System-wide Compaction and Specialization of the Linux Kernel

Dominique Chanet Bjorn De Sutter Bruno De Bus Ludo Van Put Koen De Bosschere

Ghent University, Electronics and Information Systems Department, member of the HiPEAC network
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{dchanet,brdsutte,bdebus,lvanput,kdb}@elis.ugent.be

Abstract

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 run-time 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. 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

Keywords Linux kernel, operating system, compaction, specialization, system calls

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. And at least in theory, Linux can be tweaked for any specific embedded system without needing to write a kernel from scratch. In practice however, where time-to-market enters the picture, the kernel tweaking 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.

Unfortunately, the built-in configuration options in Linux are not engineered for producing the smallest kernel, but rather for enabling the kernel's deployment on a wide range of general-purpose systems, while still having a maintainable source code base. For example, the kernel includes code to specify boot-time parameters, but this code cannot be omitted or optimized for fixed boot-time parameters. In the embedded market, where memory remains an expensive resource, and where many systems have a fixed setup for their whole lifetime, the inability to omit or optimize such general code leads to a significant amount of overhead.

In this paper, we propose a novel method to eliminate much of that overhead. Our method is based on link-time binary rewriting, and consists of a number of established link-time compaction techniques and a number of system-wide kernel specialization techniques. 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, user-space programs written in any source language that is compiled to native code can be handled. 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.

In the major contributions of this paper,

- we present a fully 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 number of elegant ways to deal conservatively, yet aggressively with kernel code peculiarities in a link-time binary rewriter. These peculiarities include, e.g., the presence of a large number of unconventional, hand-written assembler routines.
- we evaluate the proposed method on two different platforms, i386 and ARM, thus revealing a very interesting, yet not studied, problem relating to compressed software image sizes.

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 system-level specialization techniques we developed for the Linux kernel. Section 4 discusses how system code peculiarities found in the Linux kernel can be handled in a link-time rewriter. All proposed compaction and specialization techniques are then evaluated in Section 5. Section 6 discusses related work, and conclusions are drawn in Section 7.

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.

2. Background Information

This section provides the required background information on link-time binary code rewriting, in order to facilitate the discussion of our method 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 compiled or assembled object files are being linked.

In order to enable this linking, the linker needs to extract three types of information from the object files [12]. *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 used. Relocation information is now used to detect all computable addresses in programs, and hence to approximate the possible targets of indirect control flow transfers conservatively. 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 consituting a program, together with the executable produced by the standard linker, and the linker map 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 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 [2]. Next, the linked program is disassembled and a graphical 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, the graph is transformed into a linear program representation, after which the code is assembled again. All addresses are relocated, and the rewritten executable is written to 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 on 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, e.g., the unknown node is 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 method is applied on 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, 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, i.e., through computed jumps or indirect memory accesses. Figure 2 shows an example AWPCFG.

2.3 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. 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. [7].

The more fine-grained the data section nodes in the AWPCFG are, the more aggressive such inaccessible data removal will be. At link-time, 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. 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 procedure removal [5], 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 blocks are present in a program. If there are multiple procedures, all but one of them 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 [6, 8, 16]. 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.

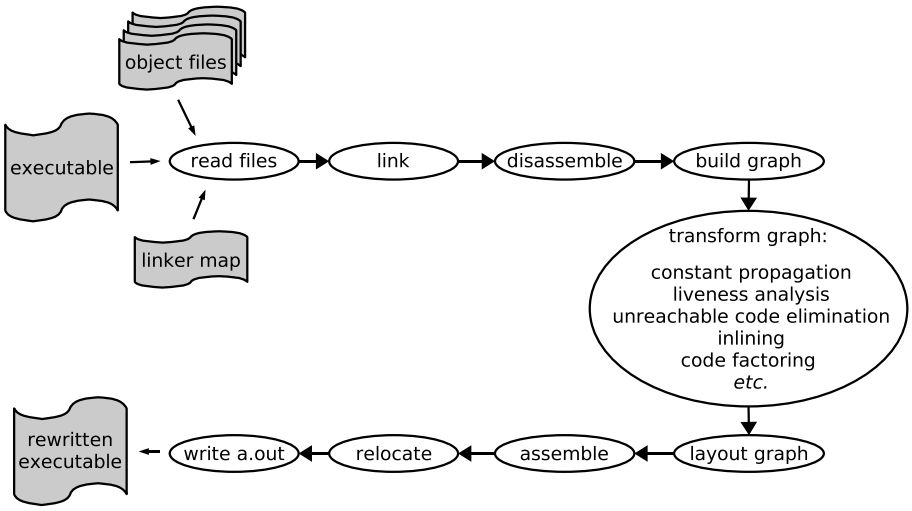

Figure 1. Overview of the operation of our link-time rewriter.

