Binary Rewriting of an Operating System Kernel *

Mohan Rajagopalan,‡ Somu Perianayagam, HaiFeng He, Gregory Andrews, Saumya Debray

Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

Email: mohan.rajagopalan@intel.com; {somu, hehf, greg, debray}@cs.arizona.edu

1. Introduction

This paper deals with some of the issues that arise in the context of binary rewriting and instrumentation of an operating system kernel. OS kernels are very different from ordinary application code in many ways, e.g., they contain a significant amount of hand-written assembly code. Binary rewriting is an attractive approach for processing OS kernel code for several reasons, e.g., it provides a uniform way to handle heterogeneity in code due to a combination of source code, assembly code and legacy code such as in device drivers. However, because of the many differences between ordinary application code and OS kernel code, binary rewriting techniques that work for application code do not always carry over directly to kernel code. This paper describes some of the issues that arise in this context, and the approaches we have taken to address them. A key goal when developing our system was to deal in a systematic manner with the various peculiarities seen in low-level systems code, and reason about the safety and correctness of code transformations, without requiring significant deviations from the regular developmental path. For example, a precondition we assumed was that no compiler or linker modifications should be required to use it and the tool should be able to process kernel binaries in the same way as it does ordinary applications.

We have implemented a prototype kernel binary rewriter as an extension to the PLTO binary rewriting toolkit [14]. PLTO takes as input a relocatable binary that it manipulates in various ways, e.g., to insert instrumentation code or to apply various optimizing transformations using optional execution profiles for guidance. It then updates code addresses as necessary, using relocation information to distinguish addresses from non-address values, and finally writes the resulting program out as an executable. PLTO currently supports the collection of several different kinds of execution profiles: basic block counts, edge counts, value profiles (especially important for resolving indirect function call targets), call-stack profiles, as well as profiles based on hardware performance counters, e.g., CPU cycles and i-cache misses. Our prototype has also been used to perform a number of program analyses and optimizations, including function inlining, guarded inlining (for indirect function calls—the inlined code is guarded by a test of the call target), constant propagation (for code specialization), dead and unreachable code elimination (especially important for code compaction), and profile-guided code layout. We have also used our system to carry out specialization and compaction of the Linux kernel. Some of this work is described in a companion paper [11].

2. Technical Challenges

This section discusses some of the issues arising in binary rewriting and instrumentation of OS kernels that are usually not encountered for ordinary application programs. Where appropriate, we quantify our observations using characteristics of a minimally-configured Linux 2.4.31 kernel (no module support; drivers for network, video, block devices, keyboard, and mouse; ext2 and ext3 filesystems; and TCP/IP and UDP stack only), compiled on an Intel x86 platform using the *gcc* compiler version 3.3.3 at optimization level `-O2`.

2.1 Disassembly

The different characteristics of kernel and application binaries means that a straightforward application of conventional disassembly algorithms can fail to correctly disassemble large parts of the kernel. A significant problem during disassembly is that of data embedded in the text section, which can confuse the disassembly process. In the Linux kernel, binary data is embedded in the instruction stream in two distinct instances:

1. Data areas that are not part of instruction stream but are located in the text section. For example, in Linux version 2.4, the page tables are placed in the text section. We identify such data areas by their associated symbols. A list of such symbols is provided as input to PLTO, which then skips over the corresponding memory areas during disassembly. There about 19 such symbols in the Linux kernel.[1]
2. Data embedded in the instruction stream that are not part of any instruction, but which may be used during execution. A typical example of this is the `ud2` instruction used in the kernel. The `ud2` instruction, which specifies an "undefined instruction," raises an invalid opcode exception and is used to raise a panic and halt the kernel in case of a bug. Typically, the source code line number and

* This work was supported in part by NSF Grants EIA-0080123, CCR-0113633, and CNS-0410918.

‡ Current address: Programming Systems Lab, Intel Research Corp., Santa Clara, CA 94054.

