Automated Reduction of the Memory Footprint of the Linux Kernel

DOMINIQUE CHANET, BJORN DE SUTTER, BRUNO DE BUS, LUDO VAN PUT, and KOEN DE BOSSCHERE
Ghent University

23

The limited built-in configurability of Linux can lead to expensive code size overhead when it is used in the embedded market. To overcome this problem, we propose the application of link-time compaction and specialization techniques that exploit the *a priori* known, fixed runtime environment of many embedded systems. In experimental setups based on the ARM XScale and i386 platforms, the proposed techniques are able to reduce the kernel memory footprint with over 16%. We also show how relatively simple additions to existing binary rewriters can implement the proposed techniques for a complex, very unconventional program, such as the Linux kernel. We note that even after specialization, a lot of seemingly unnecessary code remains in the kernel and propose to reduce the footprint of this code by applying code-compression techniques. This technique, combined with the previous ones, reduces the memory footprint with over 23% for the i386 platform and 28% for the ARM platform. Finally, we pinpoint an important code size growth problem when compaction and compression techniques are combined on the ARM platform.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data Compaction and Compression—*Program representation*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Linux kernel, operating system, compaction, specialization, system calls, compression

ACM Reference Format:

Chanet D., De Sutter B., De Bus B., Van Put L., and De Bosschere K. 2007. Automated reduction of the memory footprint of the linux kernel. ACM Trans. Embedd. Comput. Syst. 6, 4, Article 23 (September 2007), 48 pages. DOI = 10.1145/1274858.1274861 http://doi.acm.org/10.1145/1274858.1274861

Author's addresses: Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere, Electronics and Information Systems Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: dchanet,brdsutte,bdebus,lvanput,kdb@elis.ugent.be

1. INTRODUCTION

Until some years ago, most embedded system developers could choose between either writing their own operating system from scratch, or licensing a proprietary kernel. Writing an operating system from scratch results in a long time-to-market, but full control over the system. Licensing a proprietary kernel ensures a shorter time-to-market, at the expense of having less control over the kernel. Moreover, the use of proprietary kernels risks vendor lock-in.

Recently, embedded system developers have, therefore, started to use open-source operating systems, such as Linux. With Linux, there is no risk for vendor lock-in. In addition, at least in theory, Linux can be customized for any specific embedded system without needing to write a kernel from scratch. In practice, however, where time-to-market enters the picture, this kernel customization is often limited to applying a number of patches that are circulating in the kernel developer community, to using the build-time configuration options of the kernel, and to specifying the wanted boot-time parameters.

1.1 Link-Time Rewriting

More than a decade of research into whole-program optimization has demonstrated that link-time binary rewriting can be a valuable add-on to existing tool chains that consist of compilers, assemblers and linkers. In particular, link-time program optimization, through binary rewriting, has proved to be useful to eliminate program inefficiencies resulting from separate compilation [Muth et al. 2001], to overcome overhead resulting from the use of modern software-engineering techniques, from obsolete code in aging code bases, and from copy-and-paste software development [De Sutter et al. 2002], and to minimize the residual code footprint [Debray et al. 2002; De Sutter et al. 2005, 2006].

Like any large software project, the Linux kernel suffers from the aforementioned overhead. In the context of embedded systems, this overhead is further increased by the general-purpose nature of the Linux kernel. For example, the built-in configuration options in Linux are not engineered for producing the smallest kernel, but rather for having a flexible, maintainable source code base that enables the kernel's deployment on a wide range of general-purpose systems, each consisting of one particular combination of hardware components that may even change as systems get upgraded. On many embedded systems and, in particular, on systems of which the hardware and software is fixed for their whole lifetime, this flexibility is not needed. Unfortunately, however, the overhead that comes with it is not easily omitted from a compiled kernel. For example, the kernel includes code to specify boot-time parameters, but the standard kernel configuration process offers no means for omitting this capability or for optimizing the code for specific boot-time parameters that are known at system design time. Also, one of a kernel's main tasks as a layer between general-purpose hardware and user-space applications, is to present an abstract view on the hardware capabilities to a very large range of applications that are most often unknown when the kernel is developed or compiled. In many embedded systems, by contrast, the applications to be executed on the system are known in advance and, hence, only limited capabilities should be

provided by the kernel. The unneeded capabilities that consist, for example, of unused system calls, constitute nothing but overhead if they are not eliminated from the kernel. Operating systems that have been designed specifically for use in embedded systems, like eCos and VxWorks, typically have much more fine-grained configuration options, allowing the developer to discard (almost) all of the unneeded capabilities.

In this paper,[1] we propose to use link-time binary rewriting to overcome much of the aforementioned overhead in the Linux kernel. In addition to the existing link-time compaction techniques that were developed for user-space applications, we propose the use of link-time static analysis techniques that exploit *a priori* information about the specific hardware and software of embedded systems. Because the proposed method is based on link-time rewriting, our method is not limited to systems on which user-space programs are written in a limited number of supported programming languages. Instead, any compiled user-space program can be handled, regardless of its source language. This includes special-purpose languages (and compilers) that may be used to develop specialized applications. Moreover, the proposed method can be employed even when the user-space software includes third-party applications of which only the object code is available, or of which parts are written in manually optimized assembler code.

Link-time static analysis, as any static analysis, has its limitations, such as suffering from a lack of precise points-to set analysis [Hind 2001]. As a result, some unreachable code in the kernel cannot be detected automatically with static techniques. For example, code for writing to read-only filesystems will not be eliminated because it is considered reachable through function pointer tables. Such tables occur frequently in the kernel, because they are the preferred way to access hardware drivers.

Moreover, there is a large amount of reachable code that is rarely executed. Because of its nature as a layer between general-purpose hardware and user-space applications, the Linux kernel contains a significant amount of code to deal with exceptional situations,[2] such as hardware failure, and with unexpected behavior of a system's operation environment, such as network connections being broken in the middle of data transfers. This code for handling exceptional situations is as large as it is today because it must provide the necessary functionality to keep the kernel running when problems occur and because it must be able to handle a large range of errors that can occur within any possible combination of hardware components and environment settings.

In many embedded systems, by contrast, such unexpected behavior can often, to a large extent, be excluded, because all the interoperating hardware is known. Also, it is often not crucial for the operation of the system to keep running and certainly not to keep running at full speed. Sometimes graceful reboots suffice instead, or a significant, temporary slowdown can be tolerated.

---

[1]A preliminary version of this paper was published at ACM LCTES 2005 [Chanet et al. 2005].

[2]When we refer to exceptional situations or to code that handles exceptional situations, we do not refer to the specific programming language construct of "exception handlers." When we need to refer to the latter, we will always do so explicitly by precisely using that term.

Thus far, we have not yet found a static analysis to detect the involved code for automatically handling exceptional situations. Therefore, we rely on coverage analysis to infer which code is really needed for normal operation scenarios and which is not. Once the important, executed code of a kernel is collected for a specific system through coverage analysis, our link-time rewriter separates it from the noncovered code, after which the latter is compressed to save memory space. By adding a decompressor to the kernel, it can decompress the necessary code and still handle exceptional situations when they occur. This allows us to further reduce the kernel's memory footprint without compromising its correctness. Obviously, the notion of normal operation can be extended to abnormal, but known and enforceable, scenarios. It is the developer who decides on the scenarios that are covered during the coverage analysis.

In short, as major contributions of this paper,

- we present an automated method to specialize the kernel for the whole system on which it will be deployed. This includes boot-time parameter specialization and unused system call elimination.
- We present a partially automated method to reduce the overhead of infrequently executed code and exception handling code under normal operation.
- We present a number of elegant ways to deal conservatively, yet aggressively, with kernel code peculiarities in a link-time binary rewriter. These peculiarities include, for example, the presence of a large number of hand-written, position independent assembler routines. Thus, we demonstrate, for the first time, that link-time rewriting should not be limited to conventional compiler-generated code, but that robust rewriters can be engineered that reliably handle unconventional code.
- We evaluate the proposed methods on two different platforms, i386 and ARM, revealing a very interesting, yet not studied, problem relating to compressed software image sizes.

All compaction and specialization techniques described in this paper are (at least partially) automated. While manual specialization techniques exist that can be more aggressive and broader in scope, we feel there are significant advantages to using an automated tool for kernel specialization. Obviously, the user of the automated tool need not have an intimate knowledge of the kernel's structure and source code. Furthermore, automated binary rewriting is much faster than manually changing the kernel's source code. Finally, there is the issue of maintainability: upgrading to a new kernel version means reapplying all manual specializations, which at least includes the process of checking and updating a set of patches. This process is particularly error-prone when code has been moved and restructured in between two versions of the kernel, rather than simply having been extended or fixed.

Finally, we should note that although the presented techniques are applied specifically to the Linux kernel in this paper, we believe very similar techniques to be more generally applicable.

1.2 Structure of the Paper

This paper is structured as follows. Section 2 provides some background on link-time program compaction, in order to facilitate the discussion of the techniques presented in later sections. Section 3 presents the static system-level specialization techniques that we developed for the Linux kernel. Section 4 presents the code coverage analysis and compression techniques to reduce the size of kernel code that is only executed under abnormal circumstances. Section 5 discusses how system code peculiarities found in the Linux kernel can be handled by a link-time rewriter. All proposed compaction and specialization techniques are then evaluated in Section 6. Section 7 discusses related work, and conclusions are drawn in Section 8.

2. BACKGROUND INFORMATION

This section provides the required background information on link-time binary code rewriting, in order to facilitate the presentation of our new techniques in later sections.

2.1 Link-Time Rewriting

Link-time rewriting, in general, consists of a set of analyses and transformations that are applied when a program's assembled object files are being linked. In order to enable linking, a standard linker needs to extract three types of information from the object files [Levine 2000]. *Relocation information* provides information about the temporary addresses used in the object file's code and data sections, and how these temporary (or relocatable) addresses in the object files need to be adapted (or relocated) once the object file sections get a place in the final program. *Symbol information* describes the correspondence between relocatable addresses and global entities in the code and data, such as procedures and global variables. Symbol information is used by the linker to resolve each object file's references to externally declared symbols, such as global variables or procedures. Finally, *alignment information* describes how each object file section should be aligned in the linked program.

For link-time rewriting, the exact same information is, of course, available and so it is this information on which more complex link-time analyses and transformations have to rely [De Sutter et al. 2005, 2006; De Bus 2005]. Relocation information is now used to detect all computable addresses in programs and, hence, to conservatively approximate the possible targets of indirect control-flow transfers. Symbol information is used to detect additional properties of compiler-generated code. For example, if a compiled procedure is defined by a global symbol, the compiler must have generated it in accordance with the calling conventions. Otherwise, it cannot expect callers from other modules to know how to call such a procedure.

The operation of our link-time rewriter is summarized in Figure 1. The rewriter reads all object files constituting a program, together with the executable produced by the standard linker, and the linker map file produced by that linker. The latter file describes where all sections from the object files are found in the final executable. Using this map file, our rewriter first relinks

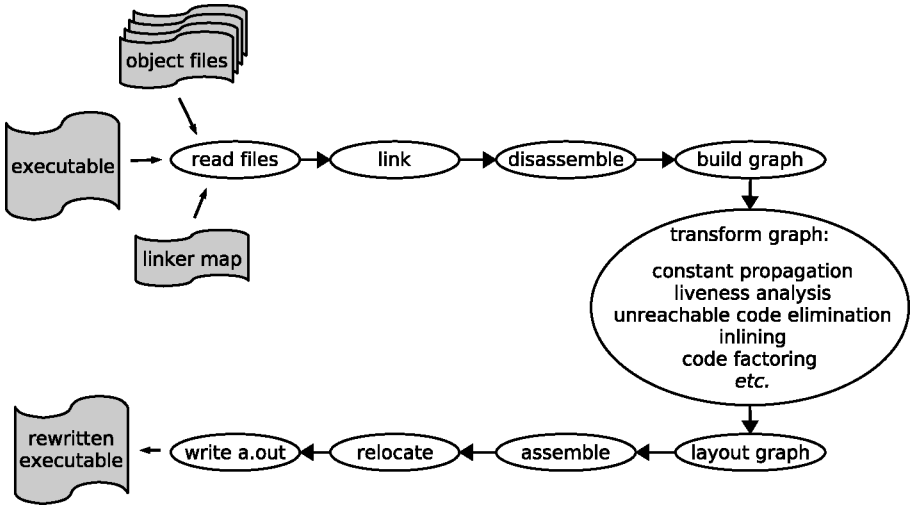

Fig. 1. Overview of the operation of our link-time rewriter.

the application in exactly the same way as the original linker. This way, the rewriter is able to collect all possible information on the executable, including the aforementioned information available in the object files, as well as any information added or used by the standard linker itself. Thus, we can guarantee that the rewriting operation is performed reliably [De Bus 2005; De Sutter et al. 2006, 2005] (see also Section 2.3). Next, the linked program is disassembled and a graph representation is constructed that is fit for program optimization. Once the diverse transformations, of which some are mentioned at the right of the figure, are applied on this graph, it is converted into a linear program representation again, after which the linear code is assembled. All addresses are relocated and the rewritten executable is stored on disk.

2.2 The Augmented Whole-Program CFG

Based on the information available in the object files, a suitable intermediate representation of the program to be rewritten needs to be constructed. Most link-time rewriters operate on a whole-program control-flow graph (WPCFG), which consists of the combined CFGs of all procedures in the program. In link-time WPCFGs, the intermediate instructions usually operate on registers as if they are global variables, and memory is treated as a black box.

To model indirect control-flow elegantly, a virtual *unknown node* is usually added to the WPCFG. As we mentioned in Section 2.1, relocation information informs us about the computable addresses in a program and, hence, on the potential targets of indirect control-flow transfers. The basic blocks at these addresses then become successors of the unknown node and basic blocks ending with indirect control-flow transfers become its predecessors. By imposing conservative properties on the unknown node, we are then able to handle unknown control-flow conservatively in any of the applied program analyses and transformations. For liveness analysis, for example, the unknown node is

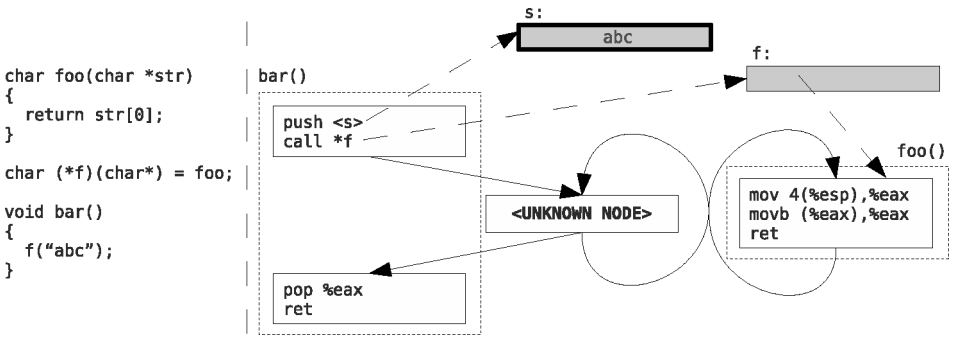

Fig. 2. On the left, an example source code fragment is shown, in which `bar()` calls `foo()` through a function pointer. On the right, the corresponding AWPCFG is shown. Solid edges are control-flow edges; dashed edges are data reachability edges. The gray blocks represent the data sections in the program; those with a thick border contain read-only data, which the others contain mutable data. In essence, `bar()` is a caller of the unknown node and `foo()` is a callee from the unknown node. In real programs, there are numerous such callers and callees. The first node of `bar()` references both `f` and the string `"abc,"` so there are data reference edges from this node to both data nodes. Because the variable `f` holds the address of `foo()`, a data reachability edge points from `f` to `foo()`. This edge corresponds to the control-flow edge from the unknown node to `foo()`.

defined as reading and writing all registers. An example of the use of the unknown node is depicted in Figure 2.

