Profile-Guided Specialization of an Operating System Kernel *

Somu Perianayagam, HaiFeng He, Mohan Rajagopalan,‡ Gregory Andrews, Saumya Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
Email: {somu, hehf, mohan, greg, debray}@cs.arizona.edu

Abstract

General-purpose operating systems such as Linux are increasingly replacing custom embedded counterparts on a wide variety of devices. Despite their convenience and flexibility, however, such operating systems may be overly general and thus incur unnecessary performance overheads in these contexts. This paper describes a new approach to mitigating these overheads by automatically specializing the OS kernel for particular execution environments. We use value profiling to identify targets for specialization such as frequent system call parameters. A novel profiling technique is used to identify frequently invoked procedure call sequences within the kernel. This information is used to sidestep the problems arising from indirect function calls when carrying out interprocedural compiler optimization. It drives a variety of compiler optimizations such as function inlining and code specialization that reduce the execution overheads along frequent paths. A prototype implementation that uses the PLTO binary rewriting system to specialize the Linux kernel is described. While overall performance data are mixed, the improvements we see argue for the potential of this approach.

1. Introduction

An operating system (OS) provides an interface between application programs on the one hand and a set of hardware resources on the other. As such, it has two complementary roles: to provide services (system calls) that aid the development and execution of application programs, and to manage the resources efficiently. Several research groups have proposed ways to customize OS services to optimize those that are heavily used—i.e., to make the common case fast [10]. One approach is to employ a micro-kernel and to allow applications to select the services that they require; examples are Mach [17], the exokernel [4], SPIN [1], and Flux [5]. Another approach is to provide support for specialized implementations of frequently used system calls; the seminal work here was dynamic specialization in the Synthesis kernel [15] and follow-on work on Synthetix [14] and an OS specialization toolkit [11].

This paper presents new, profile-guided techniques for specializing an existing operating system kernel to a given set of applications. These are implemented using the PLTO binary rewriting system [19]. The advantages of using binary rewriting are that we (1) do not require having access to kernel source code, (2) can deal with hand-written assembly code that is common within kernels, and (3) can process a kernel after it has been configured for a specific set of hardware resources.

Our focus at present is embedded systems, which have a fixed set of applications and execute on hardware platforms that often have severe resource constraints, such as limited memory and power. Because of these constraints, such systems often employ special-purpose operating systems or stripped-down versions of general-purpose systems. Our goal is to enable embedded system developers to employ a full-feature OS such as Linux for system development, and then automatically to specialize the operating system kernel to optimize the services that are frequently used.[1]

The specific contributions of the work described in this paper are the following:

- The approach is essentially automatic, requiring only a representative set of applications.
- It works on a general purpose OS (Linux) and should, in theory, be applicable to any OS.
- It employs a novel profiling technique to identify frequently invoked procedure call traces (paths) within the kernel.

The remainder of the paper is organized as follows. Section 2 gives an overview of the entire process, from profiling applications through rewriting the kernel. Section 3 describes how we identify frequent call paths within an OS kernel—including those containing indirect calls. Section 4 describes how we use the call traces to specialize the kernel. Section 5 gives experimental results. Finally, we discuss related work in Section 6 and give concluding remarks in Section 7.

* 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] We have also developed techniques to compact the kernel to exclude services that are not used, thus making the kernel both smaller and faster.

2. Overview

Our method for specializing a kernel to a given set of applications involves a series of steps to examine the applications and then to specialize the kernel. First, we execute each application using representative input and gather a trace of all system calls, including their arguments. This set of applications together with their input constitutes the *training set* used to generate profiles as described below. We post-process this trace to determine the most frequent system calls and the distribution of values of the arguments of those calls.

Second, we use PLTO to add instrumentation code to the kernel in order to generate basic block profiles. Input to PLTO is a relocatable binary program, in this case the Linux kernel.[2] PLTO first disassembles the kernel to produce an interprocedural control flow graph. PLTO then inserts instrumentation code to generate basic block and edge profiles and assembles a new kernel binary. We then execute the training set of applications to get basic block and edge profiles of kernel activity resulting from the applications.

Third, we use PLTO once again to instrument the (original) kernel so that it constructs call trace profiles for the frequently executed functions in the kernel. A *call trace* at a point p in the kernel is a sequence of call sites for the functions that are active when control reaches p. We use the basic block profiles generated in the second step to determine which functions to trace. We execute the training set of applications on the resulting instrumented kernel, in this case to produce call trace profiles. From these we can determine the most frequently executed function call paths in the kernel. Our profiling technique is able to handle indirect as well as direct function calls, and it considers paths that contain one or more common intermediate functions. Section 3 describes the details.