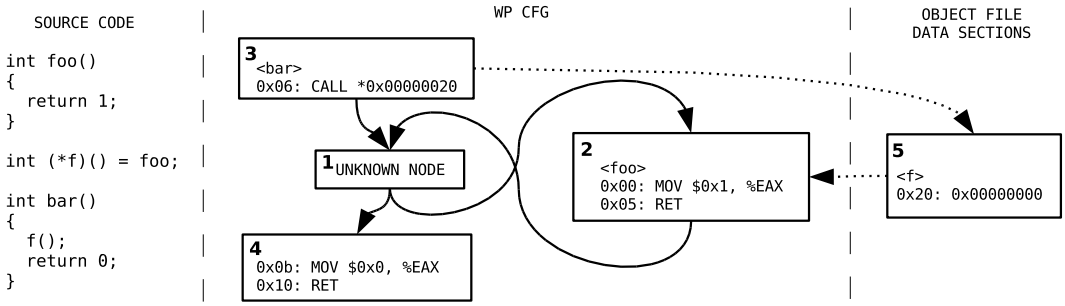

Figure 2. On the left, an example source code fragment is shown, in which `bar()` calls `foo()` through a procedure pointer. In the middle, the corresponding WPCFG is depicted. Node 1 is the unknown node, node 2 constitutes `foo()`, and nodes 3 and 4 constitute `bar()`. The full edges are the WPCFG edges. 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. Node 5 on the right of the figure is the data section containing the global variable `f`. One (dotted) data reachability edge was added because `f` holds the address of `foo()` and one was added because the address of `f` is (produced and) used in `bar()`. Together with the WPCFG, node 5 and the data reachability edges constitute the AWPCFG. Note that each data reachability edge from the data to the code or vice versa corresponds to with one WPCFG edge from or to the unknown node.

Most importantly, symbol information often allows us to determine that a procedure respects 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. Secondly, the constant data in read-only data sections can be propagated into the program during constant propagation. 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 [3, 5, 6, 7, 8, 14, 17, 21]. In order to apply them to a more complex, unconventional program such as a kernel, some special precautions need to be taken, which are discussed in Section 4. First, the next section proposes a number of additional, kernel-specific compaction techniques.

3. Kernel Compaction Techniques

In the embedded world, kernels are often installed on systems with known, fixed hardware and with a fixed set of user-space applications. Examples of such fixed-function systems are the Linksys WRT54G wireless internet gateway (http://www.linksys.com/) and the TiVo digital TV recorder (http://www.tivo.com/), both of which run the Linux kernel. In such cases, it is known *a priori* which kernel functionality is required, and which is not.

Linux kernels can be configured at build-time to only include the necessary hardware drivers. 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 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 parameters for some driver on a particular system, it is not possible to omit the code for handling such command-line parameters. Hence it is not possible to optimize the driver for its default parameter values, let alone for other, fixed val-

ues. 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.

With respect to the software needs, a fine-grained configuration is not at all available in the standard kernel. For example, most of the system calls implemented in Linux cannot be omitted with the build-time configuration, even though they may not be needed on specific systems. The system calls include, e.g., 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.

In the remainder of this section, we propose three link-time kernel compaction or specialization optimizations based on known software/hardware 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 2.3.

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. When a system call occurs, this number 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, after which control is transferred to that handler.

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, the handler will have become unreachable in the AWPCFG, and the unreachable code and data elimination discussed in Section 2.3 will eliminate it.

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 this list of system calls that can be eliminated, one has to analyze all programs that will be put on the embedded system. For architectures like the ARM, where the number of the system call is encoded literally into the 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. In theory, constant propagation will only return a conservative estimate, and as soon as the value of one system call cannot be determined, we need to assume conservatively that all system call handlers are necessary. In practice, we have found this not to be a problem. 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. On many systems, the installed system libraries are known *a priori*. When applications are only allowed to perform system calls through these libraries, 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 if their parameters are known. 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 (i.e., a register or a stack location). As such, the constant propagation algorithm itself does not even need to be adapted.