Instead of using a simple WPCFG, our link-time rewriter uses an augmented WPCFG or AWPCFG. Besides nodes modeling the program's basic blocks, the AWPCFG also contains nodes for all data sections in the object files, such as the read-only, zero-initialized or mutable data sections, and the global offset table section, etc. Furthermore, the edges in the graph are not limited to the control-flow edges that model possible execution paths. Instead the AWPCFG also contains *data reachability edges* that connect the occurrences of relocatable addresses with the nodes to which the addresses refer. For example, an instruction computing a relocatable address of some data section will be connected to the node corresponding to that section. Likewise, if the relocatable address of some data or instruction in node A is stored in a data section B, a data reachability edge from B to A will be present. As such, the data reachability edges model code/data that is reachable/accessible indirectly through computed jumps or indirect memory accesses. Figure 2 shows an example AWPCFG.

2.3 Reliability

The reliability of a link-time binary rewriting framework depends entirely on the reliability of its underlying program representation. The AWPCFG has to be conservative: it should represent at least all possible executions of the program. A number of recent publications [De Bus 2005; De Sutter et al. 2005, 2006] have described in detail how a conservative AWPCFG can be constructed by using the relocation information available at link-time and by using pattern matching. The latter is required for computed jumps, especially in hand-written assembly code or in position-independent code (PIC), because the behavior of such jumps is not defined in enough detail by the available relocation information. While

compilers typically do not produce this type of code, an assembly programmer can easily take a code address, add an unrelocated offset, and jump to the resulting address. This occurs, for example, in manually unrolled loops where the unrolled loop is entered somewhere in the middle through a computed jump because the desired number of iterations of the original loop is not divisible by the unrolling factor. Other patterns of PIC have also been observed.

To find the potential targets of such control-flow transfers, we rely on pattern matching. Whenever a use of the program counter or some unconventional control-flow transfer is detected, of which it is uncertain how it functions, the surrounding program fragment or program slice is compared to a number of patterns. To make this work reliably, the patterns to which a fragment is compared must be such that they each unambiguously define a specific behavior. For example, a pattern that matches address table lookups (that are commonly used in the implementation of C-language switch statements) should include the necessary boundary checks that check for constant values. Only if these boundary checks can be found and, consequently, the boundaries of the lookup table can be computed, it is possible to determine all potential targets of the computed jump.

More generally, the term "unambiguously" here means that a pattern should be such that the behavior of a matched fragment is known well enough to build a conservative program representation that is still precise enough to be useful. In other words, the constructed representation does not have to be an exact representation of all possible control flow, but only a precise enough, conservative estimate. When all matched patterns are unambiguous in this sense, the constructed graph of the program will be conservative and useful. Obviously, the degree of precision that is needed depends on the analyses and transformations one wants to apply.

When some code fragment cannot be matched to any unambiguous pattern, our link-time rewriter cannot build a program representation that is both conservative enough and precise enough to enable reliable and useful rewriting. Whenever such a fragment is found, the rewriter, therefore, informs the developer of this fact and aborts the rewriting. The developer then basically has three options. First, he can, of course, rewrite his program to the extent that it only includes matchable patterns. Obviously, requiring a developer to rewrite every program that contains unmatched fragments is not very user friendly, let alone automated.

Second, the developer can extend the set of patterns that are implemented in the link-time rewriter. Once a pattern is implemented, it can be reused for all programs to be rewritten. Finally, a developer can adapt the compiler, assembler, or linker in his tool chain to provide additional information on the generated code, that is used in the link-time rewriter. Again, the resulting tool chain can be reused for all of the developer's programs. To enable link-time rewriting of a program as unconventional as the Linux kernel, we have gradually implemented additional patterns and additional information provided by the compiler and linker, as discussed in Section 5.

As a result there are currently no unmatchable patterns in the Linux kernel or in any other program in our extensive regression test suite, which consists

of tens of programs compiled for multiple target architectures and runtime environments. One of the most important reasons for this is that under separate compilation code from one source code module, be it compiled or hand-written code, cannot refer to code in other modules without a description of at least some aspects of the reference through symbol and relocation information. Practically, this implies that all uses of, e.g., program counters in PIC, can only make the transformation of small pieces of code impossible. By treating that code as data that is not rewritten, the remaining code can still be transformed. We do this for some parts of the kernel, as described in Section 5.1.

The only exception to this property of separate compilation can occur when additional requirements are specified on how code fragments should be compiled and linked. Since the Linux kernel can be compiled with many different compiler and linker versions, this exception does not hold. Would it hold, however, it would be trivial to also inform the link-time rewriter of the additional requirements and to make it handle the resulting code conservatively.

2.4 Established Compaction Techniques

Unreachable code elimination is the simplest compaction technique that can be applied on the AWPCFG, by iteratively traversing reachable code in the WPCFG part of the AWPCFG. To obtain good results, this optimization needs to be performed context-sensitively such that only realizable execution paths are considered in which calls match returns. To eliminate inaccessible data from the AWPCFG as well, it suffices to apply a slightly adapted reachability analysis on the AWPCFG. In this adapted version, edges coming from the unknown node are only traversed after their corresponding data reachability edges were traversed. A more advanced version of this analysis was published by De Sutter et al. [2001].

The more fine-grained the data section nodes in the AWPCFG are, the more aggressive such inaccessible data removal will be. At link time, data section nodes in the AWPCFG can, in general, not be split into smaller nodes, because the compiler might have performed base-pointer optimizations on different pointers to the same section. There is an important exception to this observation, however, which concerns the format strings used for C-procedures, such as `printf`. These constant strings are collected in `.rodata.str` sections, and they are hence easily detected. Because the GCC compilers only generate direct accesses to these strings, any `.rodata.str` section of an object file containing multiple strings can be safely split into multiple sections and, hence, multiple nodes in the AWPCFG.

Besides unreachable code and data elimination, a number of more advanced control flow optimizations can be applied as well. These include duplicate code removal [De Sutter et al. 2002], inlining of small procedures or procedures with a single call site, and branch forwarding. The first of these detects whether multiple copies of a procedure or basic block are present in a program. If there are multiple identical procedures, all but one are eliminated, and calls to them are replaced by calls to the one remaining copy. If there are identical basic blocks, they can be outlined into a new procedure. The original occurrences of the blocks are then replaced by calls to the new procedure.

Next, there are a number of known data flow analyses and related optimizations that can be applied on the compacted graph. These include conditional constant propagation and interprocedural liveness analysis [De Sutter et al. 2005; Debray et al. 2002; Muchnick 1997]. As mentioned in Section 2.2, these data flow analyses analyze the use of registers as if they were global variables. In general, no analysis information is propagated about/through memory locations. There are three important exceptions, however.

Most importantly, symbol information allows us to determine which global procedures respect the calling conventions. Such procedures leave the callee-saved registers unchanged and, hence, we can propagate information from a call site of such a procedure to the corresponding return point or vice versa, even though the callee-saved registers may be temporarily spilled onto the stack in that procedure. Note that this symbol information is optional. It is not needed to detect procedures correctly, but only to derive additional information on their behavior. Second, when constant propagation is able to determine that some load instruction accesses a fixed memory location in a read-only data section,[3] the data at that address can be propagated into the program. Finally, when performing a simple local stack analysis as some kind of peephole optimization, one can remove redundant push and pop sequences within a single basic block. Such redundant instructions do occur even within a single basic block, because either the compiler lacked a whole-program overview, or other link-time transformations have made them redundant.

All mentioned techniques were previously studied in many user-space contexts [De Bus et al. 2004; De Sutter et al. 2001, 2002, 2005; Debray et al. 2002; Madou et al. 2004; Muth et al. 2001; Schwarz et al. 2001]. In order to apply them to a more complex, unconventional program, such as a kernel, some special precautions need to be taken; these are discussed in Section 5. First, the next two sections present a number of additional, kernel-specific specialization and compression techniques.

3. KERNEL SPECIALIZATION TECHNIQUES

Embedded devices often have known, fixed hardware and a fixed set of user-space applications. Examples of such fixed-function systems are the Linksys WRT54G wireless Internet gateway and the TiVo digital TV recorder,[4] both of which run the Linux kernel. In such cases, it is known *a priori*, which kernel functionality is required, and which is not.

The built-in configuration capabilities of the Linux kernel allow a user to select the required hardware drivers semiautomatically. While this driver selection can be done at a very fine-grained level, it can only be used to omit drivers (and some other functionality) from being compiled and linked into the kernel image. In most cases, this built-in configuration does not allow the remaining, selected parts of the kernel to be optimized for a selected configuration. For example, even though there may be no need to provide boot-time command-line

[3]This is possible because statically allocated addresses are propagated just like other numerical constants.

[4]http://www.linksys.com/ and http://www.tivo.com/.

parameters for some driver on a particular system, it is not possible to omit the code for handling such command-line parameters automatically. Hence there is no automated method for optimizing the driver for its default parameter values, let alone for other, fixed values. Now, while a user of a general-purpose computer might be interested in booting the kernel with different command-line parameters, this rarely is the case for embedded systems running Linux. On PDAs, mobile phones, and other such embedded systems, the user is most often not supposed to influence the boot process at all. Therefore, there is no reason to maintain the command-line kernel configurability for such systems or, indeed, to maintain the overhead that results from the lost optimization opportunities. On embedded systems running Linux, manual techniques are typically used to remove this overhead.

With respect to the software needs, a fine-grained configuration is not available in the standard distribution of the Linux kernel. For example, most of the system calls implemented in Linux cannot be omitted with the standard build-time configuration, even though they may not be needed on specific systems. The system calls include, for example, different versions of calls that correspond to different versions of the standard GNU C-library implementation. Such system calls are included for backward compatibility, but on a system with fixed software, including a fixed C-library, it is perfectly well known which versions of such system calls are required. Furthermore, most embedded systems are not as general-purpose as the standard kernel and, hence, embedded systems often do not require all the functionality that the kernel exposes to user-space through system calls. It should be noted that specialized, commercially supported distributions of the Linux kernel exist (e.g., LynuxWorks Bluecat and Montavista Linux) that allow more fine-grained configuration. However, this configurability is introduced manually, which means it has to be reintroduced, or at least updated, for every new release of the Linux kernel these distributions support.

In the remainder of this section, we propose three link-time kernel compaction or specialization optimizations based on a known software/hardware configuration and a fixed boot process. The major benefit of applying these techniques at link time is that they do not require the source code to be changed and that the specialization they offer, hence, does not complicate the maintenance of the kernel source. Furthermore, as the kernel-specific specializations are applied at link time, they can cooperate seamlessly with the existing link-time program transformations discussed in Section 2.4. All techniques discussed in the current section will be evaluated quantitatively in Section 6.

3.1 System Call Elimination

The first kernel specialization technique concerns the removal of unused system call handlers. In Linux, all system calls are identified with an integer number. Where a system call occurs in the code of a user-space application, this number is either encoded literally in the system call instruction or it is passed from user-space as the first parameter of the system call. The kernel then uses this number to index the system call handler table, from which it loads the address of the corresponding handler, to which control is then transferred.

Because a handler's (relocatable) address is stored in the system call handler table, our AWPCFG will include a WPCFG edge from the unknown node to the handler, together with a data reachability edge from the table to the handler. These edges keep the handler reachable in the kernel, even if the system call handler might not be called from within the kernel itself. To eliminate these edges from the graph, it suffices to nullify the handler's address in the system call handler table. As a result, if the handler is not reachable in any other way from within the kernel, it will become unreachable in the AWPCFG, and the unreachable code and data elimination discussed in Section 2.4 will eliminate it. If the handler is reachable in any other way, either because it is called directly or because its address is stored in some other data structure in the kernel, other edges in the AWPCFG will keep the handler reachable. In that case nullifying its entry in the system call handler table will not cause the handler to be removed from the kernel.

In short, the only additional feature needed to implement this specialization in a link-time kernel rewriter is the possibility to gather a list of unused system call numbers, to identify the table at the `$sys_call_table` symbol, and to nullify the unused entries.

To collect the list of system calls that can be eliminated, one has to analyze all programs that will be installed on the embedded system. For architectures like the ARM, where the number of the system call is literally encoded into the system call instruction, this is trivial: it suffices to find all system call instructions and disassemble them to generate a list of reachable system calls. On an architecture like the i386, where the system call number is passed in a register, constant propagation is needed to determine the value of this register at each system call instruction. To be conservative, we need to assume that all system call handlers are necessary as soon as the value of one system call cannot be determined by the constant propagation. In practice, we have found this not to be a problem, as a basic link-time constant propagation of register values was able to resolve all system calls in a large number of benchmarks that were linked against two different C libraries (glibc and uClibc) for the i386.

If not all user-space programs are known *a priori*, this automated specialization may still be useful. For many systems, the installed system libraries are known *a priori*. When applications are only allowed to perform system calls through these libraries, for example, for security reasons, it suffices to analyze the libraries.

3.2 System Call Specialization

After unused system call handlers have been removed, the remaining ones can sometimes also be specialized for known parameters. To add this feature to a binary rewriter, it again suffices to apply a constant propagation on all user-space applications, and to collect the known, constant parameters that are passed at system calls. This collection can be done along with the detection of used system calls, as discussed in the previous section.

In the link-time kernel specializer, the collected information is then read and constant propagation is slightly adapted to propagate the constant values,

rather than "unknown" values, into the system call handlers. In its simplest form, this can be achieved by inserting, at the system call-handler entry points, instructions that write the known parameters to their conventional location (being a register or a stack location used for parameter passing). As such, the constant propagation algorithm itself does not even need to be adapted.

System call specialization is mainly useful for removing argument validity checks from the system call handlers and for removing functionality from multiplexed system calls. A multiplexed system call is one that performs completely different actions depending on the value of one of its arguments, as, for example, the `ioctl` and `socketcall` system calls. If all possible values for the command argument are known, system call specialization can remove the code paths associated with the never-invoked actions from the system call handler.

3.3 Boot-Time Parameter Specialization

The Linux kernel is configurable at boot time through the so-called *kernel command line*. This command line, which is a string passed to the kernel by the boot loader, consists of a number of (parameter, value) pairs. These parameters, which correspond to kernel global variables, can sometimes be set at runtime, as well. In many cases, the user has no control over the boot process of an embedded system, and there often is no desire to ever change the values of these parameters at runtime.

Figure 3 shows the AWPCFG fragment corresponding to the kernel's implementation of this feature. The `parse_commandline()` procedure splits the command-line string in (parameter, value) pairs and passes them to the `process_arg()` function. In this function, the `param_handlers` table is scanned and, if a match for a parameter name is found, the appropriate handler is called with the parameter value as an argument. The handler (for example `set_debuglevel()`) then sets the corresponding kernel variable, to be used later on, during the execution of the kernel, for example, in `some_function()`.

There are two main specialization opportunities associated with this boot-time parameter feature. First, if the kernel command line that will be used on the device is known in advance and cannot be changed during the lifetime of the system, we can eliminate all unused parameter handlers from the `param_handlers` table. The user specifies the desired kernel command line to our link-time binary rewriter, which parses this command line and marks all entries in the `param_handlers` table that are needed for successful parsing of the command line. All other entries from the table can be removed. As a result, the AWPCFG data reachability edges to the superfluous parameter handlers disappear, together with their corresponding unknown control-flow edges. The handlers are thus unreachable from the entry point of the AWPCFG and can be removed by unreachable code elimination.