[1] Here and elsewhere in the paper, we use the phrase "the Linux kernel" to refer to the version of the Linux kernel mentioned above, configured and compiled as described.

	Program	Static counts			Dynamic counts ($\times 10^6$)		
		Instructions ($nIns_s$)	*Indirect calls* ($Icalls_s$)	$Icalls_s/nIns_s$ (%)	*Instructions* ($nIns_d$)	*Indirect calls* ($Icalls_d$)	$Icalls_d/nIns_d$ (%)
SPECint-2000	*bzip2*	5,647	0	0.00	41707.62	0.00	0.00
	crafty	42,287	0	0.00	34616.64	0.00	0.00
	vpr	16,667	0	0.00	5581.47	0.00	0.00
	gap	112,267	1296	1.15	7163.14	109.98	1.54
	gcc	172,957	84	0.05	1071.28	0.13	0.01
	gzip	6,448	1	0.02	33542.75	0.00	0.00
	parser	21,881	0	0.00	9124.77	0.00	0.00
	perlbmk	52,928	23	0.04	15479.29	387.67	2.50
	twolf	44,581	0	0.00	10229.97	0.00	0.00
	vortex	122,060	10	0.01	14993.09	0.03	0.00
	Linux	349,762	1368	0.39	766.16	1.07	0.14

Figure 1. Indirect function call characteristics

a pointer to the file name are stored in the six bytes following each ud2 instruction. The ud2 handler prints out this information before halting the kernel. Such usage is very kernel-specific: the references to the data bytes following the ud2 instruction are not obvious in the code containing the instruction, but instead occur (indirectly, through the address from which the exception was raised) in the ud2 exception handler. About 6% of the functions in the Linux kernel contain these instructions. A straightforward disassembly of the kernel would very likely treat the data bytes following the ud2 instructions as unreachable; however, eliminating them could potentially change the behavior of the kernel.

A crude user-level analog of this is with jump tables embedded in the text section in position-independent code. A key difference between the two situations is that references to such jump tables from within the code are relatively direct and not very difficult to identify, while references to the ud2 instructions are indirect and significantly harder to identify without specific high-level knowledge of how they are used.

Overall, we found about 21 Kbytes of data embedded within the code stream in the code sections in the Linux kernel, out of a total of 1.16 Mbytes, i.e., about 1.8%.

To address this problem, we use symbol table information to guide the disassembly, which proceeds in three phases: First, symbol information is used to identify well defined code regions such as functions, which are disassembled using the standard recursive disassembly algorithm. The second phase uses the symbol table to try and identify "stubs," which are code regions that do not appear to be conventional functions. Typical examples of such code are hand-written assembly routines, kernel entry point routines, interrupt handlers, etc.[2] The final phase of disassembly uses relocation information to discover regions of code that have been missed by the previous steps. The basic idea is to exploit relocation information that is available in the binary. In this phase, all the relocation entries are checked to see if they point to a disassemble-able region of code. This is done by checking if the source address for the relocation, the address that the relocation points to, is within the text section. If the source address is within the text section then this is treated as a potential jump target and becomes a target for recursive disassembly. This step is effective in identifying almost all the regions that were missed out in the earlier phases if they were reachable only as targets of indirect control transfers. Our results indicate that this algorithm is able to disassemble approximately 94% of the executable sections. The remaining 6% includes data blocks (several 4K pages) and padding NOP instructions, in addition to the executable code that cannot currently be disassembled. Portions of the text section that cannot be disassembled are treated as data and as such, are reinserted into the kernel executable when it is reassembled; however, any code pointers in such undisassembled code/data are identified as such, and updated correctly, using the associated relocation information. For example, jump tables in the code section are handled in this way.

[2] In the Linux kernel, stubs appear as text section symbols of type `NOTYPE`. However, not all `NOTYPE` symbols in the text section correspond to stubs: there are a handful of such symbols that should not be disassembled as code, because they point either to data or to special regions in the text section. Since it is not possible to algorithmically identify such regions, we allow the user to specify such embedded data symbols via a table that indicates the name, location, and the size of the symbol. In our current implementation, this table contains fewer than 20 entries and includes the special purpose pages that are used to initialize the memory manager.

