Automatic Operating System Specialization via Binary Rewriting

Mohan Rajagopalan   Saumya K. Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{*mohan, debray*}@*cs.arizona.edu*

Matti A. Hiltunen   Richard D. Schlichting
AT&T Labs-Research
180 Park Avenue
Florham Park, NJ 07932, USA
{*hiltunen, rick*}@*research.att.com*

Abstract

This paper explores the application of binary rewriting techniques to customization of operating system kernels. Specifically, this paper describes a new binary rewriting system, *Charon*, and its application to the synthesis of application-specific operating systems. Compiler techniques are used to analyze and transform the kernel based on holistic knowledge of the system. Preliminary experiments have been promising and argue persuasively that more opportunities for automation should be explored.

1 Introduction

Running costs, such as the memory footprint and performance overheads, are a significant concern for operating system designers in the context of embedded systems. Accordingly, OS customization techniques are becoming increasingly important in this context since there is a growing trend to replace specialized, application specific operating system software with customized versions of general purpose operating systems such as Linux. This customization has traditionally been a complicated, labor intensive, and time consuming manual process that is usually error prone. This paper explores the possibility of automating the customization process through holistic systems analysis and the application of compiler techniques.

General purpose OSes are designed without making assumptions about their deployment environment or the applications that will run on them. Almost all specialization research [4, 12, 13, 15] is based on the fact that there is a natural tradeoff between performance and generality in operating systems—operating systems can be made to perform better by making specific assumptions about the target system and its operating conditions. Accordingly, we refer to *customization* as the process of adapting and tuning the behavior of a system to make it perform better under certain assumptions. Specifically, in the context of this paper, customization has two goals : *compaction*, reducing the memory footprint of an operating system and *performance optimization*, improving performance characteristics.

Here, we describe an automatic approach to OS customization based on holistic system analysis, that takes advantage of the observation that the set of applications that run on an embedded system—or, more precisely, the set of operating system services required by such applications—is fixed, and system requirements do not change drastically or unpredictably. Program analysis is used to predict system requirements by looking at user-level applications, the underlying hardware, and the operating system itself. Next, compiler techniques such as constant propagation, code elimination, and value specialization are used to *compact* and *optimize* the kernel for the given set of requirements. The process is completely automatic and implemented using our Charon binary rewriting tool.

Traditional approaches to OS customization have taken a bottom-up approach targeting composability [1, 3, 9, 10, 12] and configurability [2, 4, 7, 12, 13, 15]. The operating system is constructed by combining specialized components, each tailored to a set of requirements. In contrast, the approach proposed in this paper is top-down: starting with the most generic operating system, the goal is to use automated analysis and code transformations to remove unneeded generality, thereby specializing the system to the particular set of of hardware and application contexts of its use. There are several advantages to such an approach. First, such automatic specialization allows us to have the generality, flexibility, and economy of using a general purpose operating system, as well as the performance advantages of an operating system customized to the specific hardware and software—in essence, allowing us to have our cake and eat it too. Second, the transformations are guaranteed to be correctness preserving, i.e., the resulting operating system will behave in the same way as the original system. Finally, our solution does not require significant modifications to existing systems and can be totally transparent to the user.

This paper describes our experiences with binary rewriting and automatic customization. We begin with a description of the new binary rewriting framework, Charon. Next, a specific instance of customization, *kernel compaction*, is described. This is followed with a brief description of other opportunities under the general theme.

2 Binary Rewriting: Charon

Charon is a binary rewriting system targeted at processing operating system kernels. The system is built as an extension to the PLTO binary rewriting toolkit [14]. Figure 1 describes the steps involved in processing binaries. The input to the system is the operating system kernel, the set of user level application binaries, and a specifications file. The last of these describes aspects of the underlying hardware relevant for the specialization process (see section 3 for details).

The input binaries are typically statically linked relocatable ELF files. Relocation information is needed to distinguish between 32 bit data values and pointers. Creating relocatable binaries is easy and done by simply setting the appropriate flag during compilation and linking. In the case of Linux, generating the appropriate input kernel was straightforward and required changing exactly one line in the regular Makefile. Since the kernel is by default a statically linked ELF file, it was merely a matter of modifying the parameters to the linker.