3.3 Boot-time Parameter Specialization

Our third kernel specialization technique concerns boot-time command-line parameters. These are kernel variables whose initial value can be specified at boot time. Sometimes their values can be changed at run time as well. If there is no desire to change the value at run time however, and the system will always boot with the same initial value, there is no need to treat this value as unspecified. Instead, the kernel can be optimized for its 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 run time. 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 proven 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 run time, 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.

3.4 Boot-time Parameter Elimination

Besides optimizing the kernel for fixed boot-time parameters, the handling of unused parameters itself can be eliminated if it does not involve side-effects. More specifically, the kernel initialization code includes a loop that checks all parameters specified in the boot-time command-line argument, and that calls the appropriate initialization handlers. When not all boot-time parameters are needed on an embedded system, some of the handlers can be eliminated.

To facilitate the evaluation of command-line arguments, the kernel contains a table (in the `.setup.init` section) with (parameter name, handler's address) pairs. To eliminate the handlers of unused boot parameters, the user needs to specify a list with all command line arguments that need not be adjustable at boot time. The link-time rewriter then iterates over the afore-mentioned table and removes the entries of which the parameter name is found in the user-provided list. As such, the AWPCFG references to the superfluous handlers disappear from the program, after which the handlers themselves become unreachable and ready for elimination.

3.5 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 afterwards. But 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 be needed afterwards. 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).

The code that can be executed after the initialization phase has ended, 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. Hence 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 afterwards.

4. Kernel Code Peculiarities

In most previous link-time rewriting research, a number of assumptions on the code to be rewritten are made. 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.

4.1 System Initialization Code

The operating system kernel begins execution in a very early stage of the system boot process, i.e., right after the boot loader, when the booting system is not yet fully initialized. On systems with virtual memory support, one of the 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 turned on 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 internals of the linker being used (the standard GNU linker ld), and its simplicity, or in other words its lack of complex analyses and transformations.

Unlike the simplicity of ld, 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 (on the ARM platform: 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. Instead, we simply treat it as a data section in AWPCFG. Be-

cause of the relatively small amount of code involved, the negative impact on the obtained compaction results is negligible.

We should note that this problem is not present on embedded systems that do not support virtual memory. Obviously, such systems only have one address space.

4.2 Manually Written Assembler Code

Besides the physical address space initialization code, there are numerous other occurrences of manually written assembler code in the kernel. Moreover, 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 calling convention 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 2.3 that exported procedures always respect the architecture's calling conventions.

In theory, there are three ways to treat such unconventional assembler code conservatively. 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 2.3.

The second theoretical 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 a viable option, since detecting whether or not a procedure's stack behavior respects the calling conventions would be either very complex (due to the problems of aliasing memory accesses [9]) or too imprecise [13], depending on the target architecture the kernel is compiled for.

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, e.g., a 3-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 that does not respect the calling conventions.

We have measured the number of procedures in the kernel for which this solution allowed us to assume that they maintain the calling conventions. Approximately, this was the case for one third of all procedures. In other words, one third of all procedures in the kernel are global and do not contain unconventional, hand-written code. When compiled for the ARM architecture, the total number of procedures was 4746, on the i386 architecture, it was 5214.

4.3 Special Instruction Sequences

Besides special privileged mode instructions that do not occur in user-space applications, some very rare instruction sequences occur in kernel mode that are usually not found in user-space applications. One example is the i386 instruction sequence `mov ... , %cr0 ; jmp` *to next instruction*. In this sequence the first instruction writes the control register, and the second instruction flushes the pipeline in order to allow the new status to become effective. In user-space code, this jump would be considered redundant, as it has no effect on the control flow of the program. In the kernel's context however, the jump is not redundant because of its side-effects on the underlying micro-architecture, which are usually invisible to the programmer. Luckily, only a very small number of such constructs exist, so it is trivial to modify a link-time rewriter to handle them correctly.

4.4 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. If the passed addresses are faulty, page faults may occur. 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 a user space provided address, 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.

In 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. 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 they cause page faults.

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 access.

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`.

5. Experimental Evaluation

To evaluate the proposed compaction and specialization techniques, we have implemented them on top of Diablo [3, 14, 4] (http://www.elis.ugent.be/diablo), a link-time rewriting framework developed in our research group. This section discusses the results obtained for two target systems, an ARM and an i386 system.