2.2 Control Flow Analysis

After disassembly, the resulting instruction sequence is organized into a inter-procedural control flow graph. Unfortunately, control flow analysis of operating system kernels is complicated by a number of factors, such as the presence of hand-written assembly code and its interaction with indirect function calls; code layout to segregate infrequently executed code in order to avoid cache pollution; and exception handling. This section discusses some of these issues.

2.2.1 Indirect Function Calls

Control flow analysis in operating system kernels is complicated by the interaction of two separate problems. First, there is a significant amount of hand-written assembly code in the kernel (Section 2.3 discusses some of the challenges this causes, and the approach we take to handle them). Second, operating system kernels often make extensive use of indirect function calls in order to enhance maintainability and extensibility. This is a problem because static analyses are generally quite conservative in their treatment of indirect

```
static inline void __down_read(struct rw_semaphore *sem)
{
        __asm__ __volatile__(
                "# beginning down_read\n\t"
LOCK_PREFIX     "  incl      (%%eax)\n\t" /* adds 0x00000001, returns the old value */
                "  js        2f\n\t"      /* jump if we weren't granted the lock */
                "1:\n\t"
                LOCK_SECTION_START("")  /*busy wait code placed in a separate subsection */
                "2:\n\t"
                "  pushl     %%ecx\n\t"
                "  pushl     %%edx\n\t"
                "  call      rwsem_down_read_failed\n\t"
                "  popl      %%edx\n\t"
                "  popl      %%ecx\n\t"
                "  jmp       1b\n"
                LOCK_SECTION_END
                "# ending down_read\n\t"
                : "=m"(sem->count)
                : "a"(sem), "m"(sem->count)
                : "memory", "cc");
}
```

Figure 2. Example of a kernel function whose code is spread over multiple subsections

function calls.[3] Each of these problems—hand-written assembly and indirect function calls—is nontrivial in its own right, and the situation is exacerbated further by the fact that they interact: the hand-written assembly code in an operating system kernels may itself contain indirect function calls, and identifying those targets requires pointer alias analysis of the assembly code.

Figure 1 shows the static and dynamic indirect function call characteristics of the SPECint-2000 benchmark suite compared to that of the Linux kernel.[4] The static counts indicate that, with the notable exception of the *gap* program, most of the programs in the SPECint-2000 suite contain relatively few indirect function calls (applications written in an object-oriented style, in a language such as C++ or Java, would likely have a higher number of indirect function calls; however, given that operating system kernels are written mostly in C and typically do not use object-orientation, this is not particularly relevant). The dynamic instruction counts indicate that the runtime behavior of the Linux kernel is not dramatically different than the behavior of the SPECint-2000 programs. Most of the programs in this particular set of applications execute relatively few indirect function calls, with *gap* and *perlbmk* being notable exceptions. The Linux kernel executes a fairly large number of indirect calls, more than most of the SPEC benchmarks considered but well below *gap* and *perlbmk* and the number of function calls it executes, relative to its total dynamic instruction count, lies well within the range of values for the SPECint-2000 suite. It is possible that this is because the indirect function call behavior of the kernel is not substantially different from that of other software; it is also possible that the particular set of benchmarks we used for profiling the kernel (the MiBench suite) simply did not exercize the kernel code very much. Since we are interested primarily in static optimization and transformation of the kernel, however, it is the static counts that are most relevant for our purposes, and here it is clear that the kernel code contains significantly more indirect function calls than most of the application programs we examined.

[3] In general, identifying the possible targets of indirect function calls is equivalent to pointer alias analysis, which is a hard problem both theoretically and in practice.

[4] The SPECint-2000 programs were compiled with *gcc* version 3.3.3 at optimization level `-O2`; their dynamic instruction counts were obtained using their profiling inputs, while those for the Linux kernel were obtained using the MiBench suite of applications [7].