The second specialization opportunity involves specializing the kernel code for specific values of the configuration variables. If the value of a variable is known to be constant throughout the lifetime of the system, the link-time

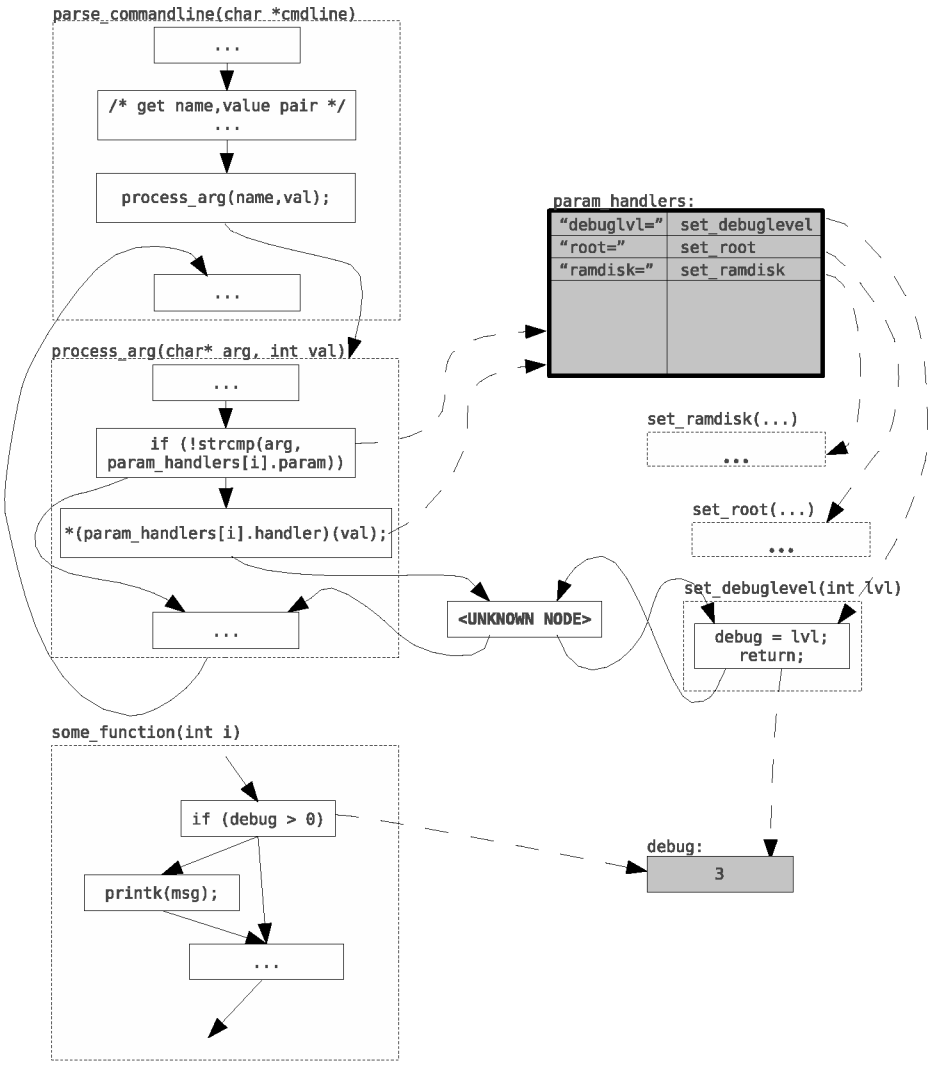

Fig. 3. Kernel command-line handling. Solid edges are control-flow edges, while dashed edges are data-reference edges. White boxes with a solid outline are basic blocks, while those with a dashed outline represent procedures (`parse_command`, `process_arg`, `set_debuglevel`, `set_ramdisk`, `set_root`, and `some_function`.) Gray blocks represent data sections; those with a thick outline contain read-only data, as in the case of the `param_handlers` table.

optimizations in our binary rewriter need not treat this value as unspecified. Instead, the kernel can be optimized for this known value.

As with the elimination of system calls, the user again has to provide a list with additional specialization information. In this case, this list identifies the kernel variables associated with the boot-time parameters and their fixed values. Using symbol information, the link-time kernel specializer looks up the memory locations of the variables, writes the specified values at those locations, and marks them as read-only. When constant propagation is then later

applied during the generic compaction of the kernel, the desired initial values are propagated into the program and, whenever possible, the program is specialized for those values. In the case of boolean variables, this typically results in compare instructions and conditional branches being removed or replaced by direct branches, thus eliminating unrealizable execution paths.

The only caveat relates to boot-time parameter variables whose value can change at runtime. For some of those mutable variables, such as the variables that determine the amount of debugging messages that should be printed, applying the proposed technique will not result in incorrect behavior. For other variables however, the incorrect assumption that a variable is constant may result in inconsistent or incorrect behavior. Fortunately, there is a simple necessary condition to test whether variables are mutable. After having eliminated all direct loads and stores that load from or store to the location of a command-line parameter, two situations can occur. On the one hand, it may happen that no relocatable address of the command-line parameter's data section occurs in the kernel anymore. That section will then be eliminated and the parameter has been proved immutable. In the other case, where the parameter's relocatable address still occurs in the program, we must conservatively assume that changes to the variable may occur at runtime and, hence, we should not specialize the kernel for that parameter (unless we know that such optimization is safe for that specific parameter of course.) In this case, the section of the parameter is not removed from the AWPCFG. The kernel specializer can detect this easily and it can inform the developer that he is trying to apply a potentially unsafe specialization, after which the user can opt for continuing or abandoning the specialization.

Suppose, for example, in Figure 3, that the user specified that the kernel variable `debug` has a fixed value of 0, and that the boot-time parameter `debuglvl`, which is associated with this kernel variable, will not be specified on the command line. Because `debuglvl` will not appear on the command line, its entry disappears from the parameter handler table. Consequently, `set_debuglevel()`, the parameter handler function, becomes unreachable and disappears from the kernel. The `debug` variable now only has one incoming data reachability edge, coming from a read operation in `some_function()`. The user has also specified that the value of `debug` should be 0, not 3, so 0 is written into this memory location, and, because there are no more write accesses to the variable, it can be marked read-only. Afterward, constant propagation can propagate the known value to specialize the code in `some_function()`, as the test can never evaluate to TRUE. The `if`-test and the `printk()` call can then be removed from the kernel by the constant propagation optimizations. This example illustrates how small specialization transformations (in this case, removing an entry from a table) interact with the established link-time analyses and optimizations to achieve the desired effect.

3.4 Initialization Code Motion

The first task of the kernel is the setup of the system and the initialization of a number of data structures and hardware devices. Most of the code and data

structures used during this initialization become useless afterward. However, unless countermeasures are taken, they keep occupying memory.

To avoid this, the Linux kernel developers annotate such initialization code and data and instruct the compiler and linker to put them into separate code and data sections, the so-called *init sections*. Once all initialization is done, the kernel releases the virtual memory pages on which the init sections reside, thus freeing the memory they occupied.

During the analysis and optimization phase of the link-time rewriter, all code sections of the kernel are joined in a single AWPCFG and compacted as a whole. Compaction techniques, such as code factoring and branch elimination can make it unclear whether a basic block should belong to the initialization sections or not. During the layout phase, when the control-flow graph is transformed into a linear representation, the link-time rewriter must, therefore, decide which code belongs in the init sections. It is important that no code ends up in the init sections by mistake, as that would mean it disappears from memory after initialization, while it may still be needed afterward. On the other hand, the optimizations should not cause too much code to be transferred from the init sections to the regular code section. Doing this may result in a smaller overall code size (which is good if optimizing the kernel image size on disk is the goal), but it would also result in a larger resident code size after initialization (which is bad if optimizing the memory footprint of the kernel is the goal).

Fortunately, most of the code that comes from the init sections can still be recognized and, with this knowledge, it is possible to identify other code that belongs in the init section as well. The code that can be executed after the initialization phase has ended and, hence, cannot be part of the init sections, is defined as all code in the WPCFG that is reachable from `free_initmem()`, the procedure that frees the init sections, through code that does not come from the original init sections. To detect this code and, hence, to detect all code that can be placed in the init sections, a simple iterative reachability algorithm suffices.

Besides finding the original code from the init section, this algorithm also finds code that can be placed in the init section, but which the kernel developers had not marked as such. Thus, the runtime footprint of the kernel will be reduced with this algorithm.

The existence of code that can be moved into the init sections at link time does not imply an oversight on the part of the kernel developers: some code may be called only in the initialization phase of the system in one specific configuration, while it may be called throughout the complete running time of the system in another configuration. An example of this is device initialization code. If the kernel supports hot-plugging of devices, the initialization code will be needed each time a device is plugged in. On the other hand, if the kernel does not support hot-plugging, the device initialization can only happen during system initialization, and the initialization code is no longer needed afterward. The annotation system used by the kernel developers to mark initialization code does not allow for conditional annotation based on the kernel configuration. Consequently, the kernel developers can only mark code that is initialization code in all possible configurations.

4. COMPRESSION OF UNEXECUTED CODE

After the application of the link-time optimization and specialization techniques described in this paper, a lot of code still remains present in the kernel that is not executed during "normal" operation of the system. On the one hand, there is unreachable code that could not be detected because of the limitations of static analysis and, on the other hand, there is code for handling exceptional situations. By performing a code coverage analysis on the kernel, it is possible to identify both types of code. It is impossible, however, to differentiate between them automatically, so it remains impossible to detect which part of this code can be removed from the kernel completely. Still, an important opportunity remains for minimizing the kernel memory footprint under normal operation, that resembles earlier work on the compression of rarely executed code.

Citron et al. [2004] call the code that is not executed during normal operation of a program *frozen code*. They have proposed a method for reducing the program's memory footprint by storing this code, and the data it accesses, in secondary memory (such as on a disk) and to load parts of it on demand. Debray and Evans [2002] take an alternative approach and propose to compress *cold code*—code that is executed very infrequently during normal operation, to keep the compressed form in memory, and to decompress code fragments into a fixed-size buffer whenever they have to be executed. To obtain high compaction ratios, their approach suffers from a significant slowdown.

For the Linux kernel, neither of these approaches are directly feasible. Kernel code is often more performance-critical than application code, so modifications to the kernel should not cause significant slowdowns. Furthermore, the kernel should work reliably at all times. Citron's approach fails in the reliability requirement. The code that they store in secondary memory is the code that deals with unexpected situations. What happens if some unexpected situation also causes this secondary storage to become unavailable? Debray's approach would keep the code in memory, where it is available under all circumstances. However, the approach of decompressing cold code in a fixed-size buffer has unacceptable worst-case behavior. Under exceptional circumstances, two cold code fragments might both become hot, after which they start to repeatedly evict each other from the decompression buffer. This would cause frequent calls to the decompressor and thus the system would be significantly slowed down. Furthermore, Debray's approach does not take into account the concurrency issues that arise in a multithreaded program, like the Linux kernel.

In the remainder of this section, we propose a compression technique that combines aspects of the two aforementioned approaches. In contrast to those approaches, however, our technique does function properly in the context of the Linux kernel.

4.1 General Overview

The frozen code in the kernel is identified by means of a code-coverage analysis and partitioned into single-entry regions. These regions are compressed and replaced by a stub that invokes a decompression routine and passes a pointer to the compressed code to it. This situation is depicted in Figure 4a. The

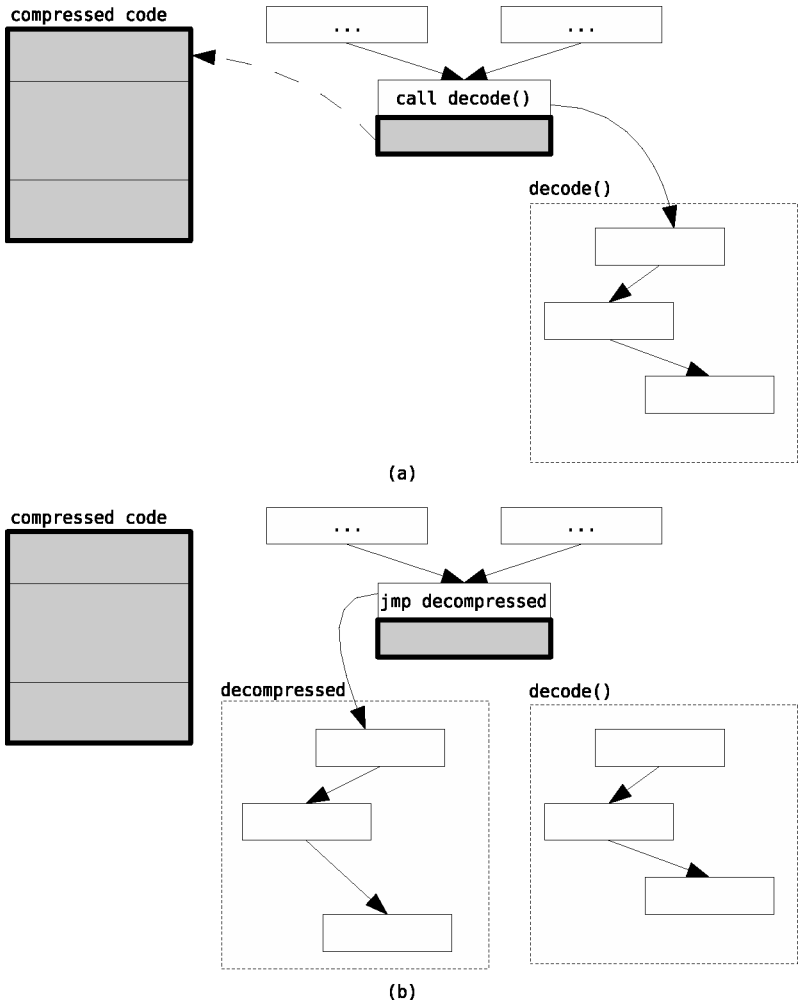

Fig. 4. A stub replacing frozen code (a) before decompression and (b) after decompression.

decompressor then allocates a buffer of the appropriate size through `kmalloc`, the kernel's dynamic memory allocation procedure, and decompresses the code into this buffer. The stub is then overwritten with a direct jump to the decompressed code so that subsequent invocations of the code no longer need to pass through the decompressor. Finally, the processor registers are restored to their state upon entry to the stub and control is transferred to the decompressed code. The situation after decompression is shown in Figure 4b. Obviously, this scheme can only reduce the kernel footprint when the decoder and the compressed code consume less space than the uncompressed code, and when most if not all of the compressed, frozen code need not be decompressed under normal operation.

When some frozen code regions need to be decompressed, however, the decompressed regions remain in memory forever. Any other eviction policy would require complicated locking schemes to make sure no thread in the kernel is

running the decompressed code while it is being evicted. The downside of this no-eviction strategy is that in the worst-case scenario, the kernel's memory footprint is still as large as without frozen-code compression. This can only be the case, however, if all frozen code was frozen because it is code for handling exceptional situations (and not because it is, in fact, unreachable, but was not statically detected as such), and if all possible exceptional situations occur between two consecutive reboots. We consider this to be unrealistic and, hence, do not take this situation into account in the remainder of this paper.

The upside of our approach is that the concurrency issues involved in frozen-code compression are significantly simplified, as we will discuss in Section 4.6, and that the performance impact of frozen-code compression is very small: the decompressor needs to be called, at most, once for each frozen-code region. After decompression, the processor's I-cache and D-cache need to be invalidated to ensure correct execution of the decompressed code. Subsequent invocations of the now unfrozen code are slowed down only by the extra direct jump they have to perform in the then updated stub to reach the decompressed code region. Under normal operation, no code should be decompressed, and no slowdown should be experienced whatsoever.