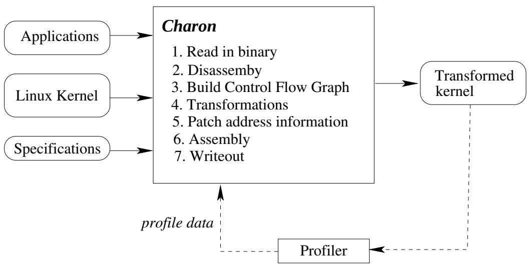

Figure 1: System overview

The first step of processing is to set up internal data structures that correspond to the sections in the ELF file. Next, the executable sections are disassembled to recover the instruction stream. Kernel binaries are unlike regular application binaries in a number of ways. For example, there is a significant amount of data embedded within executable sections, there are implicit addressing constraints, and occasional unusual instruction sequences, a result of hand-coded assembly and self modifying code. Conventional disassembly algorithms were unable to recover large parts of the kernel and hence, we use an indigenous type-based disassembly algorithm. Initially, everything in a section is conservatively marked as data. Type information (for example from the symbol table) and iterative control flow analysis is then used to discover code. The current version of Charon is able to disassemble approximately 94% of the executable sections. The remaining 6% includes data blocks (several 4K pages) and padding NOP instructions, in addition to the executable code that cannot currently be disassembled. An example of code that cannot currently be disassembled includes the code for the so-called "Duff's device."[1] Portions of the text section that cannot be disassembled are treated as data and as such, are reinserted into the kernel executable when it is reassembled; however, any code pointers in such undisassembled code/data are identified as such, and updated correctly, using the associated relocation information.

Disassembly results in an instruction stream, which is then processed to build a control flow graph (CFG). The internal representation is comprised of instructions, basic blocks, functions and different types of edges that denote different control flow transitions. This CFG forms the basis for all further analysis and transformations. The CFG of an operating system has significant differences compared to the CFGs of regular programs. For example, the kernel has multiple entry points (through system calls and interrupts), whereas regular applications typically have a single entry point. In the kernel CFG, system call and interrupt handlers, identified through the interrupt descriptor and system call table, are marked as potential entry points into the kernel. In the case of regular applications, the *__start* function is the only entry point for the application.

Once the control flow graph has been built, various transformations can be carried out on the program. The final stage of processing is *writeout*, where the processed binary is assimilated and an executable image is written out to a file. Code addresses are recomputed, and pointers into the code (in both the

[1] An optimized data copy loop invented by Tom Duff.

text and data areas), identified using relocation information, are updated appropriately. Finally the appropriate ELF headers and table entries are created and the program is written out to a file.

While binaries contain less high-level semantic information than source programs, they have the advantage that the entire program is available for analysis and optimization. For this reason, prior work in binary rewriting has been successful in the context of whole program optimization [5, 11, 16]. Our experience with using binary rewriting for operating systems has been favorable for similar reasons. Operating system kernels are self-contained—they are usually statically linked, i.e., all the code is present in one place—and contain complete symbol table information. Second, while OSes are typically large programs in which different components have been written in different languages (including hand-written assembly code, which can be difficult for source-level analyzers to process), from a binary rewriting point of view, the code is uniformly seen as a sequence of machine instructions. Third, the low level address information available can be used to our benefit; for example, some indirect jumps, such as those to system call handlers, can be resolved by looking at data structures like the *system_call_table*.

While our experience is that the information lost from the lack of source code has not posed a significant issue, one can envision cases where it might be an issue. For example, consider the issue of refactoring code, i.e., using semantic preserving transformations to improve the internal organization of the code. Operating systems are complex software systems and there is an increasing interest in trying to automate the refactoring process to, for example, eliminate code duplication. Refactoring can be hard at the binary level because of the difficulty in comparing two binary snippets even though they may have arisen from the same macro definitions or may be inlined versions of the same function.

3 Case Study: Compaction

Overview. For a variety of practical reasons related to both cost and support, there is a growing trend to replace vendor-specific custom operating systems for embedded devices with general purpose counterparts such as Linux. Embedded systems are usually restricted in terms of the memory available both for computing and storage. Hence, transformations that reduces the memory footprint are highly desirable. Traditionally this has been a manual task that is labor intensive, time consuming, and error prone.