The most important kernel-processing step is to specialize the kernel based on the information gathered from system call traces (step 1), basic block profiles (step 2) and call trace profiles (step 3). There are three main components of this specialization: (1) create a new system call for each frequently invoked system call that has one or more constant arguments; (2) aggressively inline functions into this new system call based on the hot paths from this function into deeper levels of the kernel; (3) propagate constant arguments from the new system call into the inlined functions to specialize and streamline the code. We then employ additional code optimizations to further optimize the code: load/store forwarding to eliminate redundant memory accesses, peephole optimizations to combine adjacent instructions, and code layout to improve instruction cache usage and branch behavior. Section 4 gives details on how we use the call traces to implement function inlining and how we specialize to constant arguments.

The last step is to modify the application binaries to use the new system calls that were created during the optimization step. Each new system call is assigned a previously unused index in the Linux system call table. Hence, implementing this step merely requires changing the system call numbers at those places in the application binaries where an optimized call can be employed.[3] The arguments themselves are left on the stack as usual.

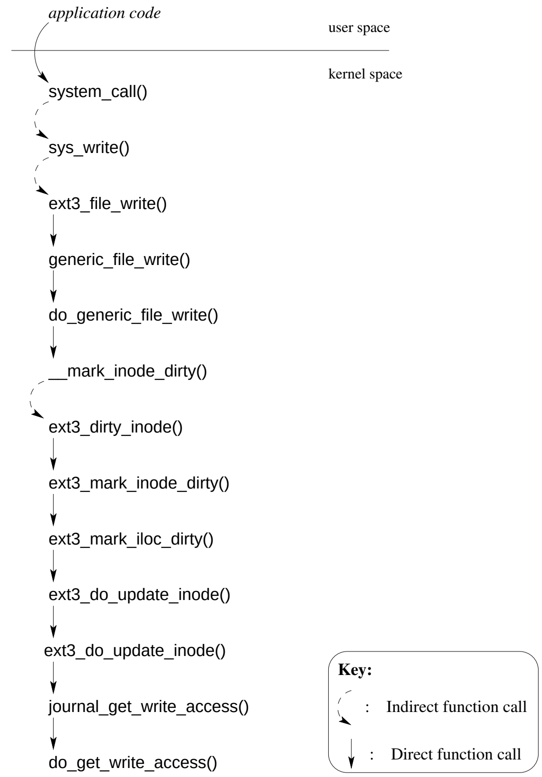

Figure 1. An example call trace in the Linux kernel

3. Profiling Call Trace Distributions

Operating system kernels often contain indirect function calls for reasons of maintainability and extensibility, and this poses a substantial challenge for specialization. Typically, execution environment information that can be exploited for specialization purposes is known at the "top" of the kernel—i.e., in the routines that are close to the application code—while the bulk of the kernel's execution time is spent in routines that are "deep" within the kernel. In order for specialization to be effective, it is necessary to communicate information to the deeper-level routines. However, the top-level routines are generally separated from the deeper-level code by several levels of function calls, many of which are indirect calls. The situation is illustrated by Figure 1, which shows a particular frequently executed sequence of function calls within the Linux kernel corresponding to the `write()` system call: it can be seen that there are three indirect calls on the call path leading from the top-level routine `system_call()` to the deeper-level routine `do_get_write_access()`.

Long call chains and indirect calls make it difficult to apply standard code specialization transformations to kernel code. For example, compilers often limit the scope of their optimizations to individual functions and do not prop-

[2] We require a relocatable binary in order to have information about entry points and relocatable addresses. A companion paper to this one [16] describes how we rewrite and instrument the kernel binary.

[3] If the application binaries use a system call that can be optimized for some cases but not all—e.g., because most calls have the same arguments, but not all of them do—then the kernel will have both the optimized and unoptimized versions.

