Code Compaction of an Operating System Kernel*

Haifeng He, John Trimble, Somu Perianayagam, Saumya Debray, Gregory Andrews
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA
{hehf, trimble, somu, debray, greg}@cs.arizona.edu

Abstract

General-purpose operating systems, such as Linux, are increasingly being used in embedded systems. Computational resources are usually limited, and embedded processors often have a limited amount of memory. This makes code size especially important. This paper describes techniques for automatically reducing the memory footprint of general-purpose operating systems on embedded platforms. The problem is complicated by the fact that kernel code tends to be quite different from ordinary application code, including the presence of a significant amount of hand-written assembly code, multiple entry points, implicit control flow paths involving interrupt handlers, and frequent indirect control flow via function pointers. We use a novel "approximate decompilation" technique to apply source-level program analysis to hand-written assembly code. A prototype implementation of our ideas on an Intel x86 platform, applied to a Linux kernel that has been configured to exclude unnecessary code, obtains a code size reduction of close to 24%.

1. Introduction

Recent years have seen increasing use of general-purpose operating systems, such as Linux, deployed in embedded contexts such as cell phones, media players, and other consumer electronics [1]. This is due in great part to technological trends that make it uneconomical for vendors to develop custom in-house operating systems for devices with shorter and shorter life cycles. At the same time, however, these operating systems—precisely because they are general-purpose—contain features that are not needed in every application context, and that incur unnecessary overheads, e.g., in execution speed or memory footprint. Such overheads are especially undesirable in embedded processors and applications because they usually have resource constraints, such as a limited amount of memory. Thus, the memory footprint of the code is especially important, and a program that requires more memory than is available will not be able to run.[1]

This paper focuses on automatic techniques for reducing the memory footprint of operating system kernels in embedded systems. Such systems tend to have relatively static configurations: at the hardware end, they are limited in the set of devices with which they interact (e.g., a cell phone or digital camera will typically not have a mouse interface); at the software end, they usually support a fixed set of applications (e.g., we do not routinely download or build new applications on a cell phone or digital camera). This implies that an embedded system will typically use only some of the functionality offered by a general-purpose operating system. The code corresponding to the unused functionality is unnecessary overhead, and should be removed. Some of this overhead can be removed simply by configuring the kernel carefully so as to exclude as much unnecessary code as possible. However, not all overheads can be removed in this manner. For example, a given set of applications running on an embedded platform will typically use only a subset of the system calls supported by the operating system; the code for the unused system calls is then potentially unnecessary. Such unnecessary code typically cannot be eliminated simply by tweaking the configuration files; additional analysis is required. This paper discusses how such analysis may be carried out in order to identify code that can be guaranteed to be unnecessary. Specifically, this paper makes the following contributions:

1. It discusses issues that arise in binary rewriting of operating system kernels and discusses how they may be handled.

2. It introduces a notion of "approximate decompila-

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

[1] Virtual memory is not always an option in embedded systems; even where it is available, the energy cost of paging out of secondary storage can be prohibitive.

tion" that allows us to apply source-level program analysis to hand-written assembly code. This improves the precision of our analysis considerably, and in a simple way, while ensuring that the safety of our transformations is not compromised.

There has been a significant body of work on code size reduction, with different objectives (e.g., reducing wire transmission time vs. reducing memory footprint during execution), on different program representations (e.g., syntax trees, byte code, or native code), and making different assumptions regarding the requirement for runtime decompression and the availability of hardware support for any such runtime decompression. See Beszédes *et al.* [4] for a survey. Almost all of this work has been done in the context of application code. By contrast, the work described in this paper focuses on operating system kernels in native code representation, and it aims to reduce their runtime memory footprint. We do so using automatic program analysis and transformations, with no programmer intervention in the form of code annotations or other similar input. A prototype implementation of our ideas on an Intel x86 platform, applied to the Linux kernel configured minimally so as to exclude all unnecessary code, is able to achieve a code size reduction of nearly 24%, on average, on the MiBench suite of embedded system applications [11].

2. Background

OS kernel code is quite different from ordinary application code. A significant problem is the presence of considerable amounts of hand-written assembly code that often does not follow the familiar conventions of compiler-generated code, e.g., with regard to function prologues, epilogues, and argument passing. This makes it difficult to use standard compiler-based techniques for whole-system analysis and optimization of kernel code. To deal with this problem, we decided to use binary rewriting for kernel compaction. However, while processing a kernel at the binary level provides a uniform way to handle code heterogeneity arising from the combination of source code, assembly code and legacy code such as device drivers, it introduces its own set of problems. For example, it is necessary to deal with a significant amount of data embedded within executable sections, implicit addressing requirements that constrain code movement, and occasional unusual instruction sequences. As a result, binary rewriting techniques applicable to application code do not always carry over directly to kernel code (the issues that arise, and our approach to handling them, are discussed in more detail elsewhere [19]).

An especially important issue for code compaction is that of control flow analysis, both intra-procedural and inter-procedural. This is because, in practice, most of the code size reductions arising from compaction comes from the detection and elimination of dead and unreachable code [7]. For soundness reasons, we have to ensure that we only eliminate code that can be guaranteed never to be needed during any future execution. This means that imprecision in control flow analysis directly affects the amount of code that can be eliminated.

