MEMORY FOOTPRINT REDUCTION OF OPERATING SYSTEM KERNELS

by

Haifeng He

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCES
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

2009
As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Haifeng He entitled Memory Footprint Reduction of Operating System Kernels and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Saumya K. Debray
Date: 16 January 2009

Gregory R. Andrews
Date: 16 January 2009

John H. Hartman
Date: 16 January 2009

Chris Gniady
Date: 16 January 2009

Final approval and acceptance of this dissertation is contingent upon the candidate’s submission of the final copies of the dissertation to the Graduate College. I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director: Saumya K. Debray
Date: 16 January 2009
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Haifeng He
ACKNOWLEDGEMENTS

The work described in this dissertation can not be completed without the teaching, help, guidance, and encouragement from a great many people. First of all, I am deeply grateful to my advisor Prof. Saumya Debray for his mentorship, guidance and encouragement to help me to see things through. Immense gratitude to Prof. Gregory Andrews for his always considerate and generous help and guidance. To both Prof. Debray and Prof. Andrews for their trust and confidence in me to help me work on my research.

I am very grateful to Prof. John Hartman and Prof. Chris Gniady for their invaluable guidance and advice.

Many aspects of this work have benefited immensely from people working hard together. I would like to express my gratitude to folks at the Solar group: Dr. Mohan Rajagopalan, Somu Perianayagam, John Trimble, and Gregg Townsend. I am incredibly honored to work with them.

Finally, I would like to give my special thanks to my family: to my father, Yanyu He, my mother, Baozhen Fan, and my brother, Haixin He, for always standing behind me and encouraging my interests; to my wife Yun, for her sacrifices, support, and patient love to enable me to complete this work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>9</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>10</td>
</tr>
<tr>
<td>CHAPTER 1  INTRODUCTION</td>
<td>12</td>
</tr>
<tr>
<td>1.1 Reducing Memory Footprint of Application Programs</td>
<td>13</td>
</tr>
<tr>
<td>1.2 Reducing Memory Footprint of OS kernels</td>
<td>15</td>
</tr>
<tr>
<td>1.2.1 Opportunities for Reducing Memory Footprint of OS kernels</td>
<td>16</td>
</tr>
<tr>
<td>1.3 Major Contributions</td>
<td>17</td>
</tr>
<tr>
<td>1.4 Outlines</td>
<td>18</td>
</tr>
<tr>
<td>CHAPTER 2  BACKGROUND</td>
<td>20</td>
</tr>
<tr>
<td>2.1 The Program Characteristics of OS Kernels</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Binary Rewriting of OS Kernels</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Technical Challenges of Binary Rewriting of OS kernels</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Disassembly</td>
<td>23</td>
</tr>
<tr>
<td>2.3.2 Control Flow Analysis</td>
<td>25</td>
</tr>
<tr>
<td>2.4 Code Transformations of OS Kernels</td>
<td>30</td>
</tr>
<tr>
<td>2.4.1 Instrumentation</td>
<td>32</td>
</tr>
<tr>
<td>2.5 FA Pointer Analysis</td>
<td>32</td>
</tr>
<tr>
<td>CHAPTER 3  CODE COMPACTION OF OS KERNELS</td>
<td>36</td>
</tr>
<tr>
<td>3.1 Challenges of Code Compaction of OS Kernel</td>
<td>36</td>
</tr>
<tr>
<td>3.2 Pointer Analysis: Resolving Indirect Function Call Targets</td>
<td>37</td>
</tr>
<tr>
<td>3.2.1 Approximate Decompilation of Kernel Code for FA Analysis</td>
<td>38</td>
</tr>
<tr>
<td>3.3 Identifying Reachable Code</td>
<td>41</td>
</tr>
<tr>
<td>3.3.1 Reachable Analysis</td>
<td>41</td>
</tr>
<tr>
<td>3.3.2 Improving the Analysis</td>
<td>42</td>
</tr>
<tr>
<td>3.4 Kernel Compaction</td>
<td>43</td>
</tr>
<tr>
<td>3.5 Experimental Results</td>
<td>44</td>
</tr>
<tr>
<td>3.5.1 Code Size Reduction</td>
<td>47</td>
</tr>
<tr>
<td>3.5.2 Effects of Different Optimizations</td>
<td>48</td>
</tr>
<tr>
<td>3.5.3 Effects of Different Indirect Call Target Analyses</td>
<td>49</td>
</tr>
<tr>
<td>3.5.4 System Call Popularity</td>
<td>50</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS – Continued

3.6 Summary ........................................................................................................ 52

CHAPTER 4  ON-DEMAND CODE LOADING OF OS KERNELS ............... 53

4.1 On-Demand Code Loading ................................................................. 54
  4.1.1 Upper-Bound of Memory Requirement for Kernel Code .......... 55
  4.1.2 Code Clustering ................................................................. 56
  4.1.3 Code Transformation ......................................................... 60
  4.1.4 Context Switches and Interrupts .................................. 63

4.2 Experimental Results ........................................................................ 65
  4.2.1 Memory Reduction Results ........................................... 65
  4.2.2 Cost of Code Loading ......................................................... 66

4.3 Summary ................................................................................................. 69

CHAPTER 5  DYNAMIC DATA COMPRESSION OF OS KERNELS .......... 70

5.1 Linux Kernel Slab Allocator .............................................................. 71

5.2 Data Structure Compression .......................................................... 72
  5.2.1 Data Structure Profiling .................................................. 73
  5.2.2 Data Compression Techniques ...................................... 74
  5.2.3 Compression and Decompression Procedures .................. 79