5.1 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 the Busybox program (http://www.busybox.net). 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 `tar`, 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 focused on small code size.

5.2 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. The resulting kernel sizes and the obtained reductions are depicted in Table 1(a) for the i386 system, and in Table 1(b) for the ARM system.

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, in which the zero-initialized sections occupy no place. The seventh column shows the size of the corresponding compressed (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.

5.2.1 Link-time Compaction Techniques

Applying well-known, general purpose link-time compaction techniques to the kernel results in a compaction of the memory footprint of about 10% on the i386, and of about 9% on the ARM platform. Approximately 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.

In practice, we learned that a kernel compiled with these flags does not operate correctly. This results from the dependence of the kernel build process on the fact that there are only two code sections, called `.text` and `.text.init`, and only a limited number of data sections, with fixed names.

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, that 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 1.5% 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 section, which explains the very small impact of the compaction techniques on the ARM data sections.

5.2.2 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 2.4% on the ARM, which corresponds to approximately 5000 and 8000 instructions that were moved to the init sections.

5.2.3 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.

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, but it did result in the removal of a number of argument validity checks in the system call handlers.

	text size		data size		init text size		init data size		image size		compressed image size		init memory footprint		memory footprint	
original kernel	767		328		47		8.1		1021		508		1150		1095	
Compaction techniques																
+ unreachable code/data elim.	731	-4.7%	306	-6.7%	46	-1.3%	8.0	-2.0%	955	-6.4%	494	-2.7%	1091	-5.1%	1037	-5.3%
+ duplicate code removal	688	-10.3%	300	-8.7%	46	-1.9%	8.0	-2.0%	907	-11.1%	516	1.7%	1042	-9.4%	988	-9.8%
+ whole-program optimization	686	-10.6%	300	-8.7%	46	-2.1%	8.0	-2.0%	903	-11.5%	514	1.2%	1039	-9.7%	985	-10.0%
Kernel specialization techniques																
+ initialization code motion	672	-12.4%	300	-8.7%	59	27.7%	8.0	-2.0%	903	-11.5%	515	1.4%	1039	-9.7%	972	-11.3%
+ system call elimination	627	-18.3%	299	-9.1%	59	27.8%	8.0	-2.0%	859	-15.8%	487	-4.1%	993	-13.7%	925	-15.5%
+ command-line specialization	624	**-18.6%**	294	**-10.3%**	59	**26.9%**	7.7	**-5.2%**	851	**-16.6%**	484	**-4.7%**	985	**-14.3%**	919	**-16.1%**
- dupl. basic block elim.	630	-17.8%	294	-10.3%	60	28.2%	7.7	-5.2%	855	-16.2%	459	-9.6%	992	-13.7%	925	-15.6%

(a) x86 results

original kernel	1311		218.4		53		5.8		1439		701		1588		1530	
Compaction techniques																
+ unreachable code/data elim.	1241	-5.4%	218.2	-0.1%	52	-1.9%	5.8	0.0%	1365	-5.2%	711	1.4%	1517	-4.5%	1459	-4.6%
+ duplicate code removal	1198	-8.6%	218.2	-0.1%	45	-14.1%	5.8	0.0%	1317	-8.5%	711	1.5%	1467	-7.6%	1416	-7.4%
+ whole-program optimization	1177	-10.3%	218.2	-0.1%	44	-16.3%	5.8	0.0%	1293	-10.2%	709	1.2%	1445	-9.0%	1395	-8.8%
Kernel specialization techniques																
+ initialization code motion	1140	-13.0%	218.2	-0.1%	81	54.7%	5.8	0.0%	1293	-10.2%	709	1.1%	1446	-9.0%	1359	-11.2%
+ system call elimination	1093	-16.6%	218.2	-0.1%	81	54.7%	5.8	0.0%	1245	-13.5%	683	-2.5%	1398	-11.9%	1311	-14.3%
+ command-line specialization	1064	**-18.9%**	218.2	**-0.1%**	80	**51.6%**	5.5	**-4.6%**	1217	**-15.5%**	667	**-4.8%**	1367	**-13.9%**	1282	**-16.2%**
- dupl. basic block elim.	1096	-16.5%	218.2	-0.1%	81	55.0%	5.5	-4.6%	1249	-13.2%	664	-5.3%	1401	-11.8%	1314	-14.1%

(b) ARM results

Table 1. Code and data sizes of different parts and different forms of the kernel for both our evaluation systems. All sizes are in kilobytes, 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 last row, for which we disabled duplicate basic block elimination.