Unfortunately, control flow analysis in operating system kernels is complicated by the interaction of two separate problems. First, there are significant amounts of hand-written assembly code, as mentioned above. 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 function calls.[2] 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.

A final 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; both have to be taken into account in order to guarantee soundness.

3. Pointer Analysis: Resolving Indirect Function Call Targets

Static analysis of OS kernel code is complicated by the presence of hand-written assembly code. 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

[2] 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.

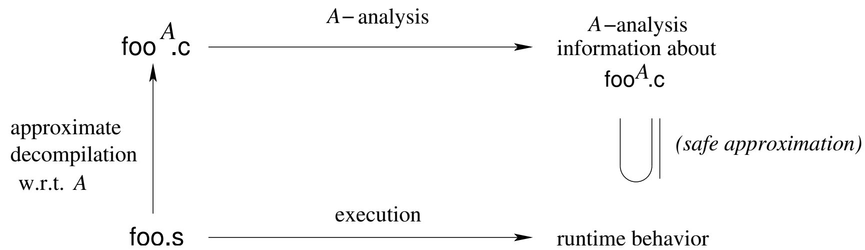


Figure 1. Using approximate decompilation for program analysis

pointer aliasing—resulting in overly conservative analysis that loses a great deal of precision. Nor can such assembly code be ignored, since soundness demands that all possible execution behaviors of the program be taken into account.

One possible solution to this problem would be to decompile the hand-written assembly code back to equivalent C source code that could then be analyzed by source-level analysis. The problem with such an approach is that it is not obvious that all of the kernel assembly code can be reverse engineered back to equivalent C source code. For example, "system instructions" on the Intel x86 architecture, such as "*load interrupt descriptor table register*" and "*invalidate TLB entry*," do not have obvious C-level counterparts. Moreover, even in situations where reverse engineering is possible, it can be complicated and involve a great deal of engineering effort. Instead, we deal with this problem using an approach we call "approximate decompilation," which automatically maps hand-written assembly code back to C source files for analysis purposes. The idea, illustrated in Figure 1, 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 use a source-level pointer alias analysis technique called FA-analysis to identify the possible targets of indirect function calls. The remainder of this section discusses how this is carried out.

3.1 FA Analysis

There is a large volume of literature on pointer alias analysis, with a variety of assumptions, goals, and trade-offs (see, for example, the discussion by Hind and Pioli [12]). In general, these analyses exhibit a trade-off between efficiency and precision: the greater the precision, the greater the analysis cost, i.e., the lower the efficiency. FA-analysis is a flow-insensitive context-insensitive pointer alias analysis, originally due to Zhang *et al.* [23, 24], that is at the efficiency end of this trade-off.

A detailed discussion of FA-analysis is beyond the scope of this paper; the interested reader is referred to the original papers on this analysis [16, 23, 24]. The essential idea in this analysis is to maintain equivalence classes of names that may alias each other in memory. Program constructs that result in values being propagated from one variable to another, e.g., assignment statements and parameter passing during a function call, cause the corresponding equivalence classes to be merged. For structure and union references, this merging propagates recursively down to the equivalence classes for the constituent fields. This merge process ignores the execution order of program statements: if two variables can be aliases anywhere in the program, then they are taken to be potential aliases everywhere in the program. This flow-insensitivity makes for a fast analysis but also makes it imprecise.

Milanova *et al.* have observed, however, that despite its low precision for general-purpose pointer alias analysis, FA analysis turns out to be quite precise in practice for identifying the targets of indirect function calls [16]. The authors attribute this to the fact that programmers typically use function pointers in a few specific and relatively simple stylistic ways.

3.2 Approximate Decompilation of Kernel Code for FA Analysis

As Figure 1 suggests, the way in which approximate decompilation is carried out depends in part on the source-level analysis that will be applied to the resulting source files. This section discusses approximate decompilation of assembly code in the Linux kernel code for FA analysis. For concreteness, we discuss kernel assembly code on the Intel x86 architecture.

The hand-written assembly instructions in the Linux kernel falls into two broad groups: (1) general-purpose instructions that perform basic data movement, arithmetic, logic, and program control flow operations, and (2) system instructions that provide support for operating systems and executives [13]. We process these instructions as follows:

– System instructions (the second group above) manipulate only the hardware (or data related to the hardware) and 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, 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.

– The remaining instructions are those that move data and those that perform arithmetic and logic operations. These instructions are translated to the corresponding operations in C. For example, a register load instruction, 'mov $0, %eax,' is translated to an assignment 'eax = 0'.

Since the results of approximate decompilation are used only by a program analysis tool, we currently do not attempt to raise the level of abstraction of the generated C code beyond that produced by this straightforward translation.

Our decompiler maps registers in the assembly code to global variables of type *int* with 32-bit values; the 16-bit and 8-bit registers (which are aliases of parts of the 32-bit registers) are also mapped to 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.