5.3 Data Structure and Source Code Transformation .......................... 79
  5.3.1 Compression and Expansion ........................................ 82
  5.3.2 Source Code Transformation ......................................... 85
  5.3.3 Soundness Considerations ............................................. 85

5.4 Experimental Evaluation ................................................................. 87
  5.4.1 Select the Dynamic Data Structures to Compress .............. 87
  5.4.2 Experimental Results ...................................................... 88
  5.4.3 Discussion ................................................................. 90

5.5 Summary ................................................................................................. 92

CHAPTER 6  RELATED WORK ....................................................................... 93

6.1 Program Size Reduction ................................................................. 93
  6.1.1 Compaction-based Techniques ...................................... 93
  6.1.2 Compression-based Techniques ...................................... 94

6.2 Operating Systems Optimization and Specialization ................. 95

6.3 Memory Footprint Reduction for Embedded Systems .................. 96

CHAPTER 7  CONCLUSIONS AND FUTURE DIRECTIONS ............... 98

7.1 Conclusions ......................................................................................... 98
  7.1.1 Static Code Compaction ............................................... 98
  7.1.2 On-Demand Code Loading ........................................... 99
TABLE OF CONTENTS – Continued

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.3</td>
<td>Dynamic Data Structure Compression</td>
<td>99</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Directions</td>
<td>100</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>BENCHMARK DATA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A.1 Experimental Results of Code Compaction</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>A.2 Experimental Results of On-Demand Code Loading</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>A.3 Experimental Results of Dynamic Data Structure Compression</td>
<td>106</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>107</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 Example of a function whose code is spread over multiple subsections . . 27
2.2 Control flow during the handling of exceptions in the Linux kernel . . . 28
2.3 An example of analysis complications due to exception edges . . . . . 31

3.1 Using approximate decompilation for program analysis . . . . . . . 37
3.2 Examples of approximate decompilation of assembly code to C code . . 40
3.3 The improved reachable analysis algorithm . . . . . . . . . . . . 43
3.4 Results of code size reduction . . . . . . . . . . . . . . . . . . . . . 47
3.5 System call “popularity” in embedded applications . . . . . . . . 51

4.1 The BlockSize function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.2 The Clustering Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.3 The problem with function calls across dynamically loaded clusters . . 62
4.4 Source code changes to handle multi-threading . . . . . . . . . . . . 64
4.5 Code size reductions for different core code growth bounds (\(Bu fSz=2KB\)) . 66
4.6 Estimated runtime overhead of on-demand code loading . . . . . . . . 68

5.1 Data structure dentry in the Linux kernel . . . . . . . . . . . . . . . . . 75
5.2 Procedures for compression and decompression . . . . . . . . . . . . . . 80
5.3 Memory layout of compressed data structure before and after expansion. . 81
5.4 Runtime impacts of dynamic data structures compression . . . . . . . . 90
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Number</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Code and data section size of minimally-configured Linux kernel 2.4.31</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>Code size and number of indirect function calls of MiBench suite</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Static kernel characteristics</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>Characteristics of the benchmarks used (from the MiBench suite)</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>Effects of different optimizations on code size</td>
<td>48</td>
</tr>
<tr>
<td>3.4</td>
<td>Effects of different call target analyses on code size</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Running time of all benchmarks on an unmodified kernel</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Profiling data of <code>dentry</code> structure (value table size = 4096)</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Bit widths computed for the fields of <code>dentry</code> structure</td>
<td>78</td>
</tr>
<tr>
<td>5.3</td>
<td>Attributes of different compression schemes</td>
<td>81</td>
</tr>
<tr>
<td>5.4</td>
<td>Procedures for accessing data in compressed space and expanded space</td>
<td>84</td>
</tr>
<tr>
<td>5.5</td>
<td>Rewrite rules for dynamic data structure compression</td>
<td>86</td>
</tr>
<tr>
<td>5.6</td>
<td>Compressed slab caches (sorted by cache size in non-increasing order)</td>
<td>87</td>
</tr>
<tr>
<td>5.7</td>
<td>Average memory savings and overheads for benchmark sets</td>
<td>89</td>
</tr>
<tr>
<td>5.8</td>
<td>Static impacts on data structure size and code size</td>
<td>91</td>
</tr>
<tr>
<td>A.1</td>
<td>Code compaction results</td>
<td>102</td>
</tr>
<tr>
<td>A.2</td>
<td>Clustering and memory reduction for different core code growth bounds</td>
<td>104</td>
</tr>
<tr>
<td>A.3</td>
<td>Cost of dynamic code loading for different core code growth bounds</td>
<td>105</td>
</tr>
<tr>
<td>A.4</td>
<td>Memory pages usage and size reduction of Linux slab allocators</td>
<td>106</td>
</tr>
<tr>
<td>A.5</td>
<td>Running time and overhead of benchmark programs</td>
<td>106</td>
</tr>
</tbody>
</table>
ABSTRACT

As the complexity of embedded systems grows, there is an increasing use of operating systems (OSes) in embedded devices, such as mobile phones, media players and other consumer electronics. Despite their convenience and flexibility, such operating systems can be overly general and contain features and code that are not needed in every application context, which incurs unnecessary performance overheads. In most embedded systems, resources, such as processing power, available memory, and power consumption, are strictly constrained. In particular, the amount of memory on embedded devices is often very limited. This, together with the popular usage of operating systems in embedded devices, makes it important to reduce the memory footprint of operating systems. This dissertation addresses this challenge and presents automated ways to reduce the memory footprint of OS kernels for embedded systems.

