC) and the University of Toronto for supporting this research, the EuroSys review committee for their detailed and constructive comments, and the extensive comments and suggestions by Davor Capalija and Tomasz Czajkowski which were instrumental in improving our paper.

10. REFERENCES

[1] B. Alpern, D. Attanasio, J. Barton, M. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. 39(1), 2000.

[2] J. Appavoo, K. Hui, C. A. N. Soules, R. W. W. D. D. Silva, O. Krieger, M. Auslander, D. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[4] F. Bellard. QEMU: a fast and portable dynamic translator. In *Proc. of USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] L. Benini, A. Bogliolo, S. Cavallucci, and B. Ricco. Monitoring system activity for OS-directed dynamic power management. In *Proc. of Intl. Symp. on Low Power Electronics and Design*, pages 185–190, Aug. 1998.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st Intl. Symposium on Code Generation and Optimization (CGO-03)*, Mar. 2003.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. of the USENIX 2004 Annual Technical Conference*, pages 15–28, Jun. 2004.

[8] C. Cifuentes, B. Lewis, and D. Ungar. Walkabout - a retargetable dynamic binary translation framework. In *Proc. of the Fourth Workshop on Binary Translation*, Sep. 2002.

[9] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of the 6th Symp. on Operating Systems Design and Implementation (OSDI'04)*, pages 231–244, Dec. 2004.

[10] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation (OSDI'02)*, pages 211–224, Dec. 2002.

[11] R. Flower, C. Luk, R. Muth, H. Patil, J. Shakshober, R. Cohn, and G. Lowney. Kernel optimizations and prefetch with the Spike executable optimizer. In *Proc. of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[12] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proc. of the 1992 USENIX Winter Technical Conference*, pages 125–138, Jan. 1992.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. of the 2005 Annual USENIX Technical Conference*, pages 1–15, Apr. 2005.

[14] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. In *PLDI '95: Proc. of the ACM SIGPLAN 1995 Conf. on Programming language design and implementation*, pages 291–300, 1995.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN Conf. on Programming language design and implementation*, pages 190–200, 2005.

[16] L. McVoy and C. Staelin. LMbench: Portable tools for performance analysis. In *Proc. of the 1996 USENIX Technical Conference*, pages 279–295, Jan. 1996.

[17] J. C. Mogul. Emergent (mis)behavior vs. complex systems. In *Proc. of EuroSys 2006*, pages 293–304, Apr. 2006.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):1–23, Oct. 2003.

[19] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, Apr. 2001.

[20] P. S. Panchamukhi. Kernel debugging with kprobes: Insert printk's into the Linux kernel on the fly, Aug 2004. http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnx%w07kprobe.

[21] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. GILK: A dynamic instrumentation tool for the Linux kernel. In *TOOLS '02: Proc. of the 12th Intl. Conf. on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 220–226, 2002.

[22] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proc. of the 2005 Ottawa Linux Symposium*, pages 49–64, Jul. 2005.

[23] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. of the 2001 Workshop on Binary Rewriting (WBT-2001)*, Sep. 2001.

[24] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools (with retrospective). In K. S. McKinley, editor, *Best of PLDI*, pages 528–539. ACM, 1994.

[25] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 117–130, Feb. 1999.

[26] K. Yaghmour and M. Dagenais. The Linux Trace Toolkit. Linux Journal, Issue no. 73, May 2000. http://www.linuxjournal.com/article/3829.