Memory locations referenced as absolute addresses in the assembly code are also treated as global variables. Since there is little type information available at the assembly level, we declare memory locations as having type `MEMOBJ`, which denotes a word in memory.[3] An object spanning a series of memory locations in the assembly code is treated as an array of `MEMOBJ` in the generated C code. This is illustrated in Figure 2(a). The segment of assembly code shown on the left side in Figure 2(a), taken from the file `entry.S` in the Linux kernel, defines the system call table that contains the function addresses of all system call handlers. Since `sys_call_table` spans a series of (initialized) memory locations in the assembly code, we map it to an (initialized) array in the generated C code shown on the right side in Figure 2(a). Moreover, since the symbols for the system call handlers are not themselves defined in `entry.S`, they are declared as `extern` objects 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.

Functions in the assembly code are identified from symbol table information and mapped to functions in the generated C code. Memory locations accessed through the stack pointer register %esp are assumed to be on the stack; these are mapped to local variables in the corresponding C function, with variables accessed via different displacements within the stack frame being mapped to different local variables in the generated C code. Actual parameters to a call are identified by similarly examining displacements in stack references, as illustrated in Figure 2(b). 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. Such prototypes are then used by the source-level analysis to identify aliasing between actuals and formals. 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, as illustrated in Figure 2(c).

4. Identifying Reachable Code

4.1 Reachability Analysis

The source-level FA analysis produces a set of possible call targets for each indirect procedure call in the kernel. Our kernel binary rewriter takes this information as an input and constructs a program call graph for the entire kernel.

Unlike ordinary applications, an operating system kernel contains multiple entry points. These entry points are the starting points for our reachability analysis. We classify kernel entry points into four categories: (1) the entry point for initializing the kernel (for the Linux kernel, this is the function `startup_32`), (2) system calls invoked during the kernel boot process, (3) interrupt handlers, and (4) system calls invoked by user applications. Once the entry points into the kernel have been

[3]Our implementation defines this type as '`typedef MEMOBJ int;`'.

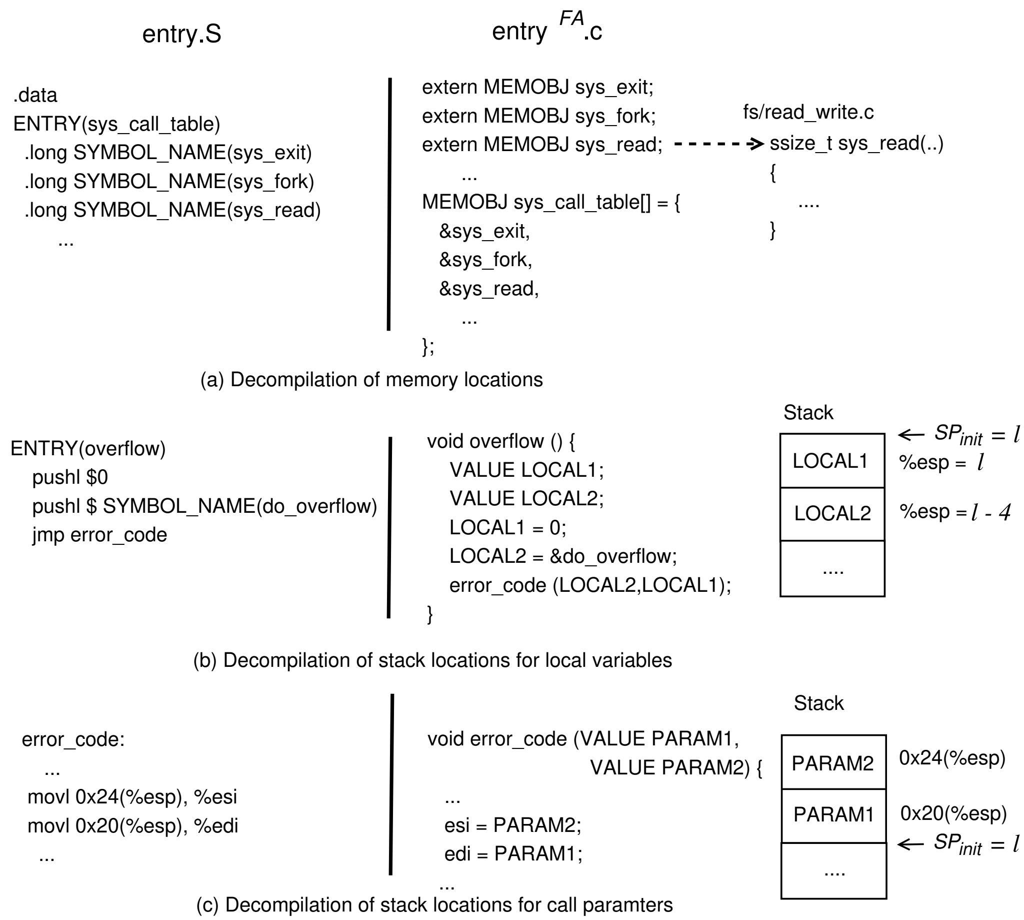


Figure 2. Examples of approximate decompilation of assembly code to C code for FA analysis.

identified, our reachability analysis performs a straightforward depth-first traversal of the program call graph to identify all the reachable functions in the kernel.