First, we present kernel code compaction, an automated approach that reduces the code size of an OS kernel statically by removing unused functionality. OS kernel code tends to be different from ordinary application code, including the presence of a significant amount of hand-written assembly code, multiple entry points, implicit control flow paths involving interrupt handlers, and frequent indirect control flow via function pointers. We use a novel “approximate decompilation” technique to apply source-level pointer analysis to hand-written assembly code. A prototype implementation of our idea on an Intel x86 platform and a minimally configured Linux kernel obtains a code size reduction of close to 24%.

Even though code compaction can remove a portion of the entire OS kernel code, when exercised with typical embedded benchmarks, such as MiBench, most kernel code is executed infrequently if at all. Our second contribution is on-demand code loading, an automated approach that keeps the rarely used code on secondary storage and loads it into main memory only when it is needed. In order to minimize the overhead of code loading,
a greedy node-coalescing algorithm is proposed to group closely related code together. The experimental results show that this approach can reduce memory requirements for the Linux kernel code by about 53% with little degradation in performance.

Last, we describe dynamic data structure compression, an approach that reduces the runtime memory footprint of dynamic data structures in an OS kernel. A prototype implementation for the Linux kernel reduces the memory consumption of the slab allocators in Linux by 17.5% when running the MediaBench suite while incurring only minimal increases in execution time (1.9%).
Embedded systems have become ubiquitous and have penetrated almost all aspects of our lives. In most embedded systems, resources such as processing power, available memory, and power consumption are strictly constrained. In particular, the amount of memory on embedded devices is often very limited by considerations such as size, weight, or cost. Even though memory costs have been dropping steadily due to advances in technology, there has been a concomitant growth in expectations of sophistication and functionality provided by embedded systems. For example, over the years, the mobile phone has evolved from being simply a phone to a versatile mobile computing device, which delivers image capturing, text messaging, gaming, e-mail and more.

As the complexity of embedded systems grows, there is an increasing use of operating systems in embedded devices, such as mobile phones, media players and other consumer electronics. A survey in 2006 shows that over 71% of embedded system developers used some form of operating systems in their projects [58]. In recent years, there is also an increasing trend to apply general-purpose operating systems, such as Linux, in embedded devices [4, 1, 2]. For example, from 2005 to 2006, there was nearly a 30% growth in commercial revenue related to operating system and tools sales for Linux-based embedded systems, and the market for embedded software solutions used in the development of Linux-based devices was well above $100 million in 2006 [5].

Applying existing general-purpose operating systems to embedded systems is in many ways a simpler and more economical approach than developing and maintaining in-house operating systems. Embedded systems, however, are special-purpose computer systems designed to perform a few dedicated functions. General-purpose operating systems—precisely because they are general-purpose—contain features and code that are not needed in every application context, which incurs unnecessary overheads, e.g., in execution speed or memory footprint. This, together with the memory constraint, makes it important to
reduce the memory footprint of operating systems for embedded systems.

1.1 Reducing Memory Footprint of Application Programs

One way to reduce the memory footprint of programs is to apply efficient memory management techniques, such as overlays and virtual memory. These techniques use secondary storage (usually hard disk space) to extend the physical memory. They allow a program to store only part of its memory data in the physical memory so that, it can still run, even if the program’s memory data is too large for the physical memory. However, such techniques either require additional efforts from the programmers (e.g., overlays require programmers to divide programs into pieces and ensure the proper overlay is loaded at the proper time.), which erodes programmer productivity; or they need special features from the hardware and operating systems (e.g., virtual memory), which is not practical for all embedded systems. Furthermore, operating systems, such as Linux, usually do not apply such techniques to the kernel memory due to the high latency involved when reading data from secondary storage.

Another way to reduce the memory footprint of programs is simply to make the programs smaller so that less memory is required. Two kinds of techniques have been considered to reduce program size:

1. compression and compaction. A large body of research has been focused on these techniques and a comprehensive survey of them was presented by Beszédes et al. [8].

**Compression** In information theory, compression (or data compression) is the process of encoding information using fewer bits than the original representation. When applied to software programs, it means compressing the program’s code and/or data so that it can be stored using less space. An easy way to compress a program is to directly use general compression tools, such as gzip. However, general compression techniques, such as the DEFLATE algorithm [20] used by gzip, typically search only for common patterns on a byte or word boundary and therefore can miss semantic common patterns such as “a

---

1Here by program size, we mean the size of both program code and data.
call instruction if often followed by a move instruction”. Ernst et al. [23] proposed approaches that exploit such semantic common patterns and achieve better compression results than using gzip.

The problem of compression-based approaches is that the compressed program generally needs to be decompressed first before the program can be executed. Although decompression time may be negligible and, in fact, may be compensated for by the savings in transmission or retrieval time [28], a more severe problem is the space required to store the decompressed code, which can be unfavorable for memory-constrained system.

However, the decompression overhead can be mitigated by compressing a program partially. Debray and Evans [18] described a method that only compresses the infrequently executed code of a program and divides compressed code into pieces so that each piece of compressed code is decompressed only when it is needed.

**Compaction** Compaction is a different approach that transforms a program into a more compact representation that is still directly executable. Compaction avoids the problem of requiring decompression. However, in order to keep the compact representation executable, compaction-based approaches are limited in ways that they can transform a program.