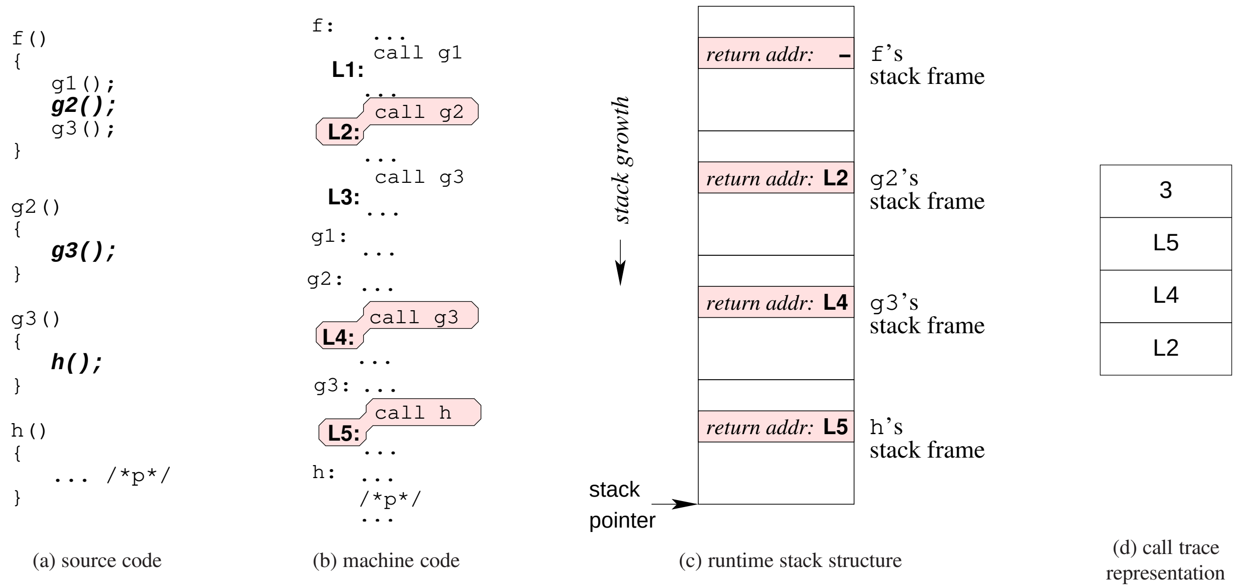

Figure 2. An example program and call trace

agate dataflow information across function boundaries. Furthermore, static analysis and optimization tools are usually very conservative in their treatment of indirect function calls: To overcome these problems, we use profiling techniques to identify common call paths, and then transform the code in such a way as to remove direct and indirect function calls for such paths and thereby allow optimizations to be carried out. There are two components to such profiling: identifying call paths, and then obtaining a profile estimating their distribution. The remainder of this section addresses each of these in turn.

3.1 Call Traces: Basic Ideas

Conceptually, a *call trace* at a point p in a program is a sequence of call sites for the function invocations that are active when control reaches p, starting with the function containing p and ending with the entry function(s) for the program. From an implementation perspective, a call trace can be viewed as simply the sequence of return addresses appearing on the call stack when control reaches p. Figure 2 illustrates this. Figure 2(a) shows a source code fragment containing a number of function calls. Now consider the program point labeled '/*p*/' in the function `h()`; suppose that control reached this point via the sequence of function calls

$$\texttt{f} \rightarrow \texttt{g2} \rightarrow \texttt{g3} \rightarrow \texttt{h}$$

Figure 2(b) shows relevant portions of the assembly/machine code corresponding to the source code of Figure 2(a). The labels L1, ..., L5 shown here are not actually present in the code, but have been added to highlight the return address associated with each `call` instruction. It can be seen from this that the return address associated with a function invocation uniquely identifies the call site, i.e., the location of the invocation. Figure 2(c) shows the runtime stack when control reaches the point '/*p*/' via the call sequence mentioned above. By traversing the runtime stack and extracting the return address from each frame, we can reconstruct the sequence of calls that are currently active at that point. A call traces is the sequence of return addresses $\langle \mathsf{L5}, \mathsf{L4}, \mathsf{L2} \rangle$ so obtained, preceded by a word indicating the length of the sequence (in this case 3); this is shown in Figure 2(d).

To construct a call trace, we traverse the stack looking for code addresses. Each time an address pointing to a code region is found, it is copied into a buffer. The resulting array of addresses is close to a call stack trace, but it may contain "noise" in the form of code addresses that are not return addresses. This happens, for example, when a function pointer is pushed on the stack. The set of call instructions—and thus the set of return addresses that can appear on the stack—are statically known assuming the kernel does not dynamically generate code containing function calls. We use this information to remove entries that are not valid return addresses.

Note that our call traces reflect only the runtime behavior of the program. For example, the program shown in Figure 2(a) contains another call path from `f` to the point `p`, namely, $\texttt{f} \rightarrow \texttt{g3} \rightarrow \texttt{h}$. However, since this path is not the one that is currently active, it is not reflected in the call trace. Furthermore, while this example considers only direct function calls, the process works exactly the same way for indirect calls.

3.2 Choosing Call Trace Profiling Points

We collect call traces at entry to each function that is considered to be "hot" with respect to some user-specifiable threshold $\theta \in (0, 1]$. The set of hot functions is determined using basic block profiles, as follows. Suppose that a basic block B containing m instructions is executed n times on some inputs, then let *weight* of B is defined as $weight(B) = mn$. The weight of a function f is the sum of the weights of its basic blocks, i.e., it gives the total number of instructions resulting from its execution. We first process the kernel using our binary rewriting tool to add instrumentation code that counts of the number of times each basic block in the program is executed. The instrumented kernel is then run on representative "training" inputs. The kernel is then processed once again by our binary rewriting tool, this time with the basic block profiles obtained from the training inputs. We then consider the functions in descending order of weight, and the set of functions whose total weight comprises at least a fraction θ of the total number of instructions executed by the kernel on the training input are labelled as "hot." For example, $\theta = 0.9$ means that the hot functions account for at least 90% of the instructions executed at runtime.