4.2 Improving the Analysis

During the initialization phase of kernel bootup (e.g., before the `init` program in Linux begins execution), execution is deterministic because there is only one active thread and execution depends only on the hardware configuration and the configuration options passed through boot command line. In other words, the initialization code can be considered to be "static" in the partial evaluation sense [14]. This means that if the configuration is not changed, we can safely remove any initialization code that is not executed. We use this idea to further improve our reachability analysis.

Our goal is to identify the static functions in the kernel, i.e., functions whose execution is completely determined once the configuration options and the hardware are fixed. To this end, we take advantage of a Linux kernel feature used to identify initialization code, most of which is not needed, and can be reclaimed, after initialization is complete. In particular, the Linux kernel simplifies this reclamation by segregating data and code used only for initialization into two sections in the ELF binary: `.text.init` and `.data.init`. Once the initialization of the kernel finishes during bootup, the kernel frees the memory pages occupied by these two sections to save physical kernel memory.

We use this knowledge to initialize the set of *static* functions to those appearing in the `.text.init` section. We then propagate this information as follows to find other functions that are not in the `.text.init` section but whose execution can be inferred to be completely determined given the command-line configuration options and hardware setup:

1. Mark all functions in `.text.init` section as *static*.

Procedure Reachability-Analysis
$worklist \leftarrow$ functions that are entry points into the kernel
while $worklist \neq \emptyset$ **do**
  $f \leftarrow$ Remove a function from $worklist$
  Mark f as reachable
  for every indirect/direct call target c of f **do**
    if c is *static* $\wedge$ c is not executed based on profile **then**
      continue
    else if c is not marked as reachable **then**
      Add c into $worklist$
    end if
  end for
end while

Figure 3. The improved reachability analysis algorithm

2. Based on the call graph of the Linux kernel, mark all functions that are not called by any other function as *static*.

3. If all the direct and indirect callers of a function F are *static*, then mark F as *static*. Repeat this process until there are no changes.

Once we have computed the set of functions that are considered to be *static* during kernel initialization, we use the results to improve our reachability analysis as shown in Figure 3. The improvement is that when a potentially reachable function is found, if the function is marked *static* and if, based on profile data, it was not called during kernel initialization, then we do not add it to the set of reachable functions.

4.3 Handling Exception Handlers

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 4: 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. 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 4, we would add such an edge from L_1 to L_2.

5. Kernel Compaction

Once all the potentially reachable code in the kernel has been identified, a variety of size-reducing code transformations can be applied to the kernel. Our transformations can be broadly grouped into three categories:

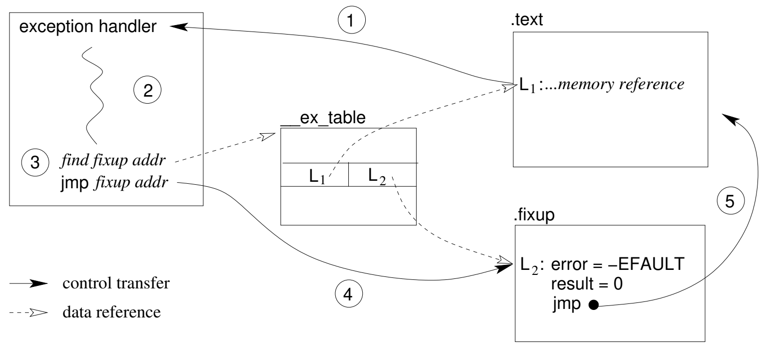
1. *Unreachable code elimination*. This identifies and deletes code that cannot be reached during execution [7].

2. *Whole function abstraction*. This identifies situations where multiple different functions have identical code, and removes all but one instance of such code. (Unlikely as this situation might seem, this optimization yields a size reduction of over 3% on the Linux 2.4.25 kernel.)

3. *Duplicate code elimination*. This involves transformations (other than whole function abstraction) where duplicate code instances are identified and eliminated. We use two different code transformations for this: tail-merging and procedural abstraction [7].

Applying these transformations to the kernel code involves some subtleties that have to be taken into account during code compaction. Here we describe two such situations.

The first involves a small number of functions in the kernel bootup code that execute prior to page table initialization, and which are required to be at specific fixed addresses. Such functions therefore cannot be moved in memory during the code compaction process. Our current implementation uses a fixed list of functions that cannot be moved: there are some 71 such functions, out of a total of roughly 4,600 functions in the input kernel binary (see Table 1).



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 back into the text segment.

Figure 4. Control flow during the handling of exceptions in the Linux kernel

The second issue is that some forms of procedural abstraction require that a global memory location be allocated to save the return address of the procedure. We currently exclude such code fragmennts for procedural abstraction within the kernel. There are two reasons for this. First, if a page fault occurs when accessing this location to store a return address and the page tables have not yet been initialized, the kernel will crash. Second, since the kernel is multi-threaded in general, using a single global location can lead to incorrect results if one thread overwrites the return address stored there by another thread; this means that the memory allocation has to be done on a per-thread basis, which complicates the implementation and reduces its benefits. However, this exclusion has no effect on procedural abstraction of code fragments that do not have to save the return address in global memory, but can leave it on the stack.