Many standard compiler optimizations, such as dead-code elimination, unreachable code elimination, and redundant-code elimination, have the effect of reducing a program’s code size [6]. Debray et al. [19] presented a tool called SQUEEZE that combines the classic compiler optimizations with additional compaction techniques such as code factoring and procedural abstraction. Together with aggressive inter-procedural analysis, it achieved a high ratio of code size reduction. In addition to reducing the program’s code size, De Sutter et al. [17] described compaction methods that can also reduce the program’s global data size.
1.2 Reducing Memory Footprint of OS kernels

Most previous work on memory footprint reduction via compression and compaction has focused only on application programs. Operating system (OS) kernels, however, are different from ordinary application programs in many ways. The differences impose challenges of program analysis that are usually not encountered in ordinary application programs. The main challenges of program analysis of OS kernels are:

1. OS kernels, such as the Linux kernel, typically contain a significant amount of hand-written assembly code, which does not follow the familiar conventions of compiler-generated code. The program analysis of OS kernels has to be able to handle the heterogeneity in the kernel code. We quantify this observation later in Section 2.1.

2. OS kernels often make extensive use of indirect function calls in order to enhance maintainability and extensibility. This complicates the program analysis because it can make constructing a precise control flow graph difficult. Identifying possible targets of indirect function calls is equivalent to pointer analysis, which is a hard problem both theoretically and in practice. The problem of indirect function calls in OS kernels is exacerbated further by the fact that the hand-written assembly code in the OS kernels may also contain indirect function calls, and identifying those targets requires pointer analysis of assembly code as well. This observation is also quantified in Section 2.1.

3. Unlike ordinary application programs, whose entry point is well-defined, OS kernels have multiple implicit entry points in the form of system calls and interrupt handlers. The program analysis has to take this into account in order to guarantee soundness.

4. Other peculiarities in OS kernels, such as non-contiguous code layout of functions and exception handling, also complicate program analysis.
To address the above challenges, we developed a framework that combines the high-level source code analysis and low-level binary rewriting technique to analyze and optimize OS kernels. At the source-level, pointer analysis is used to resolve the indirect function call targets in OS kernels. At binary-level, a binary rewriting technique is applied to process OS kernel binaries, which provides a uniform way to deal with code heterogeneity due to a combination of source code, assembly code, and legacy code such as in device drivers. Chapter 2 gives detail discussions about our framework for analyzing and optimizing OS kernels.

1.2.1 Opportunities for Reducing Memory Footprint of OS kernels

Despite the challenges arising in analyzing and optimizing OS kernels, there are also opportunities to reduce the memory footprint of OS kernels, especially for embedded systems. The opportunities are:

1. Embedded systems tend to have fixed configurations. At the hardware end, they are limited in the set of devices with which they interact (e.g., a cell phone or digital camera will typically not have a mouse interface); at the software end, they usually support a fixed set of applications (e.g., we do not routinely download or build new applications on a cell phone or digital camera). This implies that an embedded system will typically use only some of the functionality offered by a general-purpose operating system. Therefore, the code corresponding to the unused functionalities can be removed.

2. Dynamically, most OS kernel code is executed infrequently in an context of embedded environment. When running with MiBench [31] on a minimally configured Linux kernel, for example, we found that only about 32% of the kernel code was actually executed. This implies that the memory footprint of the OS kernel code can

---

2Here we refer to many existing cellphones that come with a fixed set of applications. Recently, there are more advanced devices, such as the iPhone, that support installing new applications by users and can potentially run a large set of applications. However, a user of such device typically still requires only a few applications in many cases.
be reduced if we can keep unexecuted or infrequently executed code out of main memory and load it only if it is needed.

3. Apart from reducing the memory footprint of the OS kernel code, there are also opportunities for reducing the data size of an OS kernel through compression. For example, integer-valued variables often do not require the full 32 bits allocated to them (on a conventional 32-bit architecture); other opportunities for memory usage reduction arise from redundancy in sets of pointer values whose high-order bits typically share a common prefix. However, such opportunities for data compression are usually not obvious statically, making it difficult to optimize programs to take advantage of them.

1.3 Major Contributions

This dissertation addresses the problem of reducing memory footprint of operating system kernels. It makes the following major contributions:

1. **Code compaction of OS kernel** [34]: an automatic approach for reducing the code size of an OS kernel by combining high-level source code analysis and low-level binary rewriting technique. A novel “approximate decompilation” technique is applied to apply source code analysis to hand-written assembly code. This improves the precision of our analysis considerably, and in a simple way, while ensuring that the safety of our transformation is not compromised. A prototype implementation of our ideas on an Intel x86 platform, applied to the Linux kernel configured minimally so as to exclude all unnecessary code, is able to achieve a code size reduction of nearly 24%, on average, on the MiBench suite of embedded system applications [31].

2. **On-demand code loading of OS kernel** [32]: an automatic technique for reducing the memory footprint of an OS kernel by keeping infrequently executed code on secondary storage and loading such code only if it is needed at runtime. The technique is software-based and does not require hardware or operating system support.
for virtual memory. A prototype of the technique has been implemented for the Linux kernel. We evaluate this approach with two benchmark suites, MiBench and MediaBench, and a Web server application. The experimental results show that this approach reduces memory requirements for the Linux kernel code by about 53% with little degradation in performance.

3. **Compressing dynamic data structures in OS kernel** [33]: an approach that reduces the dynamic memory footprint of an OS kernel by compressing dynamic data structures. A prototype implementation for the Linux kernel reduces the memory consumption of the slab allocators in Linux by about 17.5% when running the MediaBench suite, while incurring only minimal increases in execution time (1.9%).