The goal of our customization is to automatically reduce the memory footprint of an OS kernel. We do so by identifying and discarding parts of the OS that will never be needed for the specific set of OS services required to support a given set of applications on a given hardware platform. This is difficult to do for a general purpose system where arbitrary new applications may be installed and executed, but we observe that embedded systems typically require only a (small) subset of the services provided by a general purpose OS. For example, software running on a cell phone or a sensor network mote will most likely have no need for code to interact with a mouse. Such unneeded code can be identified and eliminated via compiler based transformations based on dead and unreachable code elimination. Although dead code elimination is a well studied and common compiler transformation, it can not be applied directly in this context for a couple of reasons—first, the system being analyzed involves multiple address spaces and second, the number of indirect jumps in the kernel make conservative analysis ineffective. Kernel specific semantic information is required to make the transformation useful. Our approach is to adapt the standard transformation and combine it with cross-address space static analysis to achieve the net goal of customization for reduced memory footprint.

Compaction. The first step in this transformation is to identify all possible entry points into the kernel. Next, starting at each entry point, depth first reachability analysis is carried out to mark all reachable parts of code.

Entry points into the OS are either system calls or interrupts. Identifying all possible system calls the OS needs to field is relatively straightforward. Static analysis is performed on each of the applications to compute the set of possible system calls the application can make. System calls are identified by looking for the `int 0x80` software trap instruction. Constant propagation is used to identify the system call number associated with each system call.

Predicting interrupts is more complicated. Interrupts are usually not generated directly, but rather are invoked implicitly in response to hardware events such as receiving a network packet or the result of side effects of system calls such as a read system call causing a page fault. To identify the set of interrupts that could be encountered we use two pieces of information: first, the set of all possible hardware interrupts that is supplied as part of the specifications file, and second, a table of static mappings associating systems calls to interrupts they may generate.

Once kernel entry points have been marked, we carry out a reachability analysis starting at these entry points. This is essentially just a depth-first traversal of the control flow graph of the kernel, with the modification that function calls and the corresponding return blocks are handled in a context-sensitive manner: the basic block that follows a function call is marked reachable only if the corresponding call site is reachable. A significant portion of the control flow in the kernel is reached through indirect jumps, and to ensure correctness the reachability analysis must be conservative in its estimate of the possible targets of such jumps. While it is easy to reason about things like system call that will never be called, reasoning about interactions among co-operating modules in the kernel is more sophisticated and is currently under investigation.

After reachability analysis, code identified as unreachable can be eliminated. Additionally, any code whose results—including any possible side effects—are used only in unreachable code is *dead*, and can also be eliminated.

Preliminary results. To obtain a preliminary measure of the potential impact of our approach, we started by measuring the system call requirements for a number of applications. Table 1 shows the number of unique calls in each application. Since Linux 2.4.22, the OS kernel being used, has 270 system calls, this means that less than 25% of the total possible system calls were seen in each application.

While we do not have actual compaction data at the time of this writing, a prototype implementation is nearing completion, and we expect to have detailed experimental results shortly. We are encouraged by the results of manual application of the ideas described here, which have yielded significant reductions in the size of the kernel. For example, we transformed a binary of roughly 1.1 Mbytes to a little over 500 Kbytes, for a size reduction of roughly 45%.

Further compaction. To date, we have been conservative and restricted our attention to specific parts of the kernel identified through simple reachability analysis. One of the goals of our research is to expand this scope and make more dramatic semantic transformations. For example, if it is known that none of the applications make explicit use of the networking facilities, we would like the customizer to eliminate the networking stack. Another possibility to be explored is the use of the specification file to provide high-level information that can then be assimilated by Charon and used for increasing the scope of customization. For example, the specification file could be used to indicate the hardware present that could be used to discard unnecessary drivers. A side effect of removing such drivers is that indirect jumps can in turn be resolved by converting them to directed jumps.