	With networking	*Without networking*
Functions	4,584	3,882
Basic blocks	72,951	55,708
Instructions	268,335	205,587
Code size (Kb)	836.70	641.61

(a) Linux 2.4.25

	With networking	*Without networking*
Functions	4,388	3,914
Basic blocks	71,118	51,609
Instructions	261,188	209,352
Code size (Kb)	830.68	645.89

(b) Linux 2.4.31

Table 1. Static kernel characteristics

6. Experimental Results

We have implemented our ideas using the PLTO binary rewriting system for the Intel x86 architecture [20] and evaluated them using two different versions of the Linux kernel: versions 2.4.25 and 2.4.31. To get an accurate evaluation of the efficacy of this system, we begin with a minimally configured kernel where as much unnecessary code as possible has been eliminated by configuring the kernel carefully. For our experiments, we therefore configured the Linux kernel to remove modules, such as the sound card and video support, that are not required to run our benchmarks. We considered two versions for each kernel: one with networking support, the other without. The kernel code was compiled with *gcc* version 3.4.4 using the compilation flags of '`-Os -fomit-frame-pointer`', which instructs the compiler to optimize for code size. Table 1 gives various size-related statistics for the resulting kernel images. In order to simplify the booting process of the Linux kernel, we modified the kernel boot

Benchmark set	Programs	No. of unique system calls	No. of non-bootup system calls
Auto./Industrial	basicmath, bitcount, qsort, susan	33	9
Consumer	jpeg, mad, lame, tiff2bw, tiff2rgba, tiffdither, tiffmedian, typeset	46	11
Network	dijkstra, patricia (blowfish, CRC32, sha)	43	12
Office	ghostscript, ispell, rsynth, stringsearch	57	15
Security	blowfish, pgp, rijndael, sha	49	10
Telecomm	adpcm, CRC32, FFT, gsm	39	11
Entertainment	jpeg, lame, mad	43	10
Cellphone	blowfish, sha, CRC32, FFT, gsm, typeset	45	12

Table 2. Characteristics of the benchmarks used (from the MiBench suite [11])

up file `inittab` so that the Linux kernel will run in single user mode (level 1). Based on the profile data, there are 81 different system calls that are invoked during the booting process.

We used programs from the MiBench suite [11], a widely used and freely available collection of benchmark programs for embedded systems, to evaluate our aproach. The MiBench suite is organized into six sets of benchmarks, corresponding to different kinds of embedded environments: Automotive and industrial control, Consumer devices, Networking, Office automation, Security, and Telecommunications; each of these sets contains several different application programs. We augmented this with two additional sets: Entertainment, representing a multi-media consumer appliance for music and digital pictures; and Cellphone, representing a cell phone with security features. Characteristics of these sets are shown in Table 2. We also considered the BusyBox embedded toolkit [22], which was used by Chanet *et al.* to evaluate their kernel compaction work [6].

Before we can carry out kernel compaction for any given benchmark set, we have to identify the system calls that can arise from programs in that set. It is not enough to examine their executions using tools such as *strace*, since this may not cover all the execution paths in the programs. Nor is it enough simply to examine the source code of the benchmarks for system calls, since these actually call library routines that may contain additional system calls not visible in the source code. We therefore analyze statically linked binaries of the programs to ensure that we find all the system calls that may be invoked. This, however, causes the entire C library to be linked in. We address this problem by first carrying out a reachability analysis on the application program binaries to identify and eliminate unreachable library routines (using a conservative approximation to deal with indirect function calls) and then traversing the resulting whole-program control flow graph to determine the set of possible system calls. These data are shown in Table 2: the third column of this table gives the number of different system calls across all of the programs in each set of benchmarks, while the fourth column gives, for each benchmark set, the number of system calls not occurring in the set of system calls invoked during the kernel bootup process. Once we have the system calls that may be invoked by a set of programs, we use them to identify and eliminate unreachable code in the kernel.

Table 3 shows the effects of code compaction. For each benchmark set, we present three sets of numbers; these give the amount of compaction achieved for the `.text.init` section (the code used for kernel bootup), the `.text` section (the kernel code used during steady-state execution), and the total amount of code (`.text.init` and `.text` together). The mean overall code size reduction achieved for the Linux 2.4.25 kernel is 19.3% for the version with networking code and 23.75% for that without networking; for the Linux 2.4.31 kernel, these numbers are 22.4% for the version with networking code and 22.6% for the version without networking code.

The rows labelled "All system calls" in Table 4 show how much code compaction is achieved if all system calls in the kernel are assumed to be invokable by the application code. There are two conclusions that can be drawn from this. First, it is evident that our optimizations are able to achieve significant code size reductions (around 12%–16%), even on a carefully configured kernel, even if we make no assumptions about what system calls can be invoked by applications. Second, it can be seen that the ability to restrict the set of possible system calls, based on knowledge of the application code, can yield significant benefits, in our case giving an additional savings of 7%–9%.