2.2.2 Hand-written Assembly Code

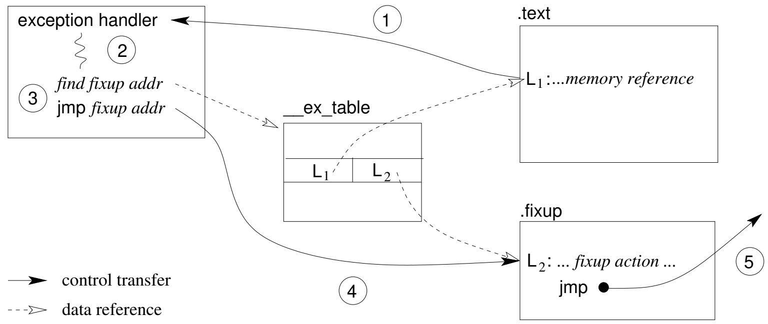
As an indication of the extent and effects of hand-written assembly code in the Linux kernel, we found that of the 5,133 functions in the kernel, 89 functions did not have the standard function prologue and 34 did not have the standard epilogue, suggesting that the code was not compiler-generated, i.e., was hand-written assembler; by contrast, in the application programs we examined, all functions had standard prologues and epilogues. The reason this is significant is that the absence of standard prologues and/or epilogues can affect the precision of analyses that examine the stack behavior of functions.

2.2.3 Implicit Entry Points

An important problem in dealing with control flow in operating system kernels is that not all entry points into the kernel, and control flow within the kernel, are explicit. There are implicit entry points such as system calls and interrupt handlers, as well as implicit control flow arising from interrupts, that have to be taken into account in order to guarantee soundness. Our implementation uses the system call table to identify system call handlers and mark them as potential entry points into the kernel control flow graph (interrupt handlers do not need to be treated specially, but are found in the course of ordinary control flow reachability analysis).

2.2.4 Non-contiguous Code Layout of Functions

A kernel developer can exploit the use of hand-coded assembly to lay out specific parts of the same kernel function in different parts of memory, so as to seperate common execution paths from less frequently taken execution paths. The



Key:

① A memory exception at L_1 causes control to branch to the exception handler.

② Exception handling code.

③ Exception handler searches `__ex_table` with the address L_1, where the exception occurred, to find the associated fixup code address L_2.

④ Control branches from the exception handler to the fixup code.

⑤ Control branches from the fixup code to some appropriate code address.

Figure 3. Control flow during the handling of exceptions in the Linux kernel

infrequently executed path is placed in a seperate subsection within the text section. There is one subsection created for all the functions belonging to the same module, and all their infrequently executed code is placed in that subsection. This is illustrated in Figure 2, where the code fragment between `LOCK_SECTION_START` and `LOCK_SECTION_END` are placed in a different subsection within the text section. This has the effect of realizing the "procedure splitting" optimization described by Pettis and Hansen [12]. However, this can lead to imprecision during control flow analysis: since the different subsections have symbols associated with them, a naive disassembler may infer that the code in these subsections belong to distinct functions. The resulting code appears to have two distinct functions, with control jumping from the middle of one into the middle of the other. This kind of interprocedural control flow—which actually occurs in some parts of the kernel code—is not easily handled in many of our program analyses, and can lead to a loss in precision. We address this problem by making a post-pass over the control flow graph to identify functions that have been split in this manner, and merge the code from the two distinct functions into a single function. We found 100 subsections in the Linux kernel, with each subsection having, on average, code for about 9 functions.