5.2.4 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 0.3% 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 with 14.3% during and 16.1% after system initialization for the i386, and with 13.9% during and 16.2% after initialization for the ARM. The kernel image size is reduced with 16.6% and 15.5% respectively, and the compressed image size with 4.7% and 4.8% respectively.

5.3 System Requirements

All compaction experiments were conducted on a 2.8 GHz Pentium 4 system with 2 GiB of RAM, running Ubuntu Linux 4.10. The compaction time was 189 seconds for the ARM kernel with all optimizations and specializations enabled, and 101 seconds for the i386 kernel. The i386 kernel takes less time because less data-flow analyses have been implemented in Diablo for the i386. Hence less analyses and optimizations are performed on the i386 kernel. The maximum memory usage observed is 240 MiB.

5.4 Impact on Compressed Image Size

It is clear that the gains obtained for the compressed image size are somewhat disappointing. To some extent the difference between image size reduction and compressed image size reduction is understandable. Compressing a file involves reducing the redundant information from the byte stream. Compaction techniques such as duplicate code removal also remove redundancy from a program, albeit on a different level. To try to understand this effect, we applied all compaction techniques, minus the duplicate basic block elimination. The resulting compaction is presented on the bottom row of Table 1.

Much to our surprise, we observe that duplicate basic block removal reduces the size of the uncompressed image, but actually *increases* the size of the compressed image! On the i386, the gain on the compressed image nearly doubles when duplicate basic block elimination is disabled. This follows from the fact that this program transformation removes a lot of compressible information from the program (identical pieces of code), and substitutes them with smaller, but completely incompressible information (such as call instructions to the newly formed procedures).

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 compressed image's size however, e.g., because the compressed image will be stored in more expensive Flash memory, this transformation is best disabled.

For the i386, the impact of duplicate basic block removal on the compressed image size completely accounts for the size increases seen at different levels of compaction in Table 1. It clearly does

not do so for the ARM however. On the ARM system, the compressed image already grows when the unreachable code and data is removed from the program, notwithstanding the fact that the corresponding uncompressed image was reduced with 5%. 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 problem.

5.5 Performance Impact

To measure the performance impact of the modifications to the kernel, we have performed several runs of LMbench 2.0.4 [15] on our evaluation systems. This benchmark set measures a number of aspects of the kernel's behaviour like system call performance, interprocess communication latencies and context switching times.

The chart in Figure 3 shows the performance degradation observed after our specialization and compaction techniques have been applied. For each system, the first bar indicates the degradation observed when all our techniques have been applied, including code factoring at the basic block level [5]. This transformation introduces run-time overhead in a program by inserting additional calls and returns to and from the factored procedures. Unlike the other compacting transformations, that we expect to speed up programs, factoring at the basic block level hence usually results in slowdowns. To evaluate this effect, we included two more bars, that show the performance degradation when all our techniques are applied, except basic block code factoring.

As can be seen from the chart, basic block code factoring clearly results in a performance degradation. On average, the performance measured by LMbench drops with 4.3% on the i386 system, and with 1.9% on the ARM system. When no basic block code factoring is applied however, no degradation occurs.

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 7%. These improvements mainly result from procedure inlining and interprocedural data flow optimizations within the call chains of the evaluated system calls.

6. Related Work

The idea of specializing the Linux kernel for a specific application was first explored by Lee et al. [11]. Based on source code analysis, a system-wide call graph is built that spans the application, the libraries and the kernel. On this graph 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.

Rajagopalan et al. [18] proposed a method to optimize a whole system for speed by profiling the address space boundary crossings (e.g., system calls), and using link-time optimization techniques to reduce the amount of crossings or the cost of each individual crossing. As a case study, system call clustering is presented. This technique groups a number of related system calls in a program and adds them to the kernel as one "super-system call", thus reducing the number of system calls the application has to perform.

Multiple link-time binary rewriters that focus on speed optimization have been implemented. We only discuss those systems that were at some time used to optimize kernels.

Spike [1] 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 [10] 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.