Table 4 shows the effects of different optimizations on code size. For each optimization, we show both the incremental improvement obtained from adding that optimization to those listed above it in the table as well

Kernel	config.	Application set		.text.init section		.text section		Total	
				size (Kb)	reduction (%)	size (Kb)	reduction (%)	size (Kb)	reduction (%)
Linux 2.4.25	with networking	Original kernel		48.54	–	788.16	–	836.70	–
		All system calls		37.15	23.46	698.97	11.32	736.12	12.02
		† Busybox		37.15	23.46	648.95	17.66	686.10	18.00
		Mibench	Automotive	37.15	23.46	633.64	19.60	670.80	19.83
			Cellphone	37.15	23.46	637.71	19.09	674.86	19.34
			Consumer	37.15	23.46	639.48	18.86	676.63	19.13
			Entertainment	37.15	23.46	636.12	19.29	673.27	19.53
			Network	37.15	23.46	633.74	19.59	670.89	19.82
			Office	37.15	23.46	640.39	18.75	677.54	19.02
			Security	37.15	23.46	638.77	18.95	675.92	19.22
			Telecomm	37.15	23.46	634.25	19.53	671.40	19.76
		Geom. mean:			**23.46**		**19.03**		**19.29**
	without networking	Original kernel		43.43	–	598.18	–	641.41	–
		All system calls		34.35	20.90	514.28	14.03	548.63	14.49
		Busybox		34.35	20.90	465.43	22.19	499.78	22.10
		Mibench	Automotive	34.35	20.90	450.37	24.71	484.72	24.45
			Cellphone	34.35	20.90	454.43	24.03	488.79	23.82
			Consumer	34.35	20.90	456.21	23.73	490.56	23.54
			Entertainment	34.35	20.90	452.85	24.30	487.20	24.07
			Network	34.35	20.90	450.46	24.69	484.81	24.44
			Office	34.35	20.90	457.11	23.58	491.47	23.40
			Security	34.35	20.90	455.50	23.85	489.85	23.65
			Telecomm	34.35	20.90	450.98	24.61	485.33	24.36
		Geom. mean:			**20.90**		**23.95**		**23.75**
Linux 2.4.31	with networking	Original kernel		48.87	–	781.81	–	830.68	–
		All system calls		37.42	23.42	669.58	14.36	707.00	14.89
		Busybox		37.43	23.41	617.10	21.07	654.53	21.21
		Mibench	Automotive	37.43	23.41	601.64	23.05	639.06	23.07
			Cellphone	37.43	23.41	606.76	22.39	644.18	22.45
			Consumer	37.43	23.41	608.76	22.13	646.19	22.21
			Entertainment	37.43	23.41	605.32	22.58	642.74	22.62
			Network	37.43	23.41	602.12	22.98	639.54	23.01
			Office	37.43	23.41	610.41	21.92	647.84	22.01
			Security	37.43	23.41	608.67	22.15	646.10	22.22
			Telecomm	37.43	23.41	603.16	22.85	640.59	22.88
		Geom. mean:			**23.41**		**22.34**		**22.40**
	without networking	Original kernel		58.90	–	586.99	–	645.89	–
		All system calls		43.70	25.81	499.12	14.97	542.81	15.96
		Busybox		43.69	25.82	460.17	21.60	503.87	21.99
		Mibench	automotive	43.69	25.82	454.71	22.54	498.40	22.84
			cellphone	43.69	25.82	455.45	22.41	499.15	22.72
			consumer	43.69	25.82	456.75	22.19	500.45	22.52
			entertainment	43.69	25.82	456.17	22.29	499.86	22.61
			network	43.69	25.82	454.78	22.52	498.48	22.82
			office	43.69	25.82	456.30	22.27	499.99	22.59
			security	43.69	25.82	455.30	22.44	498.99	22.74
			telecomm	43.69	25.82	454.87	22.51	498.56	22.81
		Geom. mean:			**25.82**		**22.31**		**22.62**

† Corresponds to the experiments of Chanet *et al.* [6].

Table 3. Code compaction results

Kernel	*Optimization*	WITH NETWORKING		WITHOUT NETWORKING	
		Incremental benefit (%)	*Cumulative benefit (%)*	*Incremental benefit (%)*	*Cumulative benefit (%)*
2.4.25	unreachable code elimination	14.56	14.56	20.09	20.09
	whole function abstraction	3.16	17.72	2.80	22.89
	duplicate code elimination	0.92	18.64	0.39	23.28
2.4.31	unreachable code elimination	21.32	21.32	21.09	21.09
	whole function abstraction	0.03	21.35	0.10	21.19
	duplicate code elimination	0.81	22.13	1.14	22.33

Table 4. Effects of different optimizations on code size

Kernel	*Analysis*	WITH NETWORKING		WITHOUT NETWORKING	
		Unreachable code elimination (%)	*Overall code size reduction (%)*	*Unreachable code elimination (%)*	*Overall code size reduction (%)*
2.4.25	Conservative	0.00	4.98	0.00	4.52
	Address-taken	11.28	15.67	17.85	21.23
	FA analysis	14.55	18.76	20.09	23.28
2.4.31	Conservative	0.00	1.34	0.00	2.4
	Address-taken	12.10	13.26	16.37	17.78
	FA analysis	21.32	22.13	21.09	22.33

Table 5. Effects of different call target analyses on code size

as the cumulative benefit of these optimizations. It can be seen that by far the largest savings, some 15%–21%, come from unreachable code elimination. For the Linux 2.4.25 kernel, whole function abstraction gives an additional improvement of about 3%; the effects of duplicate code elimination, while noticeable, are much smaller, coming in at under 1%. For the Linux 2.4.31 kernel almost all of the code size reduction comes from unreachable code elimination: the effect of whole function abstraction is essentially negligible, while that of duplicate code elimination is only around 1%. The large difference in the effects of whole function abstraction between the two versions of the kernel arise from the fact that there are a number of distinct but identical functions in the 2.4.25 kernel that have been hand-optimized away in the 2.4.31 kernel. (Note that these optimizations can have overlapping effects, which means that these numbers can be different if the optimizations are considered in a different order. For example, we might see somewhat greater benefits from duplicate code elimination, and a little less from unreachable code elimination, if the former is carried out before the latter. This can happen because in such a situation, some of the duplication eliminated might be in code that turns out to be unreachable.)