4.2 Frozen-Code Identification

The frozen code is identified by performing a code coverage analysis on the kernel. The target system is loaded with an instrumented kernel and subjected to several usage scenarios that the system developer considers "normal" operation. Of course, this may also include some "abnormal" scenarios that, although not normal, occur frequently enough for the developer to make them important scenarios. An example might be the disconnection of devices while they are being used.

The instrumented kernel keeps a record of which basic blocks were executed. After the analysis is finished, this information is extracted from the kernel through the `/proc/kcore` interface. All blocks that are not executed according to this information are considered to be frozen code. The effectiveness of frozen-code identification is entirely dependent on the thoroughness of the coverage analysis: if some important usage scenario is omitted, the associated code will be incorrectly considered frozen.

The instrumented kernel that is used for the code coverage analysis is produced by the same link-time binary rewriter as the one used to compact it. In practice, a statically allocated, zero-initialized array is added to the kernel, and each basic block in the AWPCFG is mapped to an element of this array. If a block is executed, a nonzero value is written into its element.

This means that extra code needs to be added to each basic block to overwrite its corresponding element in the coverage array. Obviously, adding this extra code should be done cautiously, as the correct functioning of the kernel must not be disrupted. We have to take care, for example, not to overwrite any live register contents in the added code. On CISC platforms such as the i386 platform, this is trivial: the address of the array element that needs to be written can be encoded directly into the store instruction, so we only need to add one instruction to each basic block: `mov $1, <address of array element>`.

On RISC architectures, like ARM, the situation is more complicated. Both the address of the array element and the value that has to be written into it have to be passed to the store instruction in registers. Fortunately, it does not matter which value is written into the element, as long as it is nonzero. By writing the value of the stack pointer, which is always nonzero, into the array element, we only need one free register in each basic block. For the ARM Linux kernel, the liveness analysis in our link-time rewriter found such a free register in 99.7% of all basic blocks. The address of the array element is then generated in this register in one or two instructions, after which a `str r13, [rX]` instruction performs the actual store.

For the remaining 0.3%, we derive (conservatively) whether they could have been executed by looking at the coverage information of their predecessors and successors in the AWPCFG. If none of a block's predecessors or successors were executed, the block itself cannot have been executed either.

As the decompressor calls the `kmalloc` and `kfree` procedures provided by the kernel, we need to make sure that these procedures are never considered frozen to avoid infinite loops. Fortunately, these procedures are so widely used throughout the kernel code that there is no risk of them being considered frozen code in the first place. We have also decided not to consider any code from the init code section as frozen code. While it is technically possible to compress the frozen initialization code as well, we consider this to be useless as this code is removed from memory after booting.

Note that we do not distinguish interrupt handling code from "regular" kernel code, so even the interrupt handlers will possibly be compressed. While the decompression process may cause an interrupt handler to respond slowly to the interrupt when it still needs to be decompressed, this will happen only once. Afterward, the code is already decompressed and subsequent occurrences of the same interrupt will be handled as quickly as without code compression. If a slow reaction to even one interrupt is unacceptable, the interrupt handler code has to be identified (either through the execution of more coverage scenarios, or otherwise manually), and excluded from the frozen code.

4.3 Frozen-Code Partitioning

After the frozen code is identified, it is partitioned into regions that form the basic units of compression. To minimize the amount of bookkeeping information related to the compressed code, we have opted to operate on single-entry regions. As such, one stub and one compressed code address suffice as bookkeeping code and data about each compressed region. If we would have allowed multiple-entry regions, we would need to keep track of the offsets of entry points in the regions as well. Furthermore, when a frozen-code region is decompressed, the decompressor has to overwrite all stubs leading to the region with direct jumps to the decompressed code. With multiple entry points, we would have needed multiple stubs per region and we would have had to keep a list of all stubs associated with each region, so they could all be overwritten immediately after decompression. Finally, the compression ratio obtained with the compression method we have implemented is independent from the partitioning method, so the single-entry requirement has no detrimental effect on the final code size savings.

The partitioning happens in three steps. First, frozen code is partitioned into single-entry regions per procedure. Each frozen basic block that has incoming AWPCFG edges coming from nonfrozen code is considered to be the start of a new region. Furthermore, all frozen code blocks that have incoming AWPCFG data reachability edges are considered to be the start of a new region as well. The latter allows us to relocate all data reachability edges that point to a compressed region to point to the corresponding stub instead. As the address of this stub is known at link time, this can be done without requiring any additional bookkeeping code or data at runtime.

In the second step, frozen-code regions are merged as much as possible, without violating the single-entry requirement. This implies that any region that has incoming edges from one other region only is merged with that region. As a consequence, frozen-code regions can cross procedure boundaries. The purpose of this step is to minimize the number of regions (and, more importantly, stubs) and to minimize the (static) number of interregion control-flow transfers. As the final address of the uncompressed frozen code is not known at link time, interregion control-flow transfers have to be encoded in binary code under the worst-case assumption that these transfers can span large displacements in the instruction memory, which is less efficient. The displacements of intraregion control-flow transfer are known at link time, by contrast, and as these are often rather small, they can be encoded more efficiently.

A side effect of the region-merging step is that, once runtime decompression is triggered, we potentially will decompress more code than strictly necessary. We consider this to be an acceptable drawback, however, as our main objective is to reduce the kernel memory footprint during normal operation, while still having a reduced, albeit not maximally reduced, size under abnormal operation.

We should note that this partitioning system is much simpler than the one proposed by Debray and Evans [2002] for cold code compression. Because their system operates with a small code buffer that stores decompressed code, they need to perform the partitioning of cold code in such a way that it balances the need for a small decompression buffer with the need to minimize the dynamic number of control-flow transfers between separate regions. If the latter would happen, they risk having to decompress previously decompressed, but evicted, code over and over again.

Finally, in a third step, we select the partitioned and merged frozen regions that will actually be compressed. In order to achieve a good overall compression ratio, it is important to compress only those regions for which compression actually brings size improvement. If a region is too small, or not very compressible, the possibility exists that the combined size of the stub and the compressed region is larger than the original, uncompressed region. As determining in advance which regions can be compressed profitably is an undecidable problem [Debray and Evans 2002], we resort to a heuristic approach to select the actual regions that will be compressed:

- All regions smaller than a certain minimum size will be excluded from compression. We determine this minimum size by assuming a fixed compression factor γ for all code. It then is profitable to compress a region if $K + \gamma \times S < S$,

with K the stub size and S the uncompressed size of the region. The minimum size S_0 for which this equation holds is $S_0 = K/(1 - \gamma)$.

- After the small regions are discarded, all regions are compressed. Any region for which compression turns out to be unprofitable is then deselected and will appear in the final kernel as uncompressed code. The reasoning behind this step is that the reason for the unprofitability of compression of these regions is probably the fact that they contain a number of uncommon (and thus hard to compress) instructions or instruction sequences.

Once the code is partitioned into regions, these regions are merged. The profitable ones are selected and are patched to remove all interregion fall-through control flow. This last step removes any memory layout dependencies between selected regions and, thus, simplifies the runtime decompression process.

4.4 Decompression Stubs

The stubs that replace the frozen-code regions invoke the decompressor with the address of the relevant compressed code. In order to keep these stubs as compact as possible, the call to the decompressor is performed without arguments, and the compressed code pointer is appended directly to the call instruction (the gray block in Figure 4a). As such, the pointer is located at the return address of the call to the decompressor, which can load this pointer by simply dereferencing its return address.

To ensure the correct working of the kernel, the values of the processor registers have to be preserved after decompression. To ensure this, first, the stub has to save all registers it changes on the stack. For an architecture like the i386, this is no problem, as the procedure call stores its return address on the stack and no registers are overwritten. On architectures like the ARM that store the procedure return address in a register, this so-called link register has to be stored on the stack first. As such, a stub on the i386 architecture would be 9 bytes large (5 bytes for the call instruction and 4 bytes for the compressed code pointer), whereas a stub on the ARM architecture will be 12 bytes large: 4 bytes for saving the link register, 4 bytes for the call, and 4 bytes for the compressed code pointer. Note that there is no need to include an instruction that restores the link register in every single stub, since this can instead be done in the decompressor itself, of which only one instance is present in the final program.

4.5 Code Compression and Decompression

This paper does not focus on finding the best software-controlled code compression technique, but rather on using compression in the context of the Linux kernel. As such, we have decided to use an established compression scheme rather than inventing our own. The code-compression algorithm we apply is basically the same as the algorithm described in Debray and Evans [2002], but we apply it on the i386 and ARM architectures instead of on the Alpha architecture. The data that needs to be compressed is a sequence of machine code instructions. Each of these instructions consists of an opcode field, followed by a number of instruction-specific fields (the presence of these fields depends on

the opcode). For each of the field types, a stream is created containing the field values for each instruction in the sequence. One additional stream is created that consists of the sizes of all frozen-code regions. For each of these streams an optimal Huffman code is generated.

Each individual frozen region is then compressed by first writing the Huffman coded version of its size, followed by the instruction sequence that makes up the region, whereby the value for every instruction field is replaced by its Huffman code.

Decompression is fairly easy. The decompressor first reads the size symbol from the beginning of the stream and decodes it. A buffer of the correct size is then allocated and the instructions can be decoded. This is done by repeatedly reading an opcode field from the stream and decoding it. Based on the opcode, the decompressor knows which field types will follow, so it can read the corresponding symbols from the stream one by one. This process is repeated until the whole allocated decompression space is filled.

After the code is decompressed, all interregion PC-relative control-flow offsets have to be recomputed. As the final address of a frozen-code region is only known at decompression time, it is impossible to compute, in advance, the interregion jump and call offsets. The most efficient way to perform this recomputation is to insert sentinel instructions (with an invalid opcode) into the instruction stream to indicate which instructions need to have their offsets recomputed. At link time, these offsets are computed as if the frozen-code region was located at address 0. During decompression, the correct offset can be computed by subtracting the final address of the decompressed code region from the precomputed offset.

4.6 Concurrency Issues

The Linux kernel is multithreaded, preemptible, and supports symmetric multiprocessing. For all of these reasons, it is very well possible that different threads have to access the decompressor or decompressed code concurrently. In order to guarantee correct operation of the kernel in all circumstances, some form of locking needs to be implemented. Our major concerns for the implementation of the locking scheme are correctness and performance. Because the decompressor needs to be preemptible to allow other interrupt handlers or higher priority threads to run during decompression, we need to minimize the locking used to prevent deadlocks and race conditions.

Consequently, our decompressor is implemented in such a way that is fully reentrant, allowing multiple threads to execute it concurrently. If this were not the case, the complete decompressor would need to be locked, after which priority inversion[5] could occur. This is obviously undesirable.

Having a reentrant decompressor that can be executed by multiple threads at the same time, however, opens the door to possible race conditions. More precisely, two race situations can occur.

[5]Priority inversion occurs when a low-priority thread holds a shared resource that is required by a high-priority thread. This forces the high-priority thread to wait until the low-priority thread is finished with the resource, effectively inverting the relative priorities of the two threads.

First, it is possible for one thread to execute the first instruction of a stub while the decoder in another thread is concurrently overwriting this same stub. This could cause the first thread to execute a partially overwritten instruction. The solution is to overwrite the stub atomically, ensuring that no thread can ever see a partially overwritten instruction. On the ARM platform, atomically overwriting the stub is trivial as we only need to overwrite the first four bytes of the stub, and it is always possible to do a four-byte atomic write. On the i386 platform, the jump instruction we need to write is 5 bytes long (1 byte for the opcode, 4 bytes for the jump offset), so one atomic write seems insufficient. Adding locking to each stub is unacceptable as well, as this would at least double the size of the stubs. Fortunately, other solutions exist: on a single-processor system, it suffices to disable the processor interrupts while the stub is being overwritten, so that no other thread can interrupt the overwriting code. However, on multiprocessor systems, this is not enough, as threads on other processors might still see an inconsistent stub state. A workaround exists, at the cost of increasing the size of each stub with 5 bytes: we just add a jump to the next instruction at the beginning of each stub. Because the displacement of this jump is 4 bytes long, this displacement can be overwritten in one atomic operation. When the stub is first executed, this prepended jump acts as a no-op. As soon as the displacement is overwritten, the jump instruction functions as the jump into the decompressed code.

The second possible race condition occurs when two threads concurrently enter the decoder from the same stub. This means the same compressed region will be decompressed twice and the stub will be overwritten twice. While this is a wasteful situation in which code is unnecessarily decompressed multiple times, it does not cause incorrect behavior. We can avoid this situation by creating one lock per region. The decoder then acquires this lock before decompressing the region and, if a second instance of the decoder is invoked for the same region, it only has to wait until the lock is released and then jump back to the (now overwritten) stub. While this is the best solution for performance, it requires keeping a lock for each region, which wastes a considerable amount of space only to guard us from possibly (but improbably) wasting space at some point in the future. Therefore, we have opted for a different solution that is somewhat slower, but does not require a lock for each region. The region is always decompressed into a fresh buffer, but before the stub is overwritten the decoder checks whether some other thread has already done so. If this is the case, the race condition has occurred and the current thread releases its decompression buffer (via the kernel's built-in `kfree` procedure) and jumps to the already overwritten stub. In order to avoid introducing new race conditions, the checking and overwriting of the stub is protected by a lock.

4.7 Section Placement

On architectures like the i386 and the ARM that do not have a centralized global offset table [Srivastava and Wall 1994], explicit addresses appear in the code nearby or in the instructions that use the addresses. On the i386, they appear as immediate operands in the instruction. On the ARM, they appear in data blocks that are intermingled with the code. Addresses that appear like this in

```
PlaceSections(0)
S = ComputeCompressedSize()
S' = 0
while S > S':
     S' = S
     PlaceSections(S)
     S = ComputeCompressedSize()
PadCompressedSection(S' - S)
```

Fig. 5. Section placement algorithm in the presence of a compressed code section. PlaceSections(x) assigns addresses to all sections, assuming size x for the compressed code section. ComputeCompressedSize() adjusts the addresses in the frozen regions and computes the compressed code section size. PadCompressedSection(x) appends x bytes to the section.

frozen-code regions have to be compressed as well. This leads to an interesting phase-ordering problem: the final size of the compressed code region depends on its contents and, thus, on the addresses that appear in the region. However, these addresses depend, in turn, on the size of the compressed code, as the placement of the code and data sections following the compressed code section depends on the size of this section.

We solved this problem with the algorithm described in Figure 5. This algorithm iteratively places all sections based on an estimate of the compressed code section size. After each placement round, this estimate is recomputed. The algorithm stops when the new estimate is lower than the previous estimate. At this point, the compressed code section will fit in the space that is reserved for it. The section is then padded to fill the complete reserved space in order to ensure that the addresses do not change afterward when the binary image of compacted kernel is generated.

5. KERNEL CODE PECULIARITIES

In most previous link-time rewriting research, a number of assumptions are made about the code to be rewritten. For example, it is often assumed that only a limited number of computations take place on code addresses and that these computations are annotated with relocation information. While such assumptions often hold for conventional, compiler-generated code, they do not necessarily hold for manually written assembler code.

As the lowest layer in the software stack of an embedded system, the operating system kernel needs to work directly with the hardware devices. As such, the kernel needs to perform many operations that are not easily described in higher-level programming languages. Consequently, the kernel contains a lot of manually written assembler code.

This section presents an overview of the unconventional behavior of that assembler code and of other kernel code peculiarities and of the countermeasures that need to be taken to handle the kernel code conservatively during link-time rewriting, yet allow aggressive compaction.