Program	No. of unique syscalls
bison	31
calc	54
screen	67
tar	58

Table 1: Unique system calls per program

4 Future Directions

A binary rewriting system such as Charon opens up many possibilities beyond compaction. In this section we discuss other potential avenues for application.

Performance optimization. Another aspect of customization is that of improving the performance. A straightforward extension to compaction would be to optimize for frequently encountered values for system call parameters. Static or dynamic profile based analysis (e.g., through value profiling) can be used to identify such values, which can then be used to guide specialization through techniques such as constant propagation, dead code elimination, and value-based code specialization.

η-kernel: application specific OS. When the set of target applications is small, it might be feasible to generate a custom operating system for each set of applications. We use the term η-kernel to describe a highly optimized and specialized operating system. First, application requirements are used to customize the operating system, which is then optimized through static analysis techniques. Next, profiling and dynamic analysis are used to further tune the operating system.

OS structuring. The availability of automated tools allows for drastic changes in the way operating systems are developed. For example, a simple application would be to bridge the gap between micro-kernels and monolithic kernels. While micro-kernels have traditionally been associated with clean composition, modular design, and ease of maintenance and structuring, they have largely been unsuccessful due to performance reasons. Monolithic kernels such as Linux on the other hand, are successful for performance reasons. A simple extension of our approach would be to structure operating systems as micro-kernels and then relying on automated tools to optimize the OS before deployment, possibly by converting the modular structure into a monolithic unit.

Improved understanding of OS structure. The Linux kernel is a very complicated piece of software consisting of close to 6 million lines of code and numerous separate source files. While modular, the kernel code is tightly interconnected with different components of the kernel using other components, thereby inducing dependencies between files. Static analysis can help by making such dependencies explicit. Specifically, the kernel CFG can be used to see what other parts of a kernel use a specific piece of code and vice versa. This makes it easier to predict if a change in a kernel source file will have undesirable side effects on the rest of the kernel.

Locating kernel bugs. Source code analysis has been used to detect possible bugs in code, including the Linux kernel source [6, 8]. Charon makes it possible to do similar analysis without access to all the source code. The fact that we work on binaries allows us to potentially examine proprietary code, e.g. device drivers, that are written in assembly or for which the source is not available.

5 Concluding Remarks

This paper has presented our preliminary experience in developing a new binary rewriting framework for transforming operating system kernels. This framework is currently being used to develop automatic techniques for application-based specialization. Specifically, we have explored the possibility of automating the customization process for porting Linux to embedded systems. The novel contribution of this research is that it explores the possibility of completely automating processes that have traditionally been performed by hand by skilled experts. A number of potential applications have been identified and are under investigation. We believe that this framework will open up new and exciting directions, particularly in the context of small mobile devices.

References

[1] A. Arpaci-Dusseau, R. Arpaci-Dusseau, N. Burnett, T. Denehy, T. Engle, H. Gunawi, J. Nugent, and F. Popovici. Transforming policies into mechanisms with infokernel. In *Proc. SOSP*, p. 90–105, Oct 2003.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. SOSP*, p. 267–284, Dec 1995.

[3] R. Campbell, N. Islam, and P. Madany. *Choices*, frameworks and refinements. *Computing Systems*, 5(3):217–257, 1992.

[4] R. Campbell and S. Tan. μ-Choices: An object-oriented multimedia operating system. In *Proc. 5th HotOS*, May 1995.

[5] R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997.

[6] Coverity, Inc. Linux report. http://linuxbugs.-coverity.com/linuxbugs.htm, Dec 2004.

[7] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. SOSP*, p. 251–266, Dec 1995.

[8] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proc. SOSP*, p. 90–105, Oct 2003.

[9] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. SOSP*, p. 38–51, Oct 1997.

[10] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proc. USENIX Annual Tech. Conf*, June 1999.

[11] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere. `alto` : A link-time optimizer for the Compaq Alpha. *Software—Practice and Experience*, 31:67–101, Jan 2001.

[12] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. SOSP*, p. 314–324, Dec 1995.

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

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

[15] C. Small and M. Seltzer. VINO: An integrated platform for operating systems and database research. TR-30-94, Harvard CS Laboratory, Cambridge, MA, 1994.

[16] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.