Since unreachable code elimination accounts for most of our code size savings, the precision of program analyses for identifying unreachable code plays a significant role in the amount of code that can be eliminated. Determining the possible targets of indirect function calls plays a crucial role in this. Table 5 shows the effects of different indirect call target analyses on both the amount of unreachable code identified as well as the overall code size reduction achieved. We consider three different scenarios:

– *Conservative* refers to the case where no program analysis is done, and every function is conservatively considered a potential target for each indirect call.

– *Address-taken* refers to an analysis that computes the set S of functions whose address is taken somewhere in the program. The set of potential targets for any indirect call is then set to S. Variations of this analysis are commonly used to deal with indirect function calls in many binary rewriting systems.

– *FA analysis* refers to the source-level pointer alias analysis described earlier in this paper.

Not surprisingly, since no function is identified as unreachable in the conservative case, unreachable code analysis yields no savings; all the code size reduction in this case comes from whole function abstraction and duplicate code elimination. This yields code size reductions of about 4.5%–5% on the 2.4.25 kernel, but (because of fewer opportunities for whole function abstraction) only about 1.3–2.4% for the 2.4.31 kernel.

The straightforward address-taken analysis does surprisingly well, identifying about 11%–12% of the code as unreachable in the 2.4.25 kernel and 12%–16% in the 2.4.31 kernel; the overall size reductions achieved range from 13% to about 18%. FA analysis identifies about 14.6%–20% of the code as unreachable in the 2.4.25 kernel and about 21% of the code in the 2.4.31 kernel; this yields overall improvements of about 19%–22%. Because of the availability of higher-level semantic information, the source-level FA analysis is able to attain improvements over the address-taken analysis ranging from 2.5%–3% for the 2.4.25 kernel to over 9% for the 2.4.31 kernel with networking.

The amount of compaction achieved for each set of applications depends on the particular set of system calls made by the set of application programs. To get an idea of the extent to which our results might generalize to other sets of embedded applications, we evaluated the "popularity" of different system calls across the MiBench suite. The popularity of a given system call s in a set of programs P is given by the fraction of programs in P that use s. Intuitively, if different kinds of applications use very different sets of system calls, i.e., many system calls have low popularity, then our results may not generalize well; on the other hand, if different kinds of applications tend to have mostly-similar sets of system calls, then we can expect these results to generalize. The results are shown in Figure 5. It can be seen that out of some 226 different possible system calls in our system,[4] there is a small core of 32 system calls that are used by every program. Popularity drops off sharply outside this core set. All of our benchmark programs, taken together, refer to only 76 system calls, i.e., about a third of the total set of system calls.

This relative uniformity in system call usage across a wide variety of applications helps explain the surprising uniformity of our code compression results across all of the benchmark sets. While the applications themselves are very different in terms of their nature and code size, the popularity data shown in Figure 5 show that they do not differ from each other hugely in terms of their interactions with the operating system kernel: for example, they typically read data from some files, process that data, and write out the results. Moreover, in addition to the system calls made by application code, the kernel itself makes 81 different system calls during the bootup process. The overall result is that the set of "non-bootup" system calls arising in the application code is relatively small, and does not vary greatly from one benchmark set to another (Table 2, col. 4). Because of this, the compaction results for the different benchmark sets tend to be similar.

[4]There were 259 syscall entries in our version of Linux; of these, 33 were not implemented ("sys_ni_syscall"), leaving a total of 226. Other sources put the number of Linux system calls much higher: e.g., Wikipedia mentions "almost 300" system calls for Linux.

7. Related Work

The work that is closest to ours is that of Chanet *et al.*, who describe a system for code compaction of the Linux kernel to reduce its memory footprint [5, 6]. The overall compaction results achieved by the two approaches are quite similar: on the one data point where the two systems can be directly compared (BusyBox with networking, identified with '†' in Figure 3), Chanet *et al.* do somewhat better than us (18.9% reduction in text size, compared to the 18.0% we get). The techniques used by the two systems are quite different, however, and a detailed comparison of the two indicates different strengths for each. For example, in the context of unreachable code elimination, Chanet *et al.* use more detailed structural information about subsections in the input binary to do a better job of analyzing pointers between the code and data sections; our system, on the other hand, uses the source-level FA analysis to obtain more precise information about indirect call targets. (Subsequent work by Chanet *et al.* obtains additional code size reductions by compressing infrequently executed code [5]. These techniques, which use Huffman compression to store infrequently executed code in a non-executable format and rely on runtime decompression to restore them to an executable format when necessary [8], are orthogonal to those discussed here.)