2.2.5 Control Flow Issues in Exception Handling

In order to identify all reachable code in the kernel, it is not enough to consider ordinary control transfers, which are explicit in the code: we also have to take into account control transfers that are implicit in the exception handling mechanisms of the kernel. For this, we examine the exception table in the kernel. Locations in the kernel where an exception could be generated are known when the kernel is built. For example, the kernel code that copies data to/from user space is known as a potential source for a page fault exception. The Linux kernel contains an exception table, `__ex_table`, that specifies, for each such location, the code that is to be executed after handling an exception. Additionally, a special section, `.fixup`, contains snippets of code that carry out the actual control transfer from the exception handlers to the appropriate destination locations. The flow of control when handling an exception is shown in Figure 3: after the exception handler deals with an exception from an address L_1, it searches `__ex_table` with L_1 as the key, finds the associated address L_2 of the corresponding fixup code, and jumps to L_2. This then carries out some fixup actions, e.g., setting flags or error values, and eventually jumps to some appropriate location in the text area. For example, when a page fault occurs, the reason could be either that the address being referenced lies in a page that is not in memory, or that it is an illegal address. In the former case, the exception handler loads the referenced page into memory and the fixup code branches back to the instruction that raised the exception, causing it to be re-executed. In the latter case, the fixup code branches to an error routine.

The key point to note here is that the control flow path from L_1 to L_2 is not explicit in the code, but is implicit in `__ex_table`. It is necessary to take such implicit execution paths into account for code compaction to ensure that we find all reachable code. We do this by examining the exception table and adding pseudo-control-flow edges to indicate such implicit control flow. For the example in Figure 3, we would add such an edge from L_1 to L_2. One implication of this is that any instruction that can raise an exception, i.e., which is referenced from the exception table, terminates a basic block.

Of the 108,611 control flow edges in the whole-program control flow graph of the Linux kernel, 698 edges were pseudo-control-flow edges resulting from the exception-handling mechanism described above.

2.3 Program Analysis

As mentioned above, operating system kernels contain a significant amount of hand-written assembly code. Since our goal is to apply optimizing transformations on the kernel

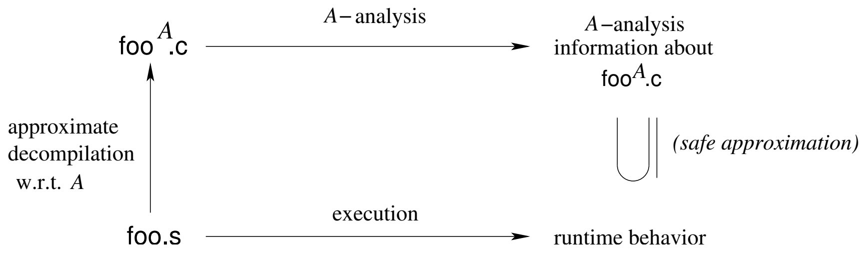


Figure 4. Using approximate decompilation for program analysis

code, we have to ensure that our analyses take all possible runtime behaviors of the code into account; in other words, we cannot sacrifice soundness. This makes program analysis problematic. On the one hand, dealing with hand-written assembly code in a source-level or intermediate-code-level analysis is messy and awkward because of the need to inject architecture-specific knowledge into the analysis—such as aliasing between registers (e.g., in the Intel x86 architecture, the register %al is an alias for the low byte of the register %eax) and idiosyncrasies of various machine instructions. On the other hand, if the analysis is implemented at the assembly code or machine code level, much of the semantic information present at the source level is lost—in particular, information about types and pointer aliasing—resulting in overly conservative analysis that loses a great deal of precision.

We deal with this problem using an approach we call "approximate decompilation," which maps hand-written assembly code back to C source files for analysis purposes. The idea, illustrated in Figure 4, is that given an assembly file foo.s and a program analysis A, we create a source file foo^{A}.c that has the property that an A-analysis of foo^{A}.c is a safe approximation of the behavior of foo.s, even though foo^{A}.c is not semantically equivalent to foo.s. For example, if A focuses on control flow analysis, then foo^{A}.c may elide those parts of foo.s that are irrelevant to control flow.

We have applied this approach to build a tool that automatically carries out approximate decompilation of Intel x86 assembly code files for use with a source-level pointer alias analysis technique called FA-analysis [9, 17, 18] to identify the possible targets of indirect function calls. This proceeds as follows.