3.3 Collecting Call Trace Profiles

Our call trace profiling technique is inspired by the value profiling work of Calder *et al.* [2]. For each hot function f we maintain a table T_f containing N pairs $(calltrace, count)$. Each time execution enters a hot function f, we construct the call trace S at that point and check to see whether S appears in T_f: If S is in T_f, we increment the corresponding count. If S is not already in T_f, we attempt to insert it. If there is no room in T_f for a new entry, we simply discard S.[4]

It may happen that the most frequently occurring value overall is not one of the first N distinct values. We provide a mechanism for allowing later values to enter the table by periodically "cleaning" the lower half of the table. This means that the values are sorted based on their counts, and the $N/2$ least frequently occurring values are evicted. Our experiments indicate that, except for very small values of the cleaning interval, the actual cleaning frequency does not significantly affect either the speed or quality of profiling. Thus, we chose the same cleaning interval as Calder *et al.* [2], namely, 1000. In other words, for each hot function f, the table T_f is cleaned after the execution enters f 1000 times. After a table has been cleaned once, we must make sure that a new value can enter the steady part of the table before we clean again. We set the new value of the cleaning interval to 1000 plus the count of the last entry in the steady part. This means that if a new value occurs almost exclusively between cleanings, its count should be higher than the last entry in the steady part of the table. It will then be moved into the steady part, evicting the least frequently encountered value.

The value N governs the number of entries in the profiling tables, so it affects both the accuracy and running time of the call trace profiler. Profiling tables must be large enough that the most frequent entries will be in the table, but the larger the value of N the greater the number of entries the profiler has to look at each time it reaches a profiling point. We have found that a table size of $N = 12$ works well in practice.

4. Specialization

4.1 Function Inlining on Hot Call Paths

Once we have call traces for hot functions, we carry out function inlining along frequently taken traces. The idea is to bring frequently executed code from different functions along the call path into the body of the same function. This has three benefits: (1) it eliminates the procedure linkage overhead by getting rid of the function call/return instructions as well as other instructions in the calling and return sequences; (2) it increases locality by bringing frequently executed code closer together, with potentially beneficial effects on instruction cache utilization; and (3) it exposes opportunities for code specialization, as described in the following section.[5]

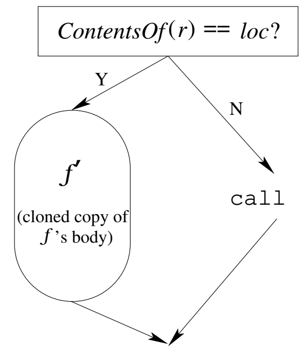

Figure 3. Guarded inlining of indirect function calls

Inlining a direct call is conceptually straightforward: given a `call` instruction I to a function f, let the next instruction be J. To inline this call, we create a clone of f, say f', then replace the call instruction I by an unconditional branch to the entry point of f', and each `return` instruction in f' with an unconditional branch to J. Subsequent profile-guided code layout [13] may optimize away some of these unconditional branches. When carrying out inlining at the binary level, there are some subtleties that have to be addressed, e.g., jump tables have to be cloned and their targets updated so as to point into the cloned code.

Inlining an indirect call is somewhat more complex, because an indirect call may have many possible targets. We currently inline at most one indirect call target. In order to ensure that the resulting code is semantically equivalent to the original code, we use "guarded inlining." Suppose we have an indirect call instruction '`call *r`' that branches to the address contained in a register r. Suppose that we decide to inline one of the targets of this call, a function f at memory location *loc*. As shown in Figure 3, guarded inlining creates a clone f' of f, and transforms the call site so that control branches to f' only if, at runtime, the target is found to be f.

Carrying out inlining within the kernel requires care, since we have to ensure that exceptions that are raised from the inlined code are handled correctly. For example, functions that copy data between user and kernel space first check whether the page to be copied is valid, and raise a page fault if it is not. Our current implementation conservatively does not inline any function that has a pointer into it from the exception handler table.

4.2 Code Specialization

Once inlining has been carried out, specialization involves propagating information about the actual arguments at the call site into the inlined cloned body and thereby generating code specialized to that call site. When the arguments are provably known constant values, the optimization boils down to the classical compiler optimization of constant

[4] The total number of discarded values gives an indication of the imprecision of the profile. This can be computed at the end of execution by subtracting the total counts for the entries in the table from the the number of times f was called, which is available from f's basic block profile.