We are not aware of a great deal of other work on binary rewriting of operating systems kernels. Flowers *et al.* describe the use of Spike, a binary optimizer for the Compaq Alpha, to optimize the Unix kernel, focusing in particular on profile-guided code layout [9]. A number of researchers have looked into specializing operating system kernel code [15, 18, 17]. These generally focus on improving execution speed rather than reducing code size and therefore use techniques very different from ours.

There is a considerable body of work on code compaction; Beszédes *et al.* give a comprehensive survey [4]. 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.

One of the technical issues of considerable importance in code compaction of an OS kernel is that of resolving the possible targets of indirect function calls. This problem resembles the problem of identifying the possible targets of virtual method invocations in object-oriented programs [2, 3, 10, 21]. Much of this work re-

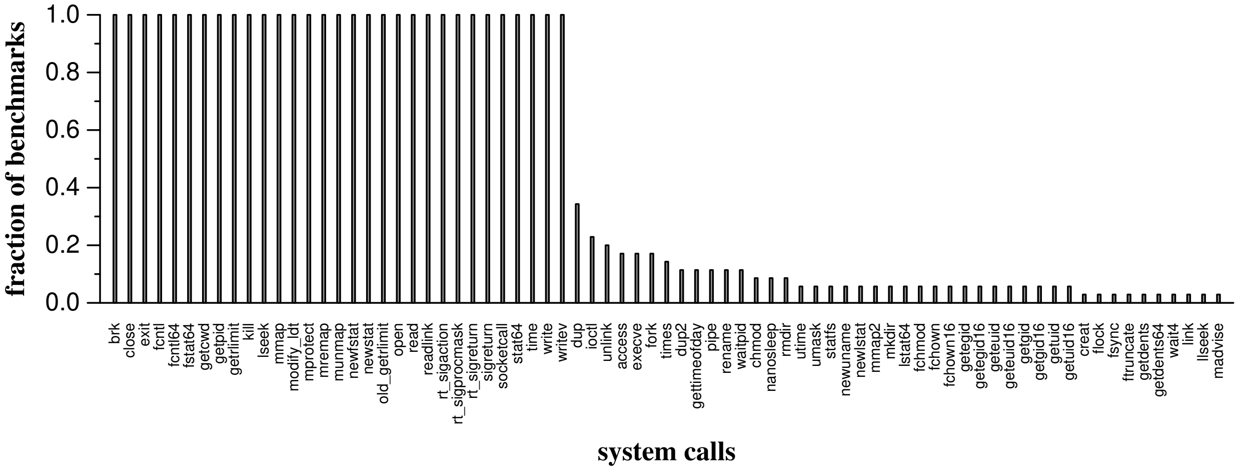


Figure 5. System call "popularity" in embedded applications

lies on using type information to resolve the possible targets of a virtual method invocation. We applied this idea of using type information to determine possible targets of indirect calls in the Linux kernel but the results were disappointing, because the available type information is not discriminating enough to give an acceptable level of precision (especially when taking into account the possibility of type casts on pointers). The FA analysis proved to be far more effective: a type-based signature matching analysis we implemented gave, on average, about 96 targets per indirect function call in the Linux kernel; with FA analysis, by contrast, the average number of targets per indirect call is 10.

8. Conclusions

Because of the limited amount of memory typically available in embedded devices, it is important to reduce the memory footprint of the code running on such devices. This paper describes an approach to code compaction of operating system kernels. We begin with the observation that embedded systems typically run a small fixed set of applications. This knowledge can be used to identify the minimal functionality required of the kernel code to support those applications and then to discard unnecessary code. We discuss a number of technical challenges that have to be addressed in order to make this work; in particular, we describe "approximate decompilation," which allows us to apply source-level program analyses to hand-written assembly code. Our ideas have been implemented in a prototype binary rewriting tool that is able to achieve a code size reduction of close to 24% on an already minimally-configured Linux kernel.

Acknowledgements

Discussions with Dominique Chanet were very helpful in improving our understanding of his work as well as improving our own.

References

[1] The Linux mobile phones showcase, February 2006. http://www.linuxdevices.com/articles/AT9423084269.html.

[2] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166. Springer, 1996.

[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 324–341, 1996.

[4] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, 2003.

[5] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Automated reduction of the memory footprint of the linux kernel. *ACM Transactions on Embedded Computing Systems*. To appear.

[6] 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.

[7] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[8] S. K. Debray and W. Evans. Profile-guided code compression. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, pages 95–105, June 2002.

[9] 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.

[10] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, pages 108–124, 1997.

[11] 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.

[12] M. Hind and A. Pioli. Which pointer analysis should I use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[13] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*.

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

[15] 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.

[16] 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.

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

[18] C. Pu *et al.* Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 314–324, Dec 1995.

[19] M. Rajagopalan, S. Perinayagam, H. He, G. Andrews, and S. Debray. Biray rewriting of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications*, October 2006.

[20] 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.

[21] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, pages 281–293, October 2000.

[22] N. Wells. BusyBox: A Swiss Army knife for Linux. *Linux Journal*, 78, October 2000.

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

[24] 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.