2.3.1 Data

Global memory locations in the assembly code are mapped to global variables of type *int* in the generated C code. Memory locations accessed through the stack pointer register %esp are assumed to be on the stack. Variables at different locations within a function's stack frame are mapped to different local variables within the corresponding function in the generated C code. Since there is little type information available at the assembly level, these are declared to be of type *int* with 32-bit values. A memory object that spans a sequence of memory locations in the assembly code is mapped to an array of *int* in the generated C code. Before we start the actual pointer analysis, we scan the entire kernel source code and match memory objects to functions so that the source-level FA analysis can deal properly with function pointers in the assembly code.

Registers in the assembly code are also mapped to global variables of type *int* in the generated C code. For example, the 32-bit register %eax is mapped to a variable eax. Since we are only interested in capturing potential aliasing relationships between objects, and not necessarily the actual values computed, we map the 16-bit and 8-bit registers (which are aliases of parts of the 32-bit registers) into the appropriate 32-bit global. Thus, the 8-bit register %al and the 16-bit register %ax, which refer to the low 8 bits and the low 16 bits of the 32-bit register %eax respectively, are both mapped to the variable eax denoting the 32-bit register %eax.

2.3.2 Code

Functions in the assembly code are identified from symbol table information and mapped to functions in the generated C code.

Arguments to a function are identified via references to stack locations that are deeper in the stack than the function's own stack frame. In this manner, by examining the references to actual parameters in the body of a function, we can determine the number of arguments it takes, and thereby generate a function prototype in the C code.

The assembly instructions in the function body are processed as follows. System instructions, which manipulate only the hardware or data related to the hardware (examples of such instructions are LIDT – *Load interrupt descriptor table*, and INVLPG – *Invalidate TLB entry*), have no effect on pointer aliasing in the kernel code. For pointer alias analysis, therefore, we simply ignore these instructions. Since FA analysis is flow-insensitive and context-insensitive, instructions whose only effect is on intra-procedural control flow, such as conditional and unconditional branches, also have no effect on the analysis. Inter-procedural control flow cannot be ignored, however, since it induces aliasing between the actual parameters at the call site and the formal parameters at the callee. Our decompiler therefore ignores conditional and unconditional control flow instructions whose targets are within the same function, but translates inter-procedural control transfers.[5] A control transfer to a symbol S is translated as a function call if either the instruction is a call instruction, or if the target S is a function. Finally, instructions that move data and perform arithmetic and logic operations are translated to the corresponding operations in C. For example, a

[5] Note that in principle, the decompiler could have translated control transfer instructions into the appropriate control transfer statements in C, which would be ignored by the FA analysis. While this would have had the benefit of allowing us to use the approximate decompiler with flow-sensitive program analyses as well, we did not do this due to time constraints.

register load instruction, ‘mov $0, %eax,’ is translated to an assignment ‘eax = 0’.

3. Code Transformations

Idiosyncrasies of the kernel also affect the way we apply transformations to the code. There are two main considerations here. The first involves code that cannot be altered or moved because its behavior is closely tied to interactions with the underlying hardware, while the second involves interactions with exception handling. These are illustrated here with some examples.

The first example is of boot up code where apparently unnecessary instructions cannot be eliminated. In the code snippet shown below, the first number on each line is the address of the instruction on that line:

```
<startup_32>
 ...
 0xc0100036    mov  %eax, %cr0
 0xc0100039    jmp  0xc010003b
 0xc010003b    mov  $0xc0100042, %eax
 0xc0100040    jmp  *%eax
 0xc0100042    lss  0xc01001e5, %esp
 ...
```

This code snippet contains two jmp instructions, shown in bold, each of which jumps to the following instruction: the first of these jumps to the next instruction, whose address (0xc010003b) is specified as an absolute operand, while the second loads the address of the instruction after it (0xc0100042) into register %eax and then jumps indirectly through this register. Each of these jmp instruction therefore appears redundant. It turns out, however, that these instructions check whether turning on paging in the hardware worked, and cannot be optimized away. Furthermore, the page tables are located immediately after the hardware initialization. These tables need to be page-aligned, and any transformation to the initial boot up code could potentially violate this alignment requirement. Violations of such alignment requirements cause the kernel to hang during boot up time.