4. Finally, this dissertation also discusses issues that arise in analyzing and optimizing OS kernels and discusses how they can be handled.

In the current implementation and experiments we focus on the Linux kernel. However, the techniques described in this dissertation are general and can be applied to other operating system kernels as well.

1.4 Outlines

The rest of this dissertation is organized as follows. Chapter 2 gives background information on the program analysis used for reducing the memory footprint of an OS kernel. It provides a brief overview of binary rewriting techniques for OS kernels and our implementation of a pointer analysis, FA analysis, for resolving targets of indirect function calls in OS kernels. Chapter 3 discusses how to reduce the code size of an OS kernel

---

3The experimental results of the three parts of this work—static code compaction, on-demand code loading, and dynamic data structure compression) are presented separately in this dissertation. The results are used to demonstrate the effectiveness of each approach. The cumulative results, however, are not presented due to the limitations of our current implementation.

4A lot of work on binary rewriting of OS kernel, especially the disassemble part, is done by Mohan Rajagopalan and Somu Perinayagam [54].

5The implementation of FA analysis used in this thesis is done by John Trimble as part of his honors thesis [57].
by combining source-level pointer analysis and low-level binary rewriting. Chapter 4 describes how the memory footprint of OS kernel code can be reduced by using dynamic code loading. Chapter 5 explains how the dynamic memory footprint of an OS kernel can be reduced by applying dynamic data structure compression. Related work is discussed in chapter 6. Conclusions and directions for future work are given in chapter 7.
CHAPTER 2

BACKGROUND

This chapter provides background information of the program analysis for memory footprint reduction of an OS kernel. It begins with discussions about the program characteristics differences between OS kernels and application programs. Second, it discusses the issues that arise in the context of binary rewriting and instrumentation of an OS kernel and how they can be handled. A lot of work described here was done by Mohan Rajagopalan and Somu Perinayagam [54]. In particular, the material described in section 2.3.1, 2.3.2 and the material described in section 2.4 except for 2.4.1 were done by Mohan Rajagopalan and Somu Perinayagam. Last, section 2.5 gives a brief introduction of FA analysis, which is the pointer analysis used to resolve the indirect function call targets in OS kernels.

2.1 The Program Characteristics of OS Kernels

This section discusses the program characteristics of OS kernel comparing with embedded application programs. We quantify our observations using a minimally-configured Linux 2.4.31 kernel. The kernel was compiled on an Intel x86 platform using GCC compiler version 3.4.4 with optimization flags “-Os -fomit-frame pointer” for optimizing the code size. The size of the kernel executable image is about 1,152KB.

Our first observation is that the kernel code takes a significant portion of the total static memory, which includes code and global data, in the OS kernel. Table 2.1 shows the size of all code sections (containing the kernel code) and the size of all data sections (containing the global data) of the Linux kernel. The total data size of the Linux kernel

\[ \text{The kernel is configured with the following options: without module-support; ext2 file systems; with TCP/IP and UDP stack and with drivers for network, video, block devices, keyboard only. Here and elsewhere in this chapter, we use the phase “the Linux kernel” to refer to this version of the Linux kernel.} \]
<table>
<thead>
<tr>
<th>Code section name</th>
<th>Size(Bytes)</th>
<th>Data Section Name</th>
<th>Size(Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>735,139</td>
<td>.rodata</td>
<td>73,472</td>
</tr>
<tr>
<td>.text.init</td>
<td>75,638</td>
<td>.data</td>
<td>43,524</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.data.init</td>
<td>12,845</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.bss</td>
<td>164,976</td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td><strong>810,777</strong></td>
<td><strong>Data size</strong></td>
<td><strong>294,817</strong></td>
</tr>
</tbody>
</table>

Table 2.1: Code and data section size of minimally-configured Linux kernel 2.4.31

<table>
<thead>
<tr>
<th>Benchmark set</th>
<th>Programs</th>
<th>Avg. code size (bytes)</th>
<th>Avg. indirect calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto./Industrial</td>
<td>basicmath, bitcount, qsort, susan</td>
<td>20,440</td>
<td>0</td>
</tr>
<tr>
<td>Consumer</td>
<td>jpeg, mad, lame, tiff2bw, tiff2rgb, tiffdither, tiffmedian, typeset</td>
<td>127,961</td>
<td>156</td>
</tr>
<tr>
<td>Network</td>
<td>dijkstra, patricia (blowfish, CRC32, sha)</td>
<td>21,58</td>
<td>0</td>
</tr>
<tr>
<td>Office</td>
<td>ghostscript, ispell, rsynth, stringsearch</td>
<td>181,314</td>
<td>454</td>
</tr>
<tr>
<td>Security</td>
<td>blowfish, ppg, rijndael, sha</td>
<td>43,212</td>
<td>4</td>
</tr>
<tr>
<td>Telecomm</td>
<td>adpcm, CRC32, FFT, gsm</td>
<td>6,185</td>
<td>2</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td><strong>63,545</strong></td>
<td><strong>103</strong></td>
</tr>
<tr>
<td>Linux</td>
<td></td>
<td><strong>810,777</strong></td>
<td><strong>1,368</strong></td>
</tr>
</tbody>
</table>

Table 2.2: Code size and number of indirect function calls of MiBench suite.

is about 294KB while the total code size of the kernel is about 810KB, which is over 2.7 times larger than the data size of the kernel. In another word, about 73% of total static memory of the Linux kernel is code.

Our second observation is that the code of an OS kernel also takes a significant portion of the total code in the whole embedded system. To show that, we use the MiBench suite [31], a widely used collection of benchmark programs for embedded systems, to compare the code size between an OS kernel and embedded application programs.