[5] A potential problem with aggressive inlining is that it can cause code bloat that adversely affects the program's instruction cache utilization and results in a loss of performance, but this problem can be assuaged by careful attention to memory footprints during the inlining process.

propagation. It turns out that specialization can be carried out even when the argument whose value we are specializing for is not provably constant. In the latter case, we add runtime tests to the code to guard the specialized code and ensure that it is entered only for the appropriate values of the arguments. Thus, consider a function that has an argument x and body C, and suppose that x takes on a value v on most—but not all—of the invocations of f. Then we can transform the body of f to

if $(x == v)$ **then** $C_{[x=v]}$ **else** C

where $C_{[x=v]}$ represents the residual code of C after it has been specialized to the value $x = v$.[6]

Carrying out code specialization at the binary level, where some of the code consists of hand-written assembly that need not adhere to standard compiler conventions, presents some challenges. In particular, since function arguments are passed on the stack in the x86 architecture, this requires the ability to reason about the caller's stack frame and its relationship to the callee's stack frame. The issues can be illustrated by the following source code fragment from the Linux kernel:

```
sys_read(unsigned int fd, char *buf, size_t count)
{
  ...
  file = fget(fd);
  if (file) {
    if (file->f_mode & FMODE_READ) {
      ret = rw_verify_area(READ, file,
                           &file->f_pos, count);
      if (!ret) {
        ...
        ret = read(file,buf, count,&file->f_pos);
    ...
}
```

Suppose that the argument `count`, shown highlighted in the code above, almost always takes on the same value. We can use this knowledge to carry out guarded specialization of the called routines `rw_verify_area()` and `read()` (whose bodies have presumably been inlined into `sys_read()` based on hot call traces). To do this, however, we have to overcome two major hurdles:

1. The caller and callee code do not refer to the actual parameters in the same way, making it nontrivial to identify correspondences between them.
2. The code we wish to specialize may be preceded by other computations which potentially could alter the value we use to specialize upon.

We address these problems via stack analyses that infer the size of each function's stack frames (to relate stack accesses in one function to those in other functions) and the possible effects of memory load and store operations on the stack.

To determine the size of a function's stack frame, we track operations that update the stack and frame pointers and use this to infer, for each basic block of a function, the net change in the size of the stack frame due to the execution of the function. This information is then propagated across the control flow graph of the function to compute the maximum size of its stack frame.

Once we have determined stack frame sizes, we use *use depth* and *kill depth* analyses to estimate the effect of function calls on the runtime stack. The *use depth* is an integer (or ∞) that represents an upper bound on the depth in the stack—relative to the top of stack when the function is called—from which the function, or any function(s) it calls, may read a value. Similarly, the *kill depth* represents an upper bound on the stack depth to which a function, or any functions it calls, may write a value. In the code fragment for the function `sys_read()` shown above, the idea would be to show that the kill depth of the function `fget()` is small enough that the stack location containing the parameter `count` can be guaranteed to not be overwritten. Use and kill depths are computed in two phases. First, a local analysis of each function computes the effects of the instructions of that function itself. This information is then propagated iteratively along the call graph of the program from callee to caller. The details of these analyses are discussed in [19].

5. Status and Experimental Results

We have implemented our ideas within the PLTO binary rewriting system. At present, we have implemented all of the instrumentation and profiling phases (basic block profiling, value profiling, call stack trace profiling), analyses, and optimizations described in the paper. This section presents initial results that show the effect that they have upon performance. However, the work is in its early stages, and we have not yet had time to tune our implementation carefully. For example, some of our analyses are currently quite conservative in their handling of indirect memory accesses, and this hampers the optimizations. We are currently working on extending these analyses to remove these sources of imprecision, as discussed at the end of this section.

We ran experiments on 3.2GHz Pentium 4 machines with 1GB RAM running the Linux 2.4.31 kernel. The kernel was configured minimally as it would be for an embedded system, e.g., there is only a single driver for each hardware device. This kernel contains 5133 functions, 81263 basic blocks, and over 349K instructions. We used PLTO to generate both specialized and instrumented kernels based on a statically linked relocatable image of the kernel binary. PLTO takes about two minutes to process and generate a new kernel.

Generating a relocatable kernel binary required changing only a single line of the original Linux *Makefile*. The new executable generated by PLTO was converted into a bootable image format by using a script that replicated the original build procedure.