The exception-handling mechanism discussed in Section 2.2.5 (see also Figure 3) also imposes implicit constraints on code transformations. The most obvious of these is that any transformation that involves code duplication—for example, function inlining—must ensure that additional exception table entries and fixup code are added for each instruction in the duplicated region that can give rise to an exception (to get around this issue, our implementation currently carries out inlining of functions only if the function being inlined does not contain any instructions that can cause an exception, i.e., does not have any entries in the exception table pointing into its body).

Exception-causing code can have other effects as well. Consider the situation illustrated in Figure 5. The pop instruction in basic block B_1 can raise a page-fault exception. This causes control to branch to an exception handler which which as discussed in Section 2.2.5, loads the referenced page into memory and then jumps to a block of fixup code; in this case, the fixup code then transfers control back to the original instruction that raised the exception, and re-executes

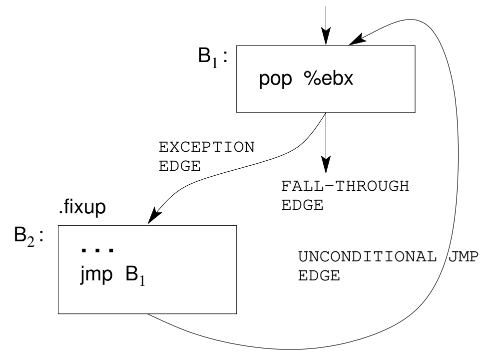


Figure 5. An example of analysis complications due to exception edges

it. The problem here is that when we consider the exit from basic block B_1, we cannot guarantee that the pop instruction in that block has been executed. One possible solution would be to propagate some of the instruction semantics to the control flow edges. For example, an stack analysis aimed at determining the height of the stack at different program points (this information is used to support a variety of other analyses, such as constant propagation and stack location liveness) would have to conclude that the pop instruction, which deallocates a word off the stack, has been executed if the fall-through edge out of block B_1 is taken, but is not executed if the exception edge $B_1 \rightarrow B_2$ is taken. While this would give correct results, such an approach is a departure from the standard treatment of control flow graphs, and has the effect of complicating the various dataflow analyses used. Our current implementation makes the simpler (but conservative) assumption that in situations where a basic block has an outgoing EXCEPTION edge, we cannot guarantee whether or not the last instruction in the block has been executed.

3.1 Instrumentation

Our system supports profiling of the kernel based on both software-managed counters (e.g., basic block and edge profiles) and hardware-managed counters (e.g., CPU cycles, cache misses). In order to obtain execution profiles, we need to know where to begin profiling as well as where to end profiling and write out the profile data. For ordinary applications, the well-defined entry and exit points serve as natural points for starting and ending profiling respectively. An OS kernel, however, has multiple entry and exit points, making it necessary to create a mechanism to begin and end profiling.

Our system uses a special (new) system call for this. One of its arguments determines whether it starts profiling or ends it and writes out the results. Another argument determines what kind of profiling is carried out (basic block counts, edge counts, or hardware-counter profiles). The code to be profiled is bracketed with calls to this system call (currently these calls are inserted manually, but in principle this step is easily automated).

While this infrastructure has been used to instrument the kernel to track different kinds of control and data flows, it is unable to instrument the initialization code that sets up interrupt and fault handlers at boot time. This is because the profiling data structures can not be accessed until page tables have been initialized. However, in practice this region of code is small and this does not lead to significant loss of information for later optimizations.