The MiBench suite is organized into six sets of benchmarks, corresponding to different kinds of embedded environments: Automotive and industrial control, Consumer devices, Networking, Office automation, Security, and Telecommunications; each of these sets contains several different application programs, which are listed in the second column in Table 2.2. We compiled the programs in the MiBench suite using GCC version 3.4.4 with dynamic-link library (i.e., excluding library code from the executable). Table
2.2 shows the average code size of these programs. Overall, the average code size of these programs is about 63KB, where the code size of our minimally-configured kernel is over 12x larger than this average number.

Third, as an indication of the extent and effects of hand-written assembly code in the Linux kernel, we found that of the 5,133 functions in the kernel, 89 functions did not have the standard function prologue and 34 did not have the standard epilogue. It suggests that the code was not compiler-generated, i.e., was hand-written assembler. By contrast, in the application programs we examined, all functions had standard prologues and epilogues.

Last, OS kernels, such as the Linux kernel, make intensive use of indirect function calls. There are 1,368 indirect function calls in the Linux kernel that we compiled. The fourth column in Table 2.2 shows the average number of indirect function calls in the programs in MiBench suite. The number indicates that, with the notable exception of the Consumer and Office benchmark set, most of the programs in MiBench contain relatively few indirect function calls. The average number of indirect function calls of all benchmark sets is 103, which is much smaller than the number of indirect function calls in the Linux kernel.

Our observations show that, OS kernel code consumes an important portion of the total memory footprint of an OS kernel as well as the whole embedded system. OS kernel code, however, contains a significant amount of hand-written assembly code and indirect function calls, both of which are usually not encountered in ordinary application programs.

2.2 Binary Rewriting of OS Kernels

Binary rewriting is an attractive approach for processing OS kernel code. It provides a uniform way to handle heterogeneity in the kernel code due to a combination of source code, assembly code and legacy code such as in device drivers. However, because of the many differences between ordinary application code and OS kernel code, binary rewriting techniques that work for application code do not always carry over directly to kernel code. This section describes some of the issues that arise in this context, and the approaches
we have taken to address them. A key goal when developing our system was to deal in a systematic manner with the various peculiarities seen in low-level systems code, and reason about the safety and correctness of code transformations, without requiring significant deviations from the regular developmental path. For example, a precondition we assumed was that no compiler or linker modifications should be required to use it and the tool should process kernel binaries in the same way it does ordinary applications.

A prototype kernel binary rewriter is implemented as an extension to the PLTO binary rewriting toolkit [56]. PLTO takes as input a relocatable binary that it manipulates in various ways, e.g., to insert instrumentation code or to apply various optimizing transformations using optional execution profiles for guidance. It then updates code addresses as necessary, using relocation information to distinguish addresses from non-address values, and finally writes the resulting program out as an executable. PLTO currently supports the collection of several different kinds of execution profiles: basic block counts, edge counts, value profiles (especially important for resolving indirect function call targets), call-stack profiles, as well as profiles based on hardware performance counters, e.g., CPU cycles and i-cache misses.

2.3 Technical Challenges of Binary Rewriting of OS kernels

This section discusses some of the issues arising in binary rewriting and instrumentation of OS kernels that are usually not encountered for ordinary application programs.

2.3.1 Disassembly

The different characteristics of kernel and application binaries means that a straightforward application of conventional disassembly algorithms can fail to correctly disassemble large parts of the kernel. A significant problem during disassembly is that of data embedded in the text section, which can confuse the disassembly process. In the Linux kernel, binary data is embedded in the instruction stream in two distinct instances:

1. Data areas that are not part of instruction stream but are located in the text section. For example, in Linux version 2.4, the page tables are placed in the text section.
We identify such data areas by their associated symbols. A list of such symbols is provided as input to PLTO, which then skips over the corresponding memory areas during disassembly.

2. Data embedded in the instruction stream that are not part of any instruction, but which may be used during execution. A typical example of this is the ud2 instruction used in the kernel. The ud2 instruction, which specifies an “undefined instruction,” raises an invalid opcode exception and is used to raise a panic and halt the kernel in case of a bug. Typically, the source code line number and a pointer to the file name are stored in the six bytes following each ud2 instruction. The ud2 handler prints out this information before halting the kernel. Such usage is very kernel-specific: the references to the data bytes following the ud2 instruction are not obvious in the code containing the instruction, but instead occur (indirectly, through the address from which the exception was raised) in the ud2 exception handler. About 6% of the functions in the Linux kernel contain these instructions. A straightforward disassembly of the kernel would very likely treat the data bytes following the ud2 instructions as unreachable; however, eliminating them could potentially change the behavior of the kernel.

A crude user-level analog of this is with jump tables embedded in the text section in position-independent code. A key difference between the two situations is that references to such jump tables from within the code are relatively direct and not very difficult to identify, while references to the ud2 instructions are indirect and significantly harder to identify without specific high-level knowledge of how they are used.

Overall, we found about 21 Kbytes of data embedded within the code stream in the code sections in the Linux kernel, out of a total of 1.16 Mbytes, i.e., about 1.8%.