In order to measure performance attributes of the kernel, we created a new system call:

```
int kprofile(command, type);
```

The first argument specifies the action to be performed; possible actions include starting measurements, writing out the data, and stopping measurements. The second parameter specifies the type of measurement to take. In the current implementation, the options include basic block profiling, edge profiling, call stack profiling, cycle counts, and various

[6] Notice that in this case, the unoptimized code on the **else**-branch actually suffers a performance degradation because of the test that has been introduced. For the specialization to be profitable, it then becomes necessary to carry out a profile-based cost-benefit analysis. The details of such analyses are beyond the scope of this paper, but we have discussed them elsewhere [12].

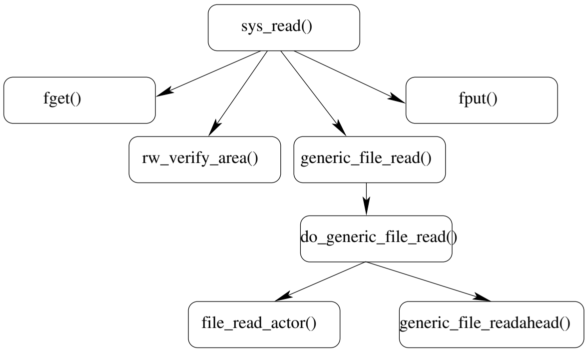

Figure 4. Frequent call sequences for read system call

	Original	*Specialized*	*Change* (%)
Functions	5133	5139	0.1
Basic blocks	81263	81498	0.2
Instructions	349768	351020	0.35
Text size (bytes)	1,133,150	1,121,664	-1.0

Figure 5. Static effects of kernel specialization

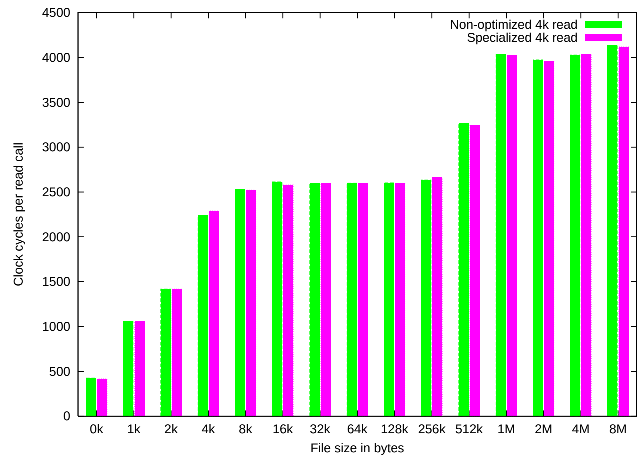

Figure 6. Performance of read pre and post specialization

cache usage counts. To measure cycle counts, we use the Pentium `RDTSC` instruction to read the hardware time stamp counter on every entry to and exit from the kernel. Each value is saved when we are doing timing measurements; they are written out by the appropriate call of `kprofile` and are post-processed to calculate the total time spent in the kernel for each system call.

In order to analyze the effects of our optimizations, we constructed what we call micro-benchmarks for selected system calls. In particular, we ran the applications in the MiBench benchmark suite for embedded systems [7], determined the most frequently used system calls, and then constructed small test programs to measure the impact of optimizing those calls. Below we describe the experiments for two of the popular system calls: `sys_read` and `sys_lseek`.

Our micro-benchmark program for read and seek is as follows:

```
while (i < repetitions) {
  i++;
  while(1) {
    if ((n = read(fd, buf, 4096)) <= 0)
        break;
  }
  lseek(fd, 0, SEEK_SET);
}
```

The body of the outer loop reads a file, then seeks back to the start of the file. Each `read` fetches a 4K block, which is the size of a page in Linux. We repeat the file read and seek several times in order to get a fairly accurate measure of the time spent in the kernel processing reads and seeks.

The benchmark program was processed as described in Section 2. In order to collect basic block profiles, we ran the micro-benchmark program using input file of sizes of 4k, 8k, 32k, 64k, 256k, 512k, 1M, 2M, and 4M. We then combined the individual basic block profiles to get a single composite profile. Basic block profiling identified 66 functions as being hot; almost all of these are along the call paths from `sys_read` and `sys_lseek`. The call path traces revealed four different hot call paths for `sys_read`, as illustrated in Figure 4. There was a single hot call path in `sys_lseek` of length three. All these hot functions were inlined into `sys_read` and `sys_lseek` and then specialized as described in Section 3. Finally, we used edge profiles of the kernel—which were computed at the same time as basic block profiles—to generate a good code layout [13].

Figure 5 summarizes how our transformations affect the size of the kernel. The one additional function in the specialized kernel corresponds to the code for the *kprofile* system call. The static number of basic blocks and instructions increase only slightly because we inlined only the hot functions. Because PLTO automatically removes unreachable code when it constructs the control flow graph of the kernel, the text size of the specialized kernel is actually smaller than that of the original kernel.