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 [8] and Squeeze++ [5, 6], 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 that 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 [3, 4] and MIPS [14]. 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 [4] applies post-pass 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 KernInst dynamic kernel instrumentation system [23] has been used to optimize parts of the UltraSPARC Solaris kernel [24]. 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.

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 [20] and HP-UX [22] platforms.

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 ARM-like architecture. But a similar problem, relating to the combined use of compiler optimization and hardware-supported code compression has been studied for VLIW architectures [19]. In their report, Ros and Sutton report that, in general, instruction scheduling did not influence compressibility when the compiler had already optimized the code for code size.

7. 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 run-time environment of embedded systems is known *a priori*. In an experimental setup, the

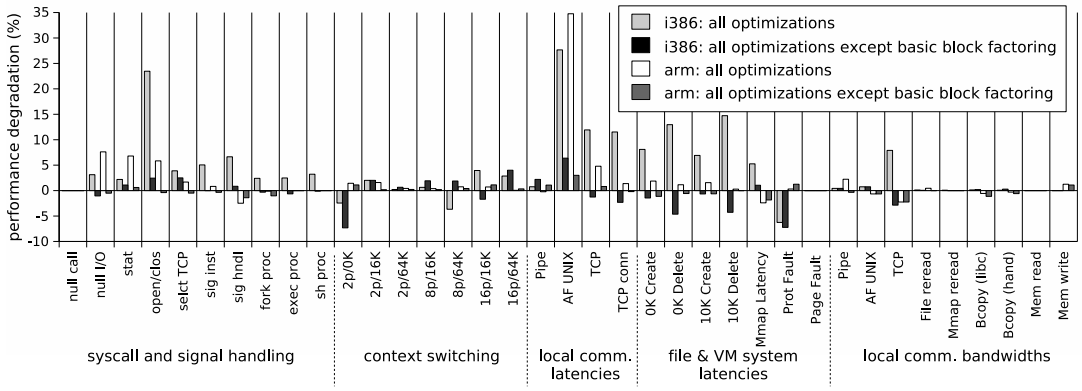

Figure 3. Performance degradation for the lmbench benchmark suite.

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.

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.

Acknowledgement

Bjorn De Sutter, as a Postdoctoral Research Fellow, and Dominique Chanet, being a PhD. student, are supported by the Fund for Scientific Research - Flanders (FWO). Bruno De Bus and Ludo Van Put are supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research is also partially supported by Ghent University.

References

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

[2] B. De Bus. *Reliable, Retargetable and Extensible Link-Time Program Rewriting*. PhD thesis, Ghent University, 2005.

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

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

[5] B. De Sutter, B. De Bus, and K. De Bosschere. 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)*, pages 275–291, 2002.

[6] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems*, 2005. To appear.

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

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

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

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

[11] C.-T. Lee, J.-M. Lin, Z.-W. Hong, and W.-T. Lee. An application-oriented linux kernel customization for embedded systems. *Journal of Information Science and Engineering*, 20(6):1093–1107, 2004.

[12] J. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

[13] C. Linn, S. Debray, G. Andrews, and B. Schwarz. Stack analysis of x86 executables. Available at http://www.cs.arizona.edu/people/debray.

[14] M. Madou, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Link-time optimization of MIPS programs. In *Proceedings of the 2004 International Conference on Embedded Systems and Applications (ESA'04)*, 2004.

[15] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[16] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.

[17] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. alto: a link-time optimizer for the compaq alpha. *Software - Practice and Experience*, 31(1):67–101, 2001.

[18] K. Rajagopalan, S. Debray, M. Hiltunen, and R. Schlichting. Cassyopia: Compiler assisted system optimization. In *Proc. 9th Usenix Workshop on Hot Topics in Operating Systems* , 2003.

[19] M. Ros and P. Sutton. Compiler optimization and ordering effects on vliw code compression. In *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*, 2003.

[20] W. Schmidt, R. Roediger, C. Mestad, B. Mendelson, I. Shavit-Lottem, and V. Bortnikov-Sitnitsky. Profile-directed restructuring of operating system code. *IBM Systems Journal*, 37(2), 1998.

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

[22] S. E. Speer, R. Kumar, and C. Partridge. Improving unix kernel performance using profile based optimization. In *1994 Winter USENIX*, pages 181–188, 1994.

[23] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

[24] A. Tamches and B. P. Miller. Dynamic kernel code optimization. In *Workshop on Binary Translation (WBT-2001)*, 2001.