4. Discussion

While in the current implementation we have focused on Linux kernel binaries, in theory, PLTO can be extended to process binaries of other operating system kernels that are based on ELF format such as *BSD, MacOS, and Solaris. Inspection of the OpenBSD and FreeBSD kernel binaries suggests that the changes required to process them would be fairly straightforward. In general, the key challenge would be adapting PLTO to handle peculiarities native to each kernel. These peculiarities are most prominent when disassembling the instruction stream and recreating the final executable. The PLTO infrastructure provides hooks that can be used to inform it about layout requirements for code regions as well as regions that require special handling, for example, parts of the text sections that need to be treated as data. While we have not yet observed any new unusual instruction sequences such as the use of the `ud2` instruction in Linux, the same approach as before may be used handle them. Another minor modification that would be required would be that of updating the system call and interrupt handler tables depending on the underlying operating system. In general, it appears that calling conventions and higher level structure are almost the same and thus the higher level infrastructure, such as the different optimizers, can be used without any alterations.

5. Related Work

We are not aware of a great deal of work on binary rewriting and optimization/specialization of operating systems kernels. Flower *et al.* describe the use of Spike, a binary optimizer for the Compaq Alpha, to optimize the Unix kernel [6]. There are many high-level similarities between their work and ours, e.g., with regard to optimizations such as unreachable code elimination and profile-guided code layout. The differences between their system and ours are mainly at the low level, arising out of the fact that theirs was carried out on the Alpha architecture, which is a fixed-instruction-size RISC architecture while ours is on the Intel x86, a variable-instruction-size CISC architecture. The main impact of this difference is in the disassembly code. Chanet *et al.* describe a system for code compaction of the Linux kernel to reduce its memory footprint [3]. Unlike the work described here, their system relies on a modified compiler tool chain and requires special annotations (currently manually applied) to deal with hand-coded assembly.

Many researchers have investigated issues related to the instrumentation and profiling of operating system kernels [1, 10, 13, 15, 16]. To enhance flexibility and usability, most of this work has focused on dynamic instrumentation. However, implementing a dynamic instrumentation tool is a non-trivial task (especially on architectures, such as the Intel x86, that have variable-length instructions). Since our primary objective was to investigate profile-driven code transformations, we opted to avoid the complications associated with dynamic instrumentation, and chose instead to implement a simple static instrumentation tool.

There have been a number of successful efforts at building program analysers for bug detection that sacrifice soundness for pragmatic reasons [2, 4, 5]. Such analyses do not suit our needs because our goals are different and require that the analyses be sound.

There has been a great deal of research on code specialization (Jones *et al.* give a comprehensive discussion and bibliography [8]). Almost all of this work is in the context of application programs in high-level languages, and does not consider the issues that arise when dealing with an OS kernel.

References

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. Vandervoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4), November 1997.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, 2000.

[3] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide compaction and specialization of the Linux kernel. In *Proc. 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 95–104, June 2005.

[4] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proc. 19th. ACM Symposium on Operating Systems Principles*, pages 237–252, October 19–22 2003.

[5] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th ACM Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 1–16, 2000.

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

[7] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. MiBench: A free, commercially representative embedded benchmark suite. pages 3–14, December 2001.

[8] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[9] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for C programs with function pointers. *Automated Software Engineering*, 11(1):7–26, 2004.

[10] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. GILK: A dynamic instrumentation tool for the linux kernel. In *Proc. 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS)*, volume 2324 of *Lecture Notes in Computer Science*, pages 220–226. Springer, April 2002.

[11] S. Perinayagam, H. He, M. Rajagopalan, G. Andrews, and S. Debray. Profile-guided specialization of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications*, October 2006.

[12] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[13] V. Prasad, F. Ch. Eigler, J. Keniston, W. Cohen, M. Hunt, and B. Chen. Locating system problems using dynamic instrumentation. In *Proc. 2005 Linux Symposium*, July 2005.

[14] B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[15] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)*, pages 117–130, February 22–25 1999.

[16] 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):263–276, Fall 1999.

[17] S. Zhang. *Practical Pointer Aliasing Analyses for C*. PhD thesis, 1998.

[18] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Proc. Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, October 1996.