In order to measure the effect of specialization, we ran the micro-benchmark program with 25 repetitions of the outer loop, and we used files of varying size from 1KB to 4MB—the same sizes for which we had done basic block profiling. For each file size, we ran the micro-benchmark five times for both the original and specialized kernels. Each experiment was carried out by rebooting the kernel on a machine that was isolated from the network and had no other users or executing applications. In each experiment, we used `kprofile` to calculate the number of cycles used by `read` and `lseek` during each of the 25 repetitions of the outer loop. We dropped the highest and lowest of the 25 values, then averaged the remaining 23 values to obtain the timing data for a single experiment. We ran five experiments for each file size, then averaged the results to obtain timing results for each file size.

Timing results for `sys_read` are shown in Figure 6; those for `lseek` are shown in Figure 7. The plotted values are the average—computed as described above—for a single system call for each file size. Even though code specialization removed about 4% of the dynamic instructions from the `sys_read` call, there is no noticeable decrease in execution time. This is because reading a file requires copying data

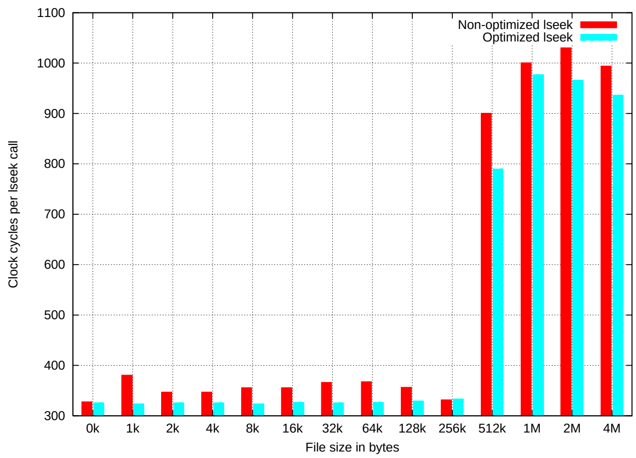

Figure 7. Performance of lseek pre and post specialization

from kernel space to user space—which is implemented by a single "block move" instruction—and this single instruction dominates execution time. On the other hand, we see an improvement of 6 to 8% for the `sys_lseek` call. This call does not involve any data transfer between kernel space and user space, so the 5% reduction in the dynamic instruction count translates into visible performance improvement.

We are encouraged by the speed improvements we see for the `lseek` call. We have verified, by turning off specialization so that profile-guided code layout is the only optimization, that these performance improvements are due mainly to code specialization. We have identified other system calls, such as `mmap` and `munmap`, that can potentially yield similar performance improvements. By performing manual optimizations on just a single parameter to these system calls, we have been able to get 4-5% gain in performance. By (hand) optimizing on 5 out of the 6 parameters for the `mmap` call, we are able to get about 20% improvement.

We are currently exploring several ways to improve performance results and increase specialization opportunities. First, many system calls, including `sys_read`, have data transfers between user space and kernel space, and as noted above, this is a time-consuming operation. However, on many embedded systems both application and kernel code reside in a single address space. We are currently exploring ways to merge user space and kernel space into a single address space automatically. This would greatly reduce the cost of data transfers and could potentially provide more opportunities for specialization. Obviously, merging of address spaces will have to be done with care so as not to expose vulnerabilities that are prevented by having separate address spaces.

Second, binary specialization of the Linux kernel is handicapped by a lack of semantic knowledge at the binary level and by the use of hand-coded assembly. For example, in order to propagate constants through the stack, we have to be able to compute use-depths and kill-depths for each function as described in Section 4.2. However, the scheduler is reachable from the `mmap` call. Because the scheduler manipulates the stack pointer in nonstandard ways, our stack analysis has to bail out and assume that nothing can be determined about the stack. This fact ripples up to the `mmap` call and precludes us from doing constant propagation along the hot path in `mmap`, even though the stack does in fact have the same contents whether or not the scheduler is called. If we had higher-level semantic knowledge of the scheduler, we could determine that it is not reassigning the stack pointer in ways that preclude constant propagation. We have been able to make productive use of semantic knowledge available at source code level in our compaction work. We are currently looking at how to use source-level semantic knowledge to aid in specialization.

Finally, we currently profile system call arguments one at a time and specialize on at most one argument. This could be changed to profile a group of system call arguments. This collective profile could then be used to specialize system calls more aggressively. Initial results are promising.

6. Related Work

There has been a lot of work on binary rewriting over the last decade, e.g., [3, 20, 18, 21]. Most of this work has dealt with translating binaries from one machine language to another, or with optimizing application programs. The only other projects we are aware of that have used binary rewriting to manipulate operating system kernels are Diablo [3] at Ghent University in Belgium and Vulcan [21] in Microsoft. The focus of Diablo has been reducing the size of the kernel in an embedded systems, not improving performance.