5.1 System Initialization Code

The operating system kernel begins execution in a very early stage of the system boot process, when the booting system is not yet fully initialized. On systems

with virtual memory support, one of the remaining initialization tasks is to turn on the memory management unit (MMU) of the processor. Before this happens, all code runs in the physical address space. All code that runs after the MMU is enabled runs in a virtual address space.

Ordinary linkers typically do not support two different address spaces in the same program. This problem is circumvented by the Linux kernel developers with some clever manual assembler programming. In particular, the pre-MMU code is written in assembler, and all addresses appearing in this code are manipulated to trick the linker into producing the correct physical addresses, even though it is unaware of the different address spaces. This trickery exploits a deep knowledge of the internals of the linker being used (the standard GNU linker ld), and it only works because ld does not perform any complex analyses and transformations.

Unlike the simplicity of the GNU linker, a link-time program rewriter is not limited to relocating addresses in the generated executable. Instead, it will also try to optimize the address computations. Consequently, the assembler code manipulations used to trick the standard linker into generating the correct addresses for this pre-MMU code will no longer work. Instead, they will confuse a standard link-time optimizer and result in faulty optimization of the address computations. To circumvent this, countermeasures need to be taken.

Fortunately, the amount of code that is executed in the physical address space is small compared to the other code. For example, on the ARM platform it is only 540 Bytes. Moreover, the code executed in the physical address space is easily identifiable. As such, the simplest way to deal with this problem is to exclude this code from all optimizations by simply treating it as a data section in the AWPCFG. Because of the relatively small amount of code involved, the negative impact on the obtained compaction results is negligible.

5.2 Manually Written Assembler Code

Besides the physical address space initialization code, there are numerous other occurrences of manually written assembler code in the kernel. Procedures written in assembler code do not always adhere to the calling conventions or the application binary interface of the target platform, even though they may be exported to other source code modules. This is the case when all call sites of a manually written assembler procedure are written in assembler as well. In such cases, the kernel developers have full control over the parameter-passing mechanism that they want to impose. When such developer-imposed conventions differ from the standard conventions, the involved, exported procedures violate the assumption put forth in Section 2.4, namely, that exported procedures always respect the architecture's calling conventions.

In theory, there are three ways to conservatively treat such unconventional assembler code. The simplest option is to neglect the existence of calling conventions altogether. If no program analysis assumes that calling conventions are maintained, no analysis will produce incorrect results where the conventions are not maintained. This option is not viable, however, because there are many cases in which assumptions about the calling conventions do yield useful

information, such as with the propagation of data-flow information of callee-saved registers, as mentioned in Section 2.4.

The second option to deal with code that does not maintain calling conventions consists of using code inspection to detect such code. After this detection, the detected fragments can then be differentiated from conventional code in all program analyses. In practice, we do not find this to be a viable option either, because detecting whether or not a procedure's stack behavior respects the calling conventions would be either very complex (because of the problems of aliasing memory accesses [Debray et al. 1998]) or too imprecise [Linn et al. 2004], depending on the target architecture for which the kernel is compiled.

This leaves us with the third option, in which the compiler informs the link-time rewriter of any manually written assembler code. This requires patching the compiler tool chain with which the kernel is compiled, but, fortunately, the required patch is extremely simple.

For the GCC tool chain, for example, a three-line patch to GCC's `specs` file (that specifies the configuration of the tool chain) forced the GNU compiler to add two labels to the generated object code for each piece of inline assembler code. In particular, a label `$handwritten` is now added at the beginning of all inline assembler code fragments in the generated object files and a label `$compiler-generated` is added at the end of each inline assembler fragment.

Furthermore, each object file produced by the GCC tool chain for ELF targets contains the name of the source file it was generated for. Hence, full assembler files (such as `head.S`) can be detected at link-time by looking at the extension of the source code file name (".S" or ".s").

During the link-time rewriting of the kernel, each procedure of which the labels or file name in the object files indicate that it contains manually written assembler code is treated as an unconventional procedure, i.e., a procedure not respecting the calling conventions. Note that these unconventional procedures can still be analyzed and optimized, the link-time rewriter just assumes they do not adhere to the calling conventions, and, hence, computes less precise, but still conservative, data-flow information on them.

In the Linux kernel configured for our ARM test platform, this solution allows us to assume that 2074 out of 4846 procedures respect the calling conventions. For our i386 test platform, we can assume the same for 1757 out of 5214 procedures.

5.3 Memory-Mapped Input/Output

In many cases, the kernel communicates with peripheral devices by means of memory-mapped input/output (I/O): by writing to or reading from special memory locations the kernel can give commands to or read data from these devices. Obviously, memory accesses to these memory-mapped I/O addresses have very different properties from accesses to regular memory. For example, two successive reads from the same memory location without an intervening write are usually expected to return the same value. For memory-mapped I/O addresses this is not necessarily the case. This means that optimizations like load–store forwarding that reorder or even remove memory operations are not allowed on memory accesses that implement memory-mapped I/O.

In practice, it is virtually impossible to distinguish regular memory accesses from I/O memory accesses at link time. Only memory accesses relative to the stack pointer (or to registers whose value was derived from the stack pointer) are guaranteed to be regular memory accesses. Consequently, we modified our link-time rewriter to only perform memory access-related optimizations on memory operations of which we know it is safe.

For a more fine-grained approach to this problem, it would be useful to modify the compiler so that it provides annotations about which memory accesses are "unsafe." The programmer has to provide this information to the compiler (for example, through the use of the `volatile` keyword in C) to inform the compiler on the memory accesses it cannot optimize safely. In theory, it would be possible to make the compiler pass this information to the linker. Of course, even then memory accesses in manually written assembler code can still not be optimized at link time as there is no guarantee the programmer has provided the necessary annotations there. However, as discussed in Section 5.2, this case can be limited to detectable code fragments. We have not implemented such a solution as we estimate that the possible gains will be overshadowed by the amount of work necessary to modify the compiler.

5.4 Special Instruction Sequences

Besides special privileged mode instructions that do not occur in user-space applications, the Linux kernel contains some sequences of seemingly innocuous instructions that require special treatment. Usually, these sequences depend on the micro-architectural side-effects of instructions to influence the processor operation on a level that is normally hidden from the application programmer. For the i386 kernel there are two such sequences:

- writing to the processor's control registers:

```
                mov %cr0, %eax
                orl $0x80000000, %eax
                mov %eax, %cr0
                jmp <next>
<next>:         ...
```

  This instruction sequence enables the memory management unit on the processor, switching the execution context from physical to virtual address mode. The first three instructions set the paging bit in the appropriate control register. The jump instruction flushes the processor's prefetch queue to ensure all subsequent instructions are interpreted in the new processor context. This jump does not alter control flow and only appears in the sequence because of its side effect. A regular link-time optimizer would not take this side effect into account and would remove the jump from the program. We modified our link-time rewriting framework to let it check whether or not such a useless jump is preceded by an instruction that alters a control register. If this is the case, the jump instruction is marked as having a side effect and will never be removed.

- the BUG sequence:

```
ud2
<source code line number encoded as 2-byte int>
<pointer to string containing source code file name>
```

  This sequence is used to signal bugs in the kernel code. The ud2 instruction causes an "undefined instruction" exception. The exception handler then uses the address of the instruction that causes the fault to locate the source code file and line number information that is printed on the console, after which execution is terminated.
  There are no explicit references to the source code information immediately following the ud2 instruction, so normally the link-time rewriter would consider this data to be unreachable and remove it from the kernel. We have adapted our link-time rewriter so that it adds a data reference edge from the ud2 instruction to the source code data. As long as the instruction is reachable, the data will remain reachable as well. During the control-flow graph layout phase, special care is taken to place the data immediately after the ud2 instruction.

For the ARM, there are a number of special sequences:

- the cpwait instruction sequence:

```
mrc p15, 0, r1, c2, c0, 0
mov r1, r1
sub pc, pc, #4
```

  This instruction sequence makes sure that after a write to the system control coprocessor (the equivalent of the i386's control registers), all following instructions are interpreted according to the new processor state. The first instruction reads some random value from the coprocessor and the second instruction forces the processor to stall until this read is completed, thus ensuring that the coprocessor write instruction is completed before execution continues. In other contexts, the second instruction would be useless as it does not change the visible processor state. The third instruction executes a jump to the next instruction, which flushes the processor pipeline. Again, this instruction would normally be considered useless as it does not alter the control flow.
- the cpwait_ret sequence combines cpwait with a function return:

```
mrc p15, 0, r1, c2, c0, 0
sub pc, lr, r1, lsr #32
```

  The mov and sub instructions from the previous sequence are now combined into a single instruction that subtracts 0 from the link register lr (which holds the return address) and stores the result in the program counter pc. A good link-time optimizer would note that shifting a 32-bit value right over 32 positions results in 0 and would replace the sub instruction with mov pc, lr. By removing the dependency of this instruction on r1, the

processor would no longer need to stall, resulting in potential execution errors.

- D-cache initialization on the PXA250 processor:

```
            bic r0, pc, #0x1f
            add r1, r0, CACHESIZE
<loop>:     ldr r2, [r0], #32
            cmp r0, r1
            bne <loop>
```

The first instruction copies the program counter in `r0` and zeroes the lowest 5 bits. Subsequently a loop is executed that loads some values into `r2` that will never be used, which makes the load instruction in this loop a prime candidate for elimination after register liveness analysis is performed. The real purpose of this loop is to initialize the data cache on the processor and this effect would be lost if the load instruction were eliminated from the kernel.

Again, we have modified our link-time optimizer to recognize these sequences and mark them as having side effects, thus ensuring that they will not be altered during the optimizations.

5.5 Page-Fault Handling

A number of code fragments in the Linux kernel need to access data at addresses that are passed from user-space through systems calls. These accesses can potentially cause page faults. To handle such page faults, the Linux kernel includes so-called *fix-up* code fragments. A page-fault handler table contains the addresses of all instructions that can perform a data access at an address passed to the kernel from user-space—it is these accesses that can cause page faults—together with the addresses of the corresponding fix-up code fragments that need to be executed to handle a page fault. When an actual page fault occurs, the address of the corresponding fix-up is looked up in the table through a binary search.

With this scheme, the addresses of the user-space memory-access instructions are stored in the data sections of the kernel. As a result, these instructions would normally be made successors of the unknown node in the AWPCFG, as is shown in Figure 6a. Clearly, this would be overly conservative: the instruction's addresses will only be used in comparisons but not as jump targets. Instead, additional control-flow transfers can occur *after* these instructions, in case a page fault actually occurs.

To model this correctly, it suffices to remove the edges from the unknown node to the memory access instructions and to add edges from the memory-access instructions to the unknown node instead, together with edges from the unknown node to the instruction *following* the memory accesses. This models the fact that unknown control flow may occur after a faulting memory access and that control may return to the instruction following the faulting instruction. The correct modeling is shown in Figure 6b.

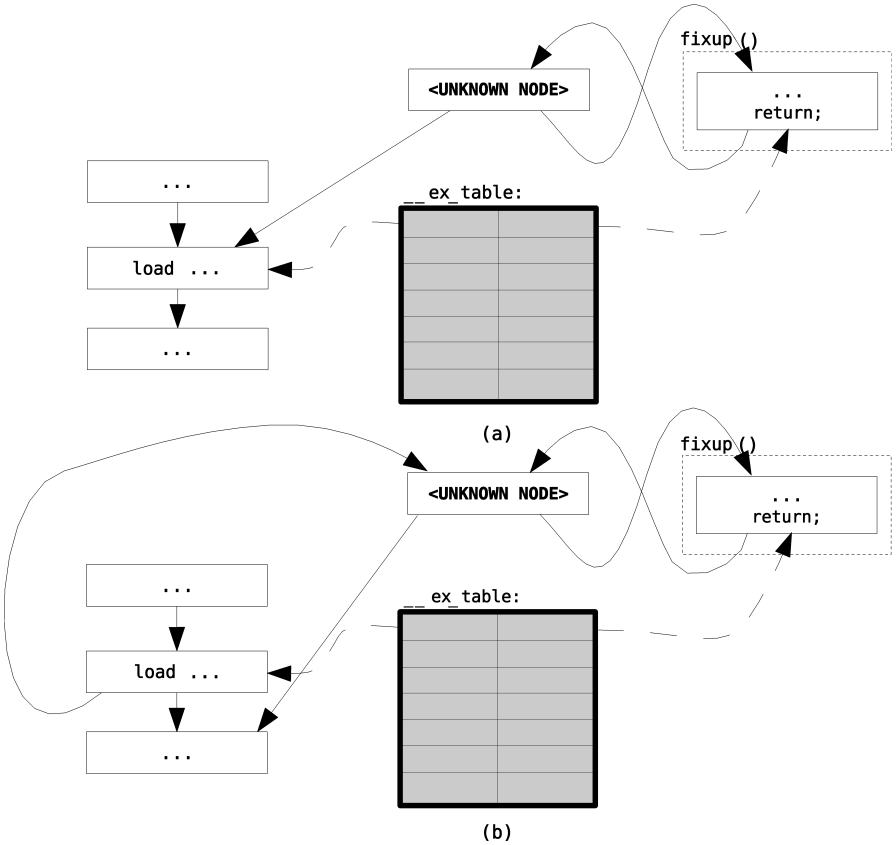

Fig. 6. Page fault handling system for kernel-mode accesses to user-space memory. Part (a) shows the way our link-time rewriter by default adds unknown control-flow edges, part (b) shows the correct modeling of unknown control flow in this case.

This solution is simplified by the fact that the page-handler fault table is stored in a separate and, hence, easily identified, data section named `__ex_table`.

If frozen-code compression is applied to the kernel, extra precautions need to be taken with regard to the `__ex_table`. As mentioned before, whenever a page fault occurs, the address of the faulting instruction is looked up in this table *through binary search*. This implies that (1) the data reference edges in the AWPCFG should *always* point to the actual memory-access instruction and should not be moved to a compression stub and (2) because of the binary search, the entries in the `__ex_table` should always be sorted. These two implications make it impossible to compress any such memory-access instruction. If the address of the instruction is not known at link time, it is impossible to correctly sort the `__ex_table`. Moreover, moving the data reference edge to the stub, whose location is known at link time, is explicitly disallowed, in this case. For these reasons, basic blocks with incoming edges from the `__ex_table` section are explicitly excluded from the partitioning of frozen and executed code.

6. EXPERIMENTAL EVALUATION

To evaluate the proposed compaction and specialization techniques, we have implemented them on top of Diablo[6] [De Sutter et al. 2006; Madou et al. 2004; Anckaert et al. 2004], a link-time rewriting framework developed in our research group. This section discusses the results obtained for a case study based on two target systems, an ARM and an i386 system, which were both set up as embedded web servers.

First, the test systems will be described in some more detail. Then, the impact of the techniques described in this paper on the size of the kernel will be studied. It turns out that the techniques have some unexpected effects with regard to the gzipped kernel image size, which are discussed as well. We will then look into the performance effects of the kernel modifications. Finally, the system requirements for our kernel optimizer are discussed.

6.1 Two Evaluation Systems

Our i386 system has a Pentium III processor, with 64 MiB of RAM, an IDE hard disk, and a Fast Ethernet network card. The Linux kernel is a vanilla 2.4.25 kernel, configured without module support and with only the necessary drivers. Compilation was done with GCC 3.3.2.

Our ARM system is an Intrynsic CerfCube 255, with a PXA255 XScale processor, with 64 MiB of RAM, 32 MiB of flash storage and a Fast Ethernet network connector. The Linux kernel is a 2.4.19 kernel, with patches supplied by the device manufacturer. The kernel is also configured without module support and with only the necessary drivers. Compilation was done with GCC 3.2.