To address this problem, we use symbol table information to guide the disassembly, which proceeds in three phases: First, symbol information is used to identify well defined code regions such as functions, which are disassembled using the standard recursive disassembly algorithm. The second phase uses the symbol table to try and identify “stubs,”
which are code regions that do not appear to be conventional functions. Typical examples of such code are hand-written assembly routines, kernel entry point routines, interrupt handlers, etc. The final phase of disassembly uses relocation information to discover regions of code that have been missed by the previous steps. The basic idea is to exploit relocation information that is available in the binary. In this phase, all the relocation entries are checked to see if they point to a disassemble-able region of code. This is done by checking if the source address for the relocation is within the text section. If the source address is within the text section then this is treated as a potential jump target and becomes a target for recursive disassembly. This step is effective in identifying almost all the regions that were missed out in the earlier phases if they were reachable only as targets of indirect control transfers. Our results indicate that this algorithm 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. Portions of the text section that cannot be disassembled are treated as data and are reinserted into the kernel executable when they are reassembled; however, any code pointers in such undisassembled code/data are identified as such, and updated correctly, using the associated relocation information. For example, jump tables in the code section are handled in this way.

2.3.2 Control Flow Analysis

After disassembly, the resulting instruction sequence is organized into a inter-procedural control flow graph. Unfortunately, control flow analysis of operating system kernels is complicated by a number of factors, such as the presence of hand-written assembly code and its interaction with indirect function calls; code layout to segregate infrequently ex-

---

2 In the Linux kernel, stubs appear as text section symbols of type \texttt{NOTYPE}. However, not all \texttt{NOTYPE} symbols in the text section correspond to stubs: there are a handful of such symbols that should not be disassembled as code, because they point either to data or to special regions in the text section. Since it is not possible to algorithmically identify such regions, we allow the user to specify such embedded data symbols via a table that indicates the name, location, and the size of the symbol. In our current implementation, this table contains fewer than 20 entries and includes the special purpose pages that are used to initialize the memory manager.
executed code in order to avoid cache pollution; and exception handling. This section discusses some of these issues.

**Hand-written Assembly Code**

As it is shown previously in Section 2.1, OS kernels contain a significant amount of hand-written assembly code. Hand-written assembly code can be a problem for program analysis. For example, functions from hand-written assembly code may not have standard prologues and epilogue. The absence of standard prologues and/or epilogues can affect the precision of analyses that examine the stack behavior of functions. For instance, it can affect stack analysis that is aimed at determining the height of the stack at different program points (This information is used to support a variety of other analyses, such as constant propagation and stack location liveness).

**Indirect Function Calls**

OS kernels often make extensive use of indirect function calls in order to enhance maintainability and extensibility. In order to precisely perform control flow analysis, such as the construction of program call graph, it is important to resolve the indirect function call targets. However, static analyses are generally quite conservative in their treatment of indirect function calls. The problem is complicated by the fact that hand-written assembly code in an OS kernel may itself contain indirect function calls, and identifying those targets requires pointer analysis of the assembly code.

**Implicit Entry Points**

An important problem in dealing with control flow in operating system kernels is that not all entry points into the kernel, and control flow within the kernel, are explicit. There are implicit entry points such as system calls and interrupt handlers, as well as implicit control flow arising from interrupts, that have to be taken into account in order to guarantee soundness. Our implementation uses the system call table to identify system call handlers and mark them as potential entry points into the kernel control flow graph (interrupt han-
static inline void __down_read(struct rw_semaphore *sem)
{
  __asm__ __volatile__{
    "# beginning down_read
    LOCK_PREFIX incl (%%eax) /* adds 0x00000001, returns the old value */
    js 2f
    /* jump if we weren’t granted the lock */
  1:
    LOCK_SECTION_START("") /* code placed in a separate subsection */
    pushl %%ecx
    pushl %%edx
    call rwsem_down_read_failed
    popl %%edx
    popl %%ecx
    jmp 1b
  }
  LOCK_SECTION_END
  "# ending down_read"
  "=m"(sem->count)
  "a"(sem), "m"(sem->count)
  "memory", "cc");
}

Figure 2.1: Example of a function whose code is spread over multiple subsections

dlers do not need to be treated specially, but are found in the course of ordinary control flow reachable analysis).

Non-contiguous Code Layout of Functions

A kernel developer can exploit the use of hand-coded assembly to lay out specific parts of the same kernel function in different parts of memory, so as to separate common execution paths from less frequently taken execution paths. The infrequently executed path is placed in a separate subsection within the text section. There is one subsection created for all the functions belonging to the same module, and all their infrequently executed code is placed in that subsection. This is illustrated in Figure 2.1, where the code fragment between LOCK_SECTION_START and LOCK_SECTION_END are placed in a different subsection within the text section. This has the effect of realizing the “procedure splitting” optimization described by Pettis and Hansen [51]. However, this can lead to imprecision during control flow analysis: since the different subsections have symbols associated with them, a naive disassembler may infer that the code in these subsections belong to distinct functions. The resulting code appears to have two distinct functions, with control jumping from the middle of one into the middle of the other. This kind of
inter-procedural control flow—which actually occurs in some parts of the kernel code—is not easily handled in many of our program analyses, and can lead to a loss in precision. We address this problem by making a post-pass over the control flow graph to identify functions that have been split in this manner, and merge the code from the two distinct functions into a single function. We found 100 subsections in the Linux kernel, with each subsection having, on average, code for about 9 functions.

**Control Flow Issues in Exception Handling**

In order to identify all reachable code in the kernel, it is not enough to consider ordinary control transfers, which are explicit in the code: we also have to take into account control transfers that are implicit in the exception handling mechanisms of the kernel. For this, we examine the exception table in the kernel. Locations in the kernel where an exception could be generated are known when the kernel is built. For example, the kernel code that

---

**Key:**

1. A memory exception at L₁ causes control to branch to the exception handler.
2. Exception handling code.
3. Exception handler searches __ex_table with the address L₁, where the exception occurred, to find the associated fixup code address L₂.
4. Control branches from the exception handler to the fixup code.
5. Control branches from the fixup code to some appropriate code address.