The work most closely related to ours is that on operating system specialization. The Synthesis kernel [15] introduced the idea of dynamically generating specialized code for system calls within a customized kernel. A follow on project, Synthetix [14], extends the ideas in Synthesis with incremental and optimistic specialization. Incremental specialization allows specialized code modules to be generated as information becomes available. Optimistic specialization allows specialized modules to depend on system states that are likely to occur but not certain. Our use of "guarded inlining" is an instance of optimistic specialization in the sense that we generate specialized code in expectation of improved performance for the common case, and guard execution of that code to ensure that we have the common case. However, we implement specialization automatically whereas it has to be done by hand with Synthetix. Moreover, we do not "plug" and "unplug" specialized code dynamically.

The latest follow-on to Synthesis and Synthetix is a collection of specialization tools and techniques described in [11]. These tools support static, dynamic, and optimistic specialization of system code. We only do static specialization, but we support optimistic, value-based specialization. The tools in [11] require significant manual intervention as well as code rewriting, whereas ours are automatic. We also use profile information to guide optimization.

Our notion of call trace profiling in some ways resembles the idea of call path profiling [6, 8, 9]. Call path profiling aims to measure the time spent along different calling contexts, with the aim of providing feedback to programmers for performance tuning purposes. By contrast, we estimate the frequency distribution of call traces, with the goal of guiding an automated optimization tool. Call path profiles would not be suitable for our needs because time spent on an optimization path is not necessarily a good predictor of optimization opportunities. The implementation techniques used for call path profiling are also very different from ours. Early approaches to call path profiling iteratively explored call paths using timer macros to estimate their cost [8]; more recent implementations have used interrupt-based sampling [6, 9]. By contrast, our implementation—in particular, the way the profiling tables are managed—is based on ideas from value profiling [2, 12].

7. Conclusions

Recent years have seen an increasing trend towards the deployment of general-purpose operating systems, such as Linux, on embedded processors. Because of the generality of such operating systems, they may incur unnecessary performance overheads. This paper describes a new approach to reducing some of these overheads via the use of binary rewriting to instrument the OS kernel, collect execution profiles, and use these profiles to carry out code specialization along frequently executed paths. A major contribution of this work is a novel profiling technique that can be used to identify frequently executed function call paths, thereby sidestepping the problems arising from indirect function calls when carrying out interprocedural compiler optimization. Among the advantages of our approach are that (i) we do not need access to source code and are able to handle hand-written assembly code; and (ii) our transformations are automated and do not need manual intervention. Our ideas have been implemented using the PLTO binary rewriting system and evaluated on the Linux 2.4.31 kernel binary. The performance results are currently mixed, with some inputs showing significant performance improvements and others showing slowdowns. We are currently in the process of tuning our implementation and investigating additional ways to specialized the code.

References

[1] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, Dec 1995.

[2] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, vol. 1, March 1999.

[3] Dominique Chanet, Bjor De Sutter, Bruno De Bus, and Koen De Dosschere. System-wide compaction and specialization of the linux kernel. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 05)*, pages 95–104, 2005.

[4] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Symposium on Operating Systems Principles (SOSP 95)*, pages 251–266, 1995.

[5] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, Saint-Malo, France, Oct 1997.

[6] N. Froyd, J. M. Mellor Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. 19th Annual International Conference on Supercomputing*, pages 81–90, June 2005.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[8] R. J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering*, pages 296–306, May 1992.

[9] R. J. Hall. Call path refinement profiles. *IEEE Transactions on Software Engineering*, 21(6):481–498, June 1995.

[10] Butler W. Lampson. Hints for computer system design. In *Operating Systems Review*, pages 33–48. ACM, October 1983.

[11] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A Goel, , and P. Wagle. Specialization tools and techniques for systematic optimization of system software. *ACM Trans. on Computer Systems*, 19(2):217–251, May 2001.

[12] R. Muth, S. Watterson, and S. K. Debray. Code specialization based on value profiles. In *Proc. 7th. International Static Analysis Symposium (SAS 2000)*, pages 340–359, June 2000.

[13] 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.

[14] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 314–324, Copper Mountain, CO, Dec 1995.

[15] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[16] Mohan Rajagopalan, Somu Perianayagam, HaiFeng He, Gregory Andrews, and Saumya Debray. Binary rewriting and instrumentation of an operating system kernel. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, October 2006.

[17] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, and Michael B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[18] Ted Romer, Geoff Voelker Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian N. Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[19] 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.

[20] A. Srivastava and A. Eustace. ATOM—A system for building customized program analysis tools. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[21] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.