For both kernels, the compiler was instructed to optimize for code size (`-Os`). All other build options were left at their standard values. Both systems have an identical user-space configuration based on Busybox[7]. This is a so-called multicall program that performs different functions depending on the name with which it is called. It is used in almost all embedded Linux systems because it provides a very compact, but complete user-space environment. On our test systems, Busybox acts as `init`, as a shell, as a `vi`-like editor, as a number of other necessary system utilities, and even as a web server. Busybox is statically linked against uClibc, an embedded C-library that is also engineered for small code size.

6.2 Compaction Results

To evaluate the proposed techniques, we have applied our kernel compactor several times, with additional techniques applied on successive runs. This started with a very simple compaction that only performs a reachability analysis on the AWPCFG and ends with a version that applies all the compaction and specialization techniques mentioned in this paper, including the compression of frozen code. The resulting kernel sizes and the obtained reductions are depicted in Table Ia for the i386 system and Table Ib for the ARM system.

[6]http://www.elis.ugent.be/diablo.
[7]http://www.busybox.net.

Table I. Code and Data Sizes of Different Parts and Different Forms of the Kernel for Both Our Evaluation Systems[a]

	Text Size	Data Size	Init Text Size	Init Data Size	Image Size	Compr. Image Size	Init Mem. Footprint	Memory Footprint
Original kernel	767	328	47	8.1	1021	508	1150	1095
(a) x86 results	Compaction techniques							
+ Unreachable code/data elim.	732 -4.6%	306 -6.7%	46 -1.3%	8.0 -2.0%	959 -6.0%	495 -2.5%	1092 -5.0%	1038 -5.2%
+ Duplicate code removal	686 -10.5%	300 -8.7%	46 -2.0%	8.0 -2.0%	907 -11.1%	498 -1.9%	1039 -9.6%	986 -10.0%
+Whole-program optimization	684 -10.9%	300 -8.7%	46 -2.1%	8.0 -2.0%	903 -11.5%	497 -2.1%	1037 -9.8%	984 -10.2%
Kernel specialization techniques								
+ Initialization code motion	670 -12.7%	300 -8.7%	60 28.5%	8.0 -2.0%	899 -11.9%	497 -2.1%	1037 -9.8%	969 -11.5%
+ System call elimination	625 -18.5%	299 -9.1%	60 28.6%	8.0 -2.0%	855 -16.2%	472 -7.1%	991 -13.8%	923 -15.7%
+ Command-line specialization	622 -18.9%	294 -10.3%	59 27.7%	7.7 -5.2%	847 -17.0%	468 -7.7%	984 -14.5%	917 -16.3%
- Dupl. basic block elim.	631 -17.7%	294 -10.3%	60 28.2%	7.7 -5.2%	859 -15.8%	460 -9.4%	993 -13.7%	925 -15.5%
Unexecuted code compression								
+ Unexecuted code compr.	287 -62.5%	553 68.4%	59 27.7%	7.7 -5.2%	772 -24.4%	521 2.6%	907 -21.1%	840 -23.3%
Original kernel	1311	218.4	53	5.8	1471	702	1588	1530
(b) ARM results	Compaction techniques							
+ Unreachable code/data elim.	1241 -5.4%	218.2 -0.1%	52 -1.9%	5.8 0.0%	1365 -7.2%	716 2.1%	1516 -4.5%	1459 -4.6%
+ Duplicate code removal	1194 -9.0%	218.2 -0.1%	51 -3.8%	5.8 0.0%	1317 -10.5%	716 2.0%	1468 -7.5%	1412 -7.7%
+Whole-program optimization	1163 -11.3%	218.2 -0.1%	49 -6.1%	5.8 0.0%	1285 -12.7%	710 1.1%	1436 -9.5%	1381 -9.7%
Kernel specialization techniques								
+ Initialization code motion	1139 -13.1%	218.2 -0.1%	74 41.0%	5.8 0.0%	1285 -12.7%	709 1.1%	1437 -9.5%	1357 -11.3%
+ System call elimination	1092 -16.7%	218.1 -0.1%	74 41.1%	5.8 0.0%	1237 -15.9%	684 -2.5%	1390 -12.5%	1310 -14.4%
+Command-line specialization	1062 -19.0%	218.1 -0.1%	72 38.0%	5.5 -4.6%	1209 -17.8%	668 -4.8%	1358 -14.4%	1281 -16.3%
- Dupl. basic block elim.	1094 -16.6%	218.1 -0.1%	74 41.0%	5.5 -4.6%	1241 -15.7%	663 -5.5%	1392 -12.4%	1312 -14.2%
Unexecuted code compression								
+ Unexecuted code compr.	455 -65.3%	646.0 195.8%	72 38.0%	5.5 -4.6%	1029 -30.1%	713 1.6%	1179 -25.8%	1101 -28.0%

[a] All sizes are in kibibytes (KiB), and all percentages are relative to the sizes in the original kernel. In each row, the techniques mentioned in the left column are applied on top of the techniques mentioned in the rows above, except for the next-to-last row, for which we disabled duplicate basic block elimination.

The leftmost column presents the (additional) level of compaction that was applied. The next two columns in the table present the sizes of the regular code and data sections of the kernel. The data size includes the read-only, writable, and zero-initialized data sections. The next two columns present the sizes of the initialization code and data sections. The next column shows the size of the resulting `vmlinux` image (the uncompressed disk image), in which the zero-initialized sections occupy no place. The seventh column shows the size of the corresponding gzipped kernel image file. We've included this number because, depending on the kind of system, one of these images is stored on disk or in ROM. The last two columns show the memory footprint of the code and statically allocated data, respectively, during and after system initialization. In the remainder of this section, we discuss the most important results.

6.2.1 *Link-Time Compaction Techniques.* Applying well-known, general purpose link-time compaction techniques to the kernel results in a reduction of the memory footprint of about 10% on both platforms. Approximately one-half of this gain can be attributed to unreachable code and data elimination. Given the size of the Linux kernel source base this does not surprise us. Any large project that has evolved over a long period of time is bound to contain unreachable code and inaccessible data. It does, however, indicate that there is still

some margin for improving the kernel configurability, through which unused code should normally be excluded.

It should be noted that some of the size reductions obtained with unreachable code and data elimination can, in theory, also be achieved by simply compiling the kernel with the compiler flags `-ffunction-sections` and `-fdata-sections`, and linking the resulting object files with the `--gc-sections` flag. These flags instruct the compiler to place every procedure and every global variable in its own section. The linker can then remove all unreferenced sections from the final binary. This is somewhat similar to our link-time compactor's unreachable code elimination, albeit at a coarser granularity. Moreover, invoking these compiler flags deteriorates the quality of the generated code, as the compiler can perform less address computation optimizations. A true link-time optimizer obviously does not suffer from this drawback.

Most of the other gain, especially on the i386, comes from duplicate code removal. Code duplication gains at the procedure level result from the fact that there are a lot of similar procedures in the kernel that operate on superficially different data structures (e.g., a list of pointers to virtual memory pages versus a list of pointers to open files), from cut-and-paste duplication by the developers, and from the fact that the GCC compiler does not always honor the inlining requests of the programmer. Code duplication gains at the basic block level result from the fact that there are a number of similar procedures in the kernel that differ enough to be unsuitable for procedure-level factoring, but have a number of basic blocks that are identical. The other major source of opportunities for basic block factoring comes from the use of inline assembler macros in the source code. These macros, which implement things like copying a value from user-space memory to kernel space, appear quite frequently and are always inlined into their callers.

The other whole-program compaction techniques implemented in Diablo add another 2% to the obtained compaction for the ARM, but amount to practically nothing for the i386. This is because most of the analyses treat memory as a black box. As the i386 architecture lacks sufficient user-visible registers, almost all computations involve the stack. This makes the data-flow analyses very imprecise.

It may seem remarkable that none of the transformations have an impact on the size of the data section for the ARM kernel, while they are capable of removing 10% of the same data on the i386. The reason is simple, however: all read-only data is incorporated into the code section on the ARM and is thus counted as code. On the i386 this data resides in a separate section and is counted as data. The compaction techniques typically have much more impact on the read-only data sections than on mutable data sections, which explains the very small impact of the compaction techniques on the ARM data sections.

6.2.2 *Initialization Code Motion.* This specialization has no impact on the total kernel image size, but it does reduce the kernel's memory footprint after system initialization with 1.3% on the i386, and with 1.6% on the ARM, which corresponds to approximately 5000 and 6000 instructions that were moved to the `.init` sections.

6.2.3 *System Call Elimination and Specialization.* To determine which system call handlers may be removed from the kernel, all software that will run on the system has to be analyzed. Thanks to the simple setup of our test systems, this means analyzing just one user-space binary, namely Busybox, and the Linux kernel itself. This analysis was performed using a modified version of Diablo that reports the known register values for each system call instruction. With this approach, we were able to determine for each system call instruction which system call was actually invoked, and, on the ARM platform, we were even able to determine all possible values for 17 system call arguments. On the i386 platform, Diablo was unable to determine values for any system call argument, mainly because Diablo currently cannot perform constant propagation through stack frames. If this were possible (such an analysis is described by Schwarz et al. [2001]), we would expect to find a comparable number of constant system call arguments as on the ARM system.

The analysis showed that of the 245 system calls offered by the Linux kernel, only 87 can actually be called on the ARM system, and only 88 on the i386 system. All other system call handlers were removed from the kernel, resulting in a 3 (ARM) or 4% (i386) reduction of the memory footprint.

On the ARM, the constant system call arguments passed to the kernel by Busybox were propagated into the kernel. The impact of propagation on program size is insignificant (0.03% of the noninitialization code was removed), but it did result in the removal of a number of argument validity checks in the system call handlers.

6.2.4 *Command-Line Specialization.* The final specialization step consists of specializing the kernel for known, fixed command-line parameters. As a first step, we determined which parameters should still be adjustable at boot time. For the i386 system, this was the "root" parameter that specifies the disk device on which the root partition is installed. For the ARM, we additionally left the parameter "console" adjustable, in order to allow us to specify the console input and output devices, as the CerfCube has no screen and keyboard. The auxiliary parsing procedures for all other parameters were removed from the kernel, leading to an additional gain of 3% in the init data size and 1% in the init code size for the i386 and 5% in the init data size and 3% in the init code size for the ARM.

The second step was to propagate the known values of the boot-time parameters, and of some driver parameters that can only be changed if the driver is compiled as a module into the kernel. For this, we selected 37 parameters for the ARM system, and 17 for the i386 system. This resulted in a gain of 0.4% in the code size and 1.2% in the data size for the i386 and a 2.3% gain in the code size for the ARM. Most of this gain is attributable to a number of debugging parameters that were fixed at 0, meaning that no debug output should be produced. As a result, the involved literal strings are eliminated from the text section on the ARM and from the data section on the i386.

Together, all compaction and specialization optimizations thus bring the memory footprint down by 14.5 during and 16.3% after system initialization for the i386, and by 14.4 during and 16.3% after initialization for the ARM.

Table II. Relevant Data Concerning the Frozen-Code Compression Case Study

		i386	ARM
Without frozen code compression	non-init code size (KiB)	622.14	1062.38
	# instructions	218128	271968
Partitioning overhead	code size growth (KiB)	4.05	18.85
	total non-init code size before compression (KiB)	626.18	1081.22
Frozen code	# regions	1503	2221
	# instructions	115528	167332
	frozen code size (KiB)	353.24	653.64
	compressed size (KiB)	257.96	427.82
	compression ratio	0.73	0.65
Decompressor overhead	total stub size (KiB)	13.21	26.03
	decompressor code size (KiB)	1.14	1.39
	decompressor code size (KiB)	0.32	0.02
	section placement padding (KiB)	0.03	0
	net gain (KiB)	76.57	179.53

The kernel image size is reduced by 17.0 and 17.8%, respectively; the gzipped image size is reduced by 7.7 and 4.8%, respectively.

6.2.5 *Frozen-Code Compression.* In order to evaluate the frozen-code compression technique described in this article, we conducted a case study in which we used our test systems as embedded web servers. For the coverage analysis, a number of usage scenarios were executed: long idle time, high load web serving, low load web serving, requests for nonexisting pages, etc. Failure conditions like network errors and file system corruption were considered to be abnormal behavior and, as such, were not included in our usage scenarios. The code for handling these failure conditions was consequently considered frozen. The correctness of the kernel after frozen-code compression was tested by triggering some conditions that did not occur during the code-coverage analysis, like unplugging the network cable during transfer of a large file, running a file system check on the system's hard drive, etc. Furthermore, we executed a large number of system utility programs that were not executed during the coverage analysis. These tests resulted in considerable amounts of code being decompressed, without incorrect kernel behavior being observed.

In order to estimate the effectiveness of the frozen-code identification, we observed our test systems during a 5-day period. In this period, the i386 system decompressed 10 regions for a total of 4502 Bytes and the ARM system decompressed 4 regions for a total of 2144 Bytes. In both cases, the decompressed code was responsible for handling a UDP connection request to the systems. During the coverage analysis, there were no UDP connection requests to either of the two systems and we do not consider this to be normal behavior either, because the systems only serve as HTTP servers and HTTP is a TCP service. Consequently, we conclude that the coverage analysis performed on our test systems is sufficiently accurate for this test case.

Table II shows the results of our case study. After all previously mentioned compaction and customization techniques are applied, the i386 kernel has 622 KiB of noninitialization code and the ARM kernel has 1062 KiB. This corresponds to 218,128 and 271,968 instructions, respectively. Splitting the code in frozen and non-frozen parts and partitioning the frozen code in single-entry

regions incurs some overhead: a number of jump instructions have to be inserted to ensure correct control flow. Furthermore, because the final addresses of the frozen-code regions are unknown, some constructs have to be encoded less efficiently. For instance, on the i386 interregion jumps will always have a 4-Byte offset whereas in some cases it would have been possible to use the shorter 1-Byte offset jump instruction if the code had not been split up. On the ARM platform, absolute addresses cannot be generated in one instruction and the efficiency of the instruction sequences that can be used to generate an address is dependent on the code layout [De Sutter et al. 2006]. In the frozen-code regions, one always has to use the most conservative (meaning longest) instruction sequence. For the i386 platform, the total partitioning overhead is 4.05 KiB and for the ARM platform it is 18.85 KiB.

For the i386, 56.4% (353.24 KiB) of the noninitialization code was considered frozen and partitioned into 1503 single-entry regions. For the ARM, 60.5% (653.64 KiB) of the noninitialization code was frozen and partitioned into 2221 regions. These frozen-code percentages are lower than those reported by Cours et al. [2004]. This was to be expected, as the preceding link-time compaction and specialization transformations have already removed part of the unreachable code from the kernel.

After compression, the frozen code for the i386 kernel is 257.96 KiB large, which means a compression ratio (= size compressed code/size original code) of 0.73 is achieved. For the ARM kernel, the frozen-code size after compression is 427.82 KiB, so the compression ratio is 0.65. While the compression ratio for ARM code is on par with the ratio reported by Debray and Evans [2002] for a similar compression scheme for the Alpha architecture, significantly less compression was achieved for the i386 code. This should be no surprise, as i386 CISC code is typically much denser than the RISC code of the Alpha and ARM architectures, leaving less opportunities for compression.

Taking into account the overhead incurred by the stubs, the code and data size of the decompressor and the padding bytes added for section placement (see Section 4.7), a net gain of 76.57 KiB is achieved for the i386 kernel and 179.53 KiB for the ARM kernel. As shown in the last row of Table Ia and Ib, this results in an additional reduction of the memory footprint, after initialization, under normal operation of 7% for the i386 kernel and 11.7% for the ARM kernel. Additional compaction of the kernel image is 7.4 and 12.3%, respectively.

Given the compression ratio and stub size for both architectures, we can now determine the minimum region size for compression, as explained in Section 4.3. According to the formula explained there, the minimum region size for the i386 architecture should be $9/(1 - 0.73) \approx 34$ bytes; for the ARM architecture this is $12/(1 - 0.65) \approx 35$ bytes. However, brute-force searching shows the best minimum size to be 50 bytes for both architectures, so our measurements were done with this minimum region size. In both cases, the difference in results was, however, smaller than 0.2%, which shows that the formula does give a good approximation for the optimal minimum region size.

Enabling all compaction, specialization, and compression transformations allows us to reduce the kernel's memory footprint by 21.1 during and 23.3% after system initialization for the i386 and by 25.8 during and 28.0% after

initialization for the ARM test system. The size of the kernel image was reduced by 24.4 and 30.1%, respectively. Disappointingly, the size of the resulting gzipped kernel image grows by 2.6% on the i386 test system and by 1.6% on the ARM test system. This is discussed in the next section.

6.3 Impact on Gzipped Image Size

It is clear that the gains obtained for the gzipped image size are somewhat disappointing. Surprisingly, some of the gzipped versions of compacted, specialized and compressed images are larger than the gzipped version of the original, larger image. This is important, because in many systems it is this gzipped version of the kernel that is stored in (expensive) flash memory. In this section, we will try to pinpoint the causes of this phenomenon. As the impact of frozen-code compression is so large, it will be discussed separately from the impact of the other techniques described in this paper.

6.3.1 *Compaction and Specialization Techniques.* To some extent the difference between image size reduction and gzipped image size reduction is understandable. Gzipping, like any compression technique, reduces the amount of redundant information from a byte stream. Compaction techniques such as duplicate code removal also remove redundancy from a program, albeit on a different level. In fact, from Table I, it is immediately clear that on both platforms duplicate code removal increases the size of the gzipped image, even though it significantly decreases the size of the unzipped image. The main culprit for the increase in gzipped image size is the duplicate basic block removal, as this transformation replaces easily compressible information (identical pieces of code) by information that is much harder to compress (calls to the extracted procedure, which all contain different relative displacements). Duplicate code removal at the procedure level does not have the same detrimental effect, as it does not involve the insertion of additional, hard-to-compress function call instructions in the program code. Instead, duplicate procedure elimination only replaces calls to one function by calls to another function.

To quantify the effect of duplicate basic block elimination on the compressibility of the kernel image, we applied all compaction and specialization techniques, minus the duplicate basic block elimination and the frozen-code compression. The resulting total compaction is presented on the last row of Table Ia and Ib. These results confirm our argument, as turning off duplicate basic block removal decreases the gzipped image size by 1.7 on the i386 and by 0.7% on the ARM, even though the unzipped image sizes increase significantly!

The viability of duplicate basic block elimination, hence, depends on one's optimization target. When the goal is reducing the memory footprint, this optimization should certainly be applied. If the goal is reducing the gzipped image's size, however, e.g., because the gzipped image will be stored in more expensive flash memory, this transformation is best disabled.

While the effect of duplicate basic block elimination completely accounts for the increase of the compressed image size for the i386 test system, this is clearly not the case for the ARM system. Here, the gzipped image already grows when the unreachable code and data is removed from the kernel, notwithstanding the

Table III. Gzip Compressability of Various Code Types[a]

	Unzipped	Gzipped	Ratio
i386 all code	637069	371347	0.58
i386 non-frozen code	294187	173813	0.59
i386 compressed frozen code	264147	252120	0.95
ARM all code	1087872	611344	0.56
ARM non-frozen code	465920	247762	0.53
ARM compressed frozen code	438088	409742	0.94

[a]The first two columns of numbers indicate code sizes (in KiB) before and after gzipping, the third column shows the compression ratios obtained

fact that the corresponding unzipped image was reduced by 5%. The only difference between this kernel image and the original one is that the unreachable code and data have disappeared from the image and that the code and data layout has changed. To understand this result, we have examined a large number of regular ARM programs (from the SPEC and Mediabench benchmark suites), on which we applied several different code and data layout algorithms before gzipping them. Although we always observed a similar behavior, we have, until this day, not been able to pinpoint precise causes of this behavior, as we have not observed any systematic relation between code layout properties (such as average branch displacement) and compressibility. Consequently, this remains an open, and in our opinion, very intriguing question.

6.3.2 *Frozen-Code Compression.* As the last line of Table Ia shows, compressing the frozen code in the kernel completely undoes the gain in gzipped image size. Even though the unzipped kernel image size decreases significantly, the gain on the compressed image size turns into a 1–3% loss.

In order to understand the cause of this loss, we have applied gzip compression separately against (1) the completely uncompressed code sections, against (2) the nonfrozen code-sections, and (3) against the compressed frozen-code sections of the kernel for both architectures. The results are shown in Table III. This table clearly shows the cause of the problem: the remaining uncompressed code is just as easily compressible as it was before, but the already compressed code is almost not compressible at all. Since the compression ratio of our compression scheme (0.73 for i386 and 0.65 for ARM) is higher than that of gzipping (0.58–0.59), the net effect is that the total gzipped size of all code grows by 53 KiB for i386 and 45 KiB for ARM, which accounts for almost all the compressed image size loss.

Even more than with duplicate basic block elimination, the viability of frozen code compression as a compaction strategy depends on the ultimate optimization goal. While this technique can significantly reduce the size of the kernel image in memory, it should not be used if the main optimization goal is to reduce the kernel's gzipped image size.

6.4 Performance Impact

There are only two techniques described in this paper that might theoretically have a detrimental effect on execution speed: duplicate code elimination at

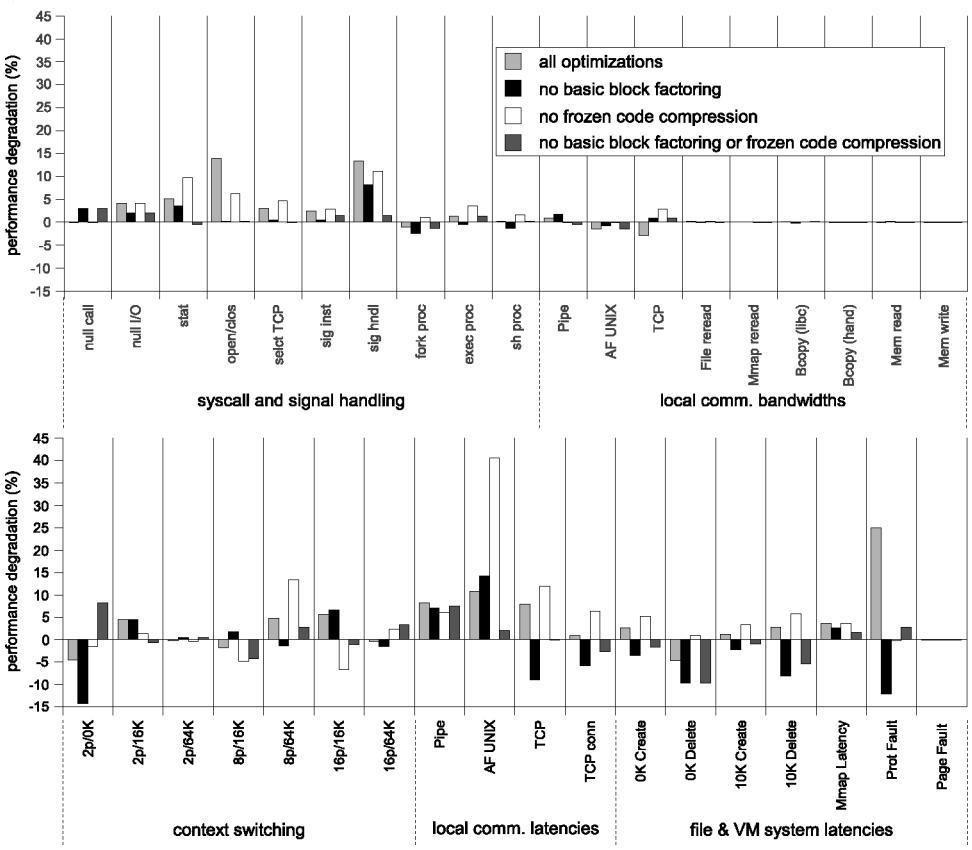

Fig. 7. Performance degradation for the LMBench benchmark suite on the i386 test system.

the basic block level [De Sutter et al. 2002] and frozen-code compression. To measure the performance impact of our compaction, specialization, and compression techniques on the kernel, we have performed several runs of LMBench 2.0.4 [McVoy and Staelin 1996] on our evaluation systems. This benchmark set measures a number of aspects of the kernel's behavior, like system call performance, interprocess communication latencies, and context-switching times.

The charts in Figures 7 and 8 show the performance degradation observed after our specialization and compaction techniques have been applied to the i386 and ARM test system, respectively. For each system, the first bar indicates the degradation observed when all techniques described in this paper are applied. The second bar shows performance degradation when only duplicate basic block elimination was disabled. For the third bar, only frozen-code compression was disabled; for the fourth bar, both were disabled.

As we used the coverage analysis results from the embedded web server use case to guide the frozen-code identification, a lot of code was decompressed during the LMBench runs: for the i386 test system 69 regions (21,062 bytes) were decompressed and for the ARM test system 113 regions (33,784 bytes) were decompressed. Even a single run of the LMBench suite runs most benchmarks

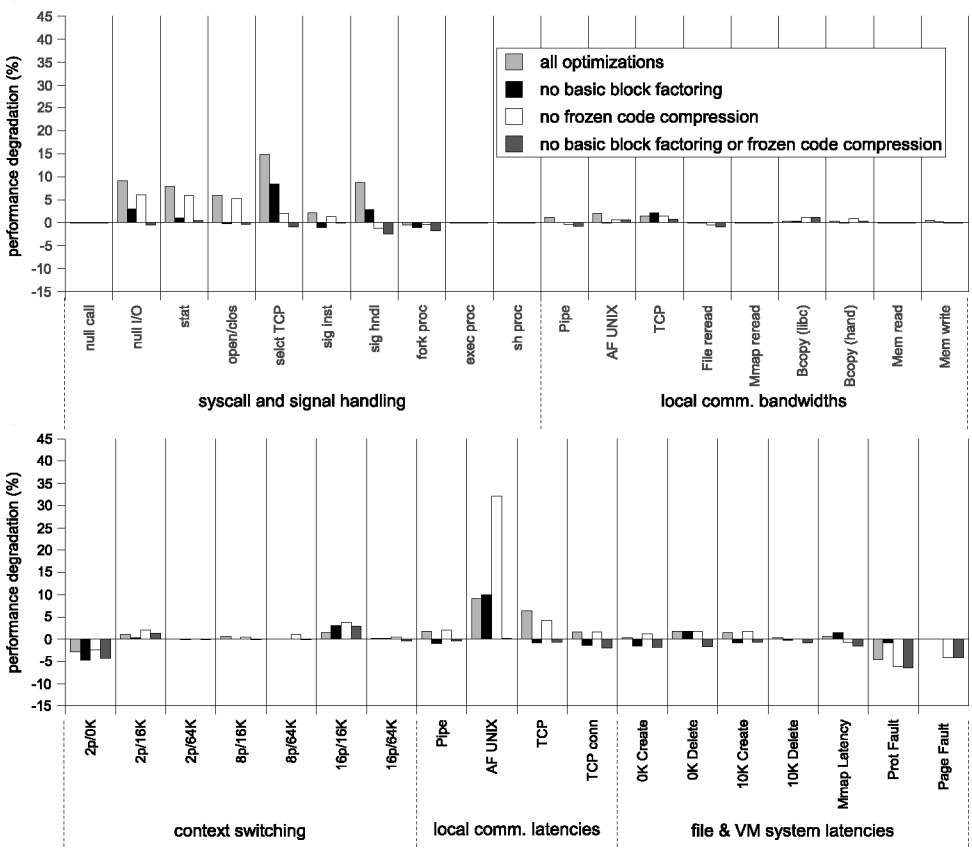

Fig. 8. Performance degradation for the LMBench benchmark suite on the ARM test system.

multiple times, so the effects of the decompression (which happens only once for each region) are not easily measurable, but the slowdown for subsequent executions of the region (because of the jump instruction in the overwritten stub that transfers control to the decompressed code) will be recorded in the results.

With all optimizations enabled, the average performance degradation is 2.86% for the i386 test system and 1.97% on the ARM test system. Disabling just basic block factoring turns the performance loss into a 0.39% average performance gain for the i386 and reduces the average performance loss on the ARM to 0.56%. Disabling only frozen-code compression resulted in a 3.66% average performance loss on the i386 and a 1.67% average performance loss on the ARM. Disabling both transformations brought the average performance loss down to 0.26% for the i386 and achieved a 0.69% average performance gain for the ARM.

These results clearly show that duplicate basic block elimination has the largest negative impact on execution speed. For the ARM architecture, the frozen-code compression does not appear to have a significant negative effect on performance. For the i386 architecture, the frozen-code compression even has

a (marginally) positive effect on execution speed. We suspect this is because of cache effects: splitting the code in executed and nonexecuted parts allows for better I-cache utilization and, hence, for better performance.

Unfortunately, no significant average performance improvement is seen either. This results from the fact that we have not been able to optimize the system calls evaluated by LMBench for specific arguments. As their calling contexts are unknown, little optimization of these system calls is possible. Still some optimization proved to be possible, as some benchmarks show performance improvements of up to 14%. These improvements mainly result from procedure inlining and interprocedural data-flow optimizations within the call chains of the evaluated system calls.

6.5 System Requirements

All compaction experiments were conducted on a 2.8-GHz Pentium 4 system with 2 GiB of RAM, running Ubuntu Linux 5.10. The compaction time was 306 s for the ARM kernel with all optimizations and specializations and frozen-code compression enabled, and 128 s for the i386 kernel. The i386 kernel takes less time because less data-flow analyses have been implemented in Diablo for the i386. Thus, less analyses and optimizations are performed on the i386 kernel. The maximum memory usage observed is 240 MiB.

7. RELATED WORK

In this section, we will present an overview of the related work in the areas of whole-program compaction techniques, OS kernel specialization, other OS kernel optimization approaches, and code compression techniques.

7.1 Whole-Program Compaction Techniques

To the best of our knowledge, link-time optimization has never been used to compact kernels. However, there are other link-time optimization systems that target program size. These include Squeeze [Debray et al. 2002] and Squeeze++ [De Sutter et al. 2002, 2005], two evolutions of a proof-of-concept implementation on the Alpha architecture. Code-size reductions of up to 62% for large C++ benchmarks were obtained with Squeeze++. This number is much higher than what was obtained on the Linux kernel in this paper. The most important reason is that C++ code contains much more duplicated code as a consequence of the use of templates. Moreover, the programs evaluated with Squeeze++ were statically linked user-space applications, that include large amounts of system library code. Such library code, because it is written with general applicability in mind, provides much more opportunities for unreachable code elimination than application-specific code, or indeed, kernel code. Besides Squeeze++, we have also developed the Diablo framework, with which we evaluated link-time compaction on user-space applications for a number of platforms, including ARM [De Bus et al. 2003, 2004] and MIPS [Madou et al. 2004]. The results obtained in that work are comparable to the results obtained here. Combined with the results in this paper, our results prove that we are now able to compact, at link time, all software on fixed-function devices.

Whereas the aforementioned tools deal with object code files, aiPop [De Bus et al. 2003] applies postpass optimization for the C16x architecture on the assembler code of a whole program. With aiPop, code-size reductions ranging from 5 to 20% have been achieved on real-life customer applications. However, no kernel-specific results of aiPop were presented until today.

The idea of eliminating frozen code from the memory image of a running program was explored by Citron et al. [2004]. Frozen-code fragments are replaced by stubs that load the code when it is accessed. The frozen code is, however, not compressed, but only removed from the loadable image and thus results in larger compaction ratios than our approach. This technique achieves size reductions of 78% for the MediaBench suite.

7.2 Operating System Specialization

The idea of specializing the Linux kernel for a specific application was first explored by Lee et al. [2004]. Based on source code analysis, a system-wide call graph is built that spans the application, the libraries, and the kernel. On this graph, a reachability analysis is performed, resulting in a compaction of the kernel of 17% in a simple, but very unclear case study. We believe our approach to be more general, as it is source-language independent, and because more optimizations can be performed at link time.

An alternative approach to customize an OS for use in embedded devices is proposed by Bhatia et al. [2004]. Instead of manually customizing the OS for specific hardware features and handcrafting the generic code base to a hardware-specific version, the authors of this paper propose to remotely customize OS modules on demand. A customization server provides a highly optimized and specialized version of an OS function on demand of an application. The embedded device needs to send the customization context and the required function to the server and on receipt of the customized version, applications can start using it. The size of the customized code is reduced by up to a factor of 20 for a TCP/IP stack implementation for ARM Linux, while the code runs 25% faster and throughput increases by up to 21%.

While our approach to minimizing the kernel's memory footprint is top-down in that we start with a full-featured kernel and strip away as much unneeded functionality as possible, there are a number of projects that take a bottom-up approach. The Flux OSKit [Ford et al. 1997], Think [Fassino et al. 2002], and TinyOS [Gay et al. 2003] are operating system building frameworks that offer a library of system components to the developer, allowing him to assemble an operating system kernel containing only the needed functionality for the system.

Krintz and Wolski [2005] propose applying specialization to the Linux kernel to improve the performance of scientific applications in batch-processing systems. In such systems, only one application is running at any time, so it is possible to tailor the kernel to this one application. For every new job that is run on the system, a new, specialized kernel is loaded. The objective of this work is to improve the performance of the scientific application, not to reduce the memory footprint of the kernel.

McNamee et al. [2001] introduce a toolkit for the systematic specialization of operating system code. Their approach is based on the partial evaluation of source code, generating specialized versions of parts of the kernel. The specialization can happen both statically, at system build time, and dynamically, at runtime, but only the static approach is really useful with regard to memory-footprint reduction. The focus of this work is also more on achieving performance improvements than on kernel memory-footprint reduction.

In the past, there have been many research projects that focused on dynamically specializing OS kernel subsystems in order to improve the performance of specific applications. A good overview of the varied approaches is given by Denys et al. [2002]. However, none of these approaches is particularly useful in reducing the memory footprint of a kernel.

7.3 Other Operating System Kernel Optimization Approaches

Spike [Cohn et al. 1997] is a (post-)link-time optimizer for the Alpha architecture. Spike includes profile-guided code layout to improve cache usage. Spike has also been used to optimize Tru64Unix kernels [Flower et al. 2001] for speed, both through profile-guided code layout and through the profile-guided insertion of data prefetching instructions. Performance improvements of up to 40% on a set of benchmarks running on an optimized kernel were reported for this Spike version.

By contrast with link-time optimization, most kernels are traditionally optimized by the compiler only. To detect kernel bottlenecks, profile information is used. Profile-guided restructuring of the operating system for the optimization of its throughput or latency has been studied for AS400 [Schmidt et al. 1998] and HP-UX [Speer et al. 1994] platforms.

The KernInst dynamic kernel instrumentation system [Tamches and Miller 1999] has been used to optimize parts of the UltraSPARC Solaris kernel [Tamches and Miller 2001]. As its approach is dynamic, it cannot optimize for code size: the entire kernel has to remain in memory. KernInst uses a code-positioning scheme, similar to the one used by Spike, which results in speedups up to 17.6% for selected functions.

7.4 Code Compression

Code compression has been used in several research initiatives. Most of this work focuses on compressing executable files in order to save on storage or transmission costs. Schemes have been used where either the code is decompressed to the original size [Ernst et al. 1997; Franz 1997; Franz and Kistler 1997; Fraser 1999; Pugh 1999], as in our approach, or where special hardware support needs to be provided for executing the compressed code directly [Lekatsas et al. 2003; Kemp et al. 1998; Kirovski et al. 1997; TriMedia Technologies Inc. 2000; Corliss et al. 2003].

Using another branch of the Squeeze (see Section 7.1) code, Debray and Evans [2002] added software-controlled code compression to already compacted binaries. They used profile data to identify infrequently executed code fragments, which they compressed to a nonexecutable form. At runtime, the

fragments are decompressed into a buffer on demand and executed. They obtained additional code-size reductions of 13.7 to 18.8% (including the decompressor and buffer). The influence on performance ranges from a slight speedup to a 28% slowdown. The main difference between their techniques and the compression method described in this article are (1) that we never need to evict compressed code, (2) that we can, therefore, keep the decompressor and the stubs simpler, and (3) that we deal with concurrency. The latter was completely neglected by Debray and Evans [2002].

Proebsting [1995] has studied compression at the compiler level. He collects repeated patterns in the intermediate program representation and creates superoperators for the most frequently occurring patterns. The selection of superoperators is application-specific. Subsequently, the superoperators are used to extend a virtual instruction architecture, for which the program is compiled. To execute the superoperators, an interpreter that can interpret the extended instruction set is generated in C. This interpreter is then compiled for and run on the original target architecture. An average compression ratio of 50% is achieved, but the impact on execution speed is tremendous.

Evans and Fraser [2001] use a similar scheme for generating compact, byte-coded instruction sets and corresponding interpreters. Their system transforms a grammar for programs, creating an expanded grammar that represents the same language as the original grammar, but permits a shorter derivation of programs. Typically the program size reductions obtained are much larger than the increase in the size of the adapted interpreter.

Following a similar approach, Clausen et al. [2000] adapted the Java Virtual Machine so that it can decode macros that combine frequently occurring byte-code instruction sequences. They achieve a compression ratio of 15%, on average.

The compressed program size growth problem when software compaction and software-supported compression techniques are combined has, to the best of our knowledge, not been studied in the context of an ARM-like architecture. However, a similar problem, relating to the combined use of compiler optimization and hardware-supported code compression has been studied for VLIW architectures [Ros and Sutton 2003]. In this paper, Ros and Sutton report that, in general, instruction scheduling did not influence compressibility when the compiler had already optimized the code for code size.

8. CONCLUSIONS

In this paper, we proposed to apply established whole-program compaction techniques and new whole-system specialization techniques to the Linux kernel at link-time. The whole-system specialization techniques exploit the fact that the runtime environment of embedded systems is known *a priori*. In an experimental setup, the presented techniques reduced the memory footprint of the Linux kernel with over 16%. A major contribution of this paper is the simplicity with which existing link-time program rewriters can be extended to perform the presented transformations to a complex, unconventional program, such as the Linux kernel.

We observed that, even after application of the compaction and specialization techniques, a lot of frozen, meaning seemingly never executed, code remains in the kernel. To reduce the overhead caused by this code, we proposed to store this code in a compressed form that allows for on-demand runtime decompression. Combined with the aforementioned techniques, this technique reduced the memory footprint of the Linux kernel with over 23% for the i386 architecture and 28% for the ARM architecture.

Furthermore, we identified a definite, but unexplained program growth problem when compaction and compression are combined on the ARM architecture. This remains an open problem and an interesting topic for future work.

ACKNOWLEDGMENTS

Dominique Chanet is supported by the Fund for Scientific Research—Flanders as a PhD. student. Bjorn De Sutter was sponsored by the same organization as a Postdoctoral Research Fellow. Ludo Van Put is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Bruno De Bus is sponsored by the SARC European research project. This research is also partially supported by Ghent University and the HiPEAC European Network of Excellence.

REFERENCES

Anckaert, B., Vandeputte, F., De Bus, B., De Sutter, B., and De Bosschere, K. 2004. Link-time optimization of IA64 binaries. In *Proceedings of the 2004 Euro-Par Conference*. 284–291.

Bhatia, S., Consel, C., and Pu, C. 2004. Remote customization of systems code for embedded devices. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. ACM Press, New York. 7–15.

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

Citron, D., Haber, G., and Levin, R. 2004. Reducing program image size by extracting frozen code and data. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. ACM Press, New York. 297–305.

Clausen, L., Schultz, U., Consel, C., and Muller, G. 2000. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems 22*, 3 (May), 471–489.

Cohn, R., Goodwin, D., Lowney, P., and Rubin, N. 1997. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*. 17–24.

Corliss, M., Lewis, E., and Roth, A. 2003. A DISE implementation of dynamic code decompression. In *Proceedings of the ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. 232–243.

Cours, J., Navarro, N., and Hwu, W. 2004. Using coverage-based analysis to automate the customization of the Linux kernel for embedded applications. M.S. thesis, University of Illinois at Urbana-Champaign.

De Bus, B. 2005. Reliable, retargetable and extensible link-time program rewriting. Ph.D. thesis, Ghent University.

De Bus, B., Kästner, D., Chanet, D., Van Put, L., and De Sutter, B. 2003. Post-pass compaction techniques. *Communications of the ACM 46*, 8 (Aug.), 41–46.

De Bus, B., De Sutter, B., Van Put, L., Chanet, D., and De Bosschere, K. 2004. Link-time optimization of ARM binaries. In *Proc. of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 211–220.

De Sutter, B., De Bus, B., De Bosschere, K., and Debray, S. 2001. Combining global code and data compaction. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*. 29–38.

De Sutter, B., De Bus, B., and De Bosschere, K. 2002. Sifting out the mud: low level C++ code reuse. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 275–291.

De Sutter, B., De Bus, B., and De Bosschere, K. 2005. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems 27*, 5 (9), 882–945.

De Sutter, B., Van Put, L., Chanet, D., De Bus, B., and De Bosschere, K. 2006. Link-time compaction and optimization of ARM executables. *ACM Transactions on Embedded Computing Systems*. To appear.

Debray, S., Muth, R., and Weippert, M. 1998. Alias analysis of executable code. In *Proceedings of the ACM 1998 Symposium on Principles of Programming Languages (POPL'98)*. 12–24.

Debray, S. and Evans, W. 2002. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, New York. 95–105.

Debray, S., Evans, W., Muth, R., and De Sutter, B. 2002. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems 22*, 2 (3), 378–415.

Denys, G., Piessens, F., and Matthijs, F. 2002. A survey of customizability in operating systems research. *ACM Comput. Surv. 34*, 4, 450–468.

Ernst, J., Evans, W., Fraser, C., Lucco, S., and Proebsting, T. 1997. Code compression. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*. 358–365.

Evans, W. S. and Fraser, C. W. 2001. Bytecode compression via profiled grammar rewriting. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM Press, New York. 148–155.

Fassino, J.-P., Stefani, J.-B., Lawall, J. L., and Muller, G. 2002. Think: A software framework for component-based operating system kernels. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA. 73–86.

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

Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. 1997. The flux oskit: a substrate for kernel and language research. In *SOSP '97: Proceedings of the 16th ACM symposium on Operating systems principles*. ACM Press, New York. 38–51.

Franz, M. 1997. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. In *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds. Number 1222 in LNCS. Springer, New York. 263–276.

Franz, M. and Kistler, T. 1997. Slim binaries. *Communications of the ACM 40*, 12 (Dec.), 87–94.

Fraser, C. 1999. Automatic inference of models for statistical code compression. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*. 242–246.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, New York. 1–11.

Hind, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. Snowbird, UT.

Kemp, T. M., Montoye, R. M., Harper, J. D., Palmer, J. D., and Auerbach, D. J. 1998. A decompression core for PowerPC. *IBM J. Research and Development 42*, 6 (Nov.).

Kirovski, D., Kin, J., and Mangione-Smith, W. H. 1997. Procedure based program compression. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*.

KRINTZ, C. AND WOLSKI, R. 2005. Using phase behavior in scientific application to guide Linux operating system customization. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)—Workshop 10*. IEEE Computer Society, Washington, DC. 219.1.

LEE, C.-T., LIN, J.-M., HONG, Z.-W., AND LEE, W.-T. 2004. An application-oriented Linux kernel customization for embedded systems. *Journal of Information Science and Engineering 20*, 6, 1093–1107.

LEKATSAS, H., HENKEL, J., CHAKRADHAR, S., JAKKULA, V., AND SANKARADASS, M. 2003. Coco: a hardware/software platform for rapid prototyping of code compression technologies. In *Proceedings of the 40th conference on Design Automation (DAC)*. 306–311.

LEVINE, J. 2000. *Linkers & Loaders*. Morgan Kaufmann, San Matco, CA.

LINN, C., DEBRAY, S., ANDREWS, G., AND SCHWARZ, B. 2004. Stack analysis of x86 executables. Available from http://www.cs.arizona.edu/people/debray.

MADOU, M., DE SUTTER, B., DE BUS, B., VAN PUT, L., AND DE BOSSCHERE, K. 2004. Link-time optimization of MIPS programs. In *Proceedings of the 2004 International Conference on Embedded Systems and Applications (ESA'04)*.

MCNAMEE, D., WALPOLE, J., PU, C., COWAN, C., KRASIC, C., GOEL, A., WAGLE, P., CONSEL, C., MULLER, G., AND MARLET, R. 2001. Specialization tools and techniques for systematic optimization of system software. *ACM Trans. Comput. Syst. 19*, 2, 217–251.

MCVOY, L. AND STAELIN, C. 1996. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*.

MUCHNICK, S. S. 1997. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, San Matco, CA.

MUTH, R., DEBRAY, S., WATTERSON, S., AND DE BOSSCHERE, K. 2001. Alto: a link-time optimizer for the Compaq Alpha. *Software—Practice and Experience 31*, 1, 67–101.

PROEBSTING, T. 1995. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the ACM SIGPLAN-SIGACT 1995 Symposium on Principles of Programming Languages (POPL'95)*. 322–332.

PUGH, W. 1999. Compressing Java class files. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. 247–258.

ROS, M. AND SUTTON, P. 2003. Compiler optimization and ordering effects on VLIW code compression. In *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 95–103.

SCHMIDT, W., ROEDIGER, R., MESTAD, C., MENDELSON, B., SHAVIT-LOTTEM, I., AND BORTNIKOV-SITNITSKY, V. 1998. Profile-directed restructuring of operating system code. *IBM Systems Journal 37*, 2.

SCHWARZ, B., DEBRAY, S., ANDREWS, G., AND LEGENDRE, M. 2001. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*.

SPEER, S. E., KUMAR, R., AND PARTRIDGE, C. 1994. Improving UNIX kernel performance using profile based optimization. In *1994 Winter USENIX*. 181–188.

SRIVASTAVA, A. AND WALL, D. 1994. Link-time optimization of address calculation on a 64-bit architecture. In *Proc. of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 49–60.

TAMCHES, A. AND MILLER, B. P. 1999. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*. 117–130.

TAMCHES, A. AND MILLER, B. P. 2001. Dynamic kernel code optimization. In *Workshop on Binary Translation (WBT-2001)*.

TRIMEDIA TECHNOLOGIES INC. 2000. *TriMedia32 Architecture*. TriMedia Technologies Inc.

Received October 2005; revised April 2006; accepted June 2006