Figure 2.2: Control flow during the handling of exceptions in the Linux kernel
copies data to/from user space is known as a potential source for a page fault exception. The Linux kernel contains an exception table, **__ex_table**, that specifies, for each such location, the code that is to be executed after handling an exception. Additionally, a special section, **.fixup**, contains snippets of code that carry out the actual control transfer from the exception handlers to the appropriate destination locations. The flow of control when handling an exception is shown in Figure 2.2: after the exception handler deals with an exception from an address $L_1$, it searches **__ex_table** with $L_1$ as the key, finds the associated address $L_2$ of the corresponding fixup code, and jumps to $L_2$. This then carries out some fixup actions, e.g., setting flags or error values, and eventually jumps to some appropriate location in the text area. For example, when a page fault occurs, the reason could be either that the address being referenced lies in a page that is not in memory, or that it is an illegal address. In the former case, the exception handler loads the referenced page into memory and the fixup code branches back to the instruction that raised the exception, causing it to be re-executed. In the latter case, the fixup code branches to an error routine.

The key point to note here is that the control flow path from $L_1$ to $L_2$ is not explicit in the code, but is implicit in **__ex_table**. It is necessary to take such implicit execution paths into account for code compaction as discussed in Chapter 3 to ensure that we find all reachable code. We do this by examining the exception table and adding pseudo-control-flow edges to indicate such implicit control flow. For the example in Figure 2.2, we would add such an edge from $L_1$ to $L_2$. One implication of this is that any instruction that can raise an exception, i.e., which is referenced from the exception table, terminates a basic block.

Of the 108,611 control flow edges in the whole-program control flow graph of the Linux kernel, 698 edges were pseudo-control-flow edges resulting from the exception-handling mechanism described above.
2.4 Code Transformations of OS Kernels

Idiosyncrasies of the kernel also affect the way we apply transformations to the code. There are two main considerations here. The first involves code that cannot be altered or moved because its behavior is closely tied to interactions with the underlying hardware, while the second involves interactions with exception handling. These are illustrated here with some examples.

The first example is of boot up code where apparently unnecessary instructions cannot be eliminated. In the code snippet shown below, the first number on each line is the address of the instruction on that line:

```
<startup_32>
...
0xc0100036 mov %eax, %cr0
0xc0100039 jmp 0xc010003b
0xc010003b mov $0xc0100042, %eax
0xc0100040 jmp *%eax
0xc0100042 lss 0xc01001e5, %esp
...
```

This code snippet contains two `jmp` instructions, shown in bold, each of which jumps to the following instruction: the first of these jumps to the next instruction, whose address (0xc010003b) is specified as an absolute operand, while the second loads the address of the instruction after it (0xc0100042) into register `%eax` and then jumps indirectly through this register. Each of these `jmp` instruction therefore appears redundant. It turns out, however, that these instructions check whether turning on paging in the hardware worked, and cannot be optimized away. Furthermore, the page tables are located immediately after the hardware initialization. These tables need to be page-aligned, and any transformation to the initial boot up code could potentially violate this alignment requirement. Violations of such alignment requirements cause the kernel to hang during boot up time.

The exception-handling mechanism discussed in Section 2.3.2 (see also Figure 2.2) also imposes implicit constraints on code transformations. The most obvious of these is that any transformation that involves code duplication—for example, function inlining—
must ensure that additional exception table entries and fixup code are added for each instruction in the duplicated region that can give rise to an exception (to get around this issue, our implementation currently carries out inlining of functions only if the function being inlined does not contain any instructions that can cause an exception, i.e., does not have any entries in the exception table pointing into its body).

Exception-causing code can have other effects as well. Consider the situation illustrated in Figure 2.3. The `pop` instruction in basic block B1 can raise a page-fault exception. This causes control to branch to an exception handler which as discussed in Section 2.3.2, loads the referenced page into memory and then jumps to a block of fixup code; in this case, the fixup code then transfers control back to the original instruction that raised the exception, and re-executes it. The problem here is that when we consider the exit from basic block B1, we cannot guarantee that the `pop` instruction in that block has been executed. One possible solution would be to propagate some of the instruction semantics to the control flow edges. For example, a stack analysis aimed at determining the height of the stack at different program points would have to conclude that the `pop` instruction, which deallocates a word off the stack, has been executed if the fall-through edge out of block B1 is taken, but is not executed if the exception edge B1 → B2 is taken. While this would give correct results, such an approach is a departure from the standard treatment of control flow graphs, and has the effect of complicating the various data-flow analyses used. Our current implementation makes the simpler (but conservative) assumption that

Figure 2.3: An example of analysis complications due to exception edges

![Diagram showing exception and control flow edges](image-url)
in situations where a basic block has an outgoing EXCEPTION edge, we cannot guarantee whether or not the last instruction in the block has been executed. One of the side effects is that the height of the stack after the pop instruction cannot be determined. This affects the precision of analyses that depend on such information.

2.4.1 Instrumentation

Our system supports profiling of the kernel based on both software-managed counters (e.g., basic block and edge profiles) and hardware-managed counters (e.g., CPU cycles, cache misses). In order to obtain execution profiles, we need to know where to begin profiling as well as where to end profiling and write out the profile data. For ordinary applications, the well-defined entry and exit points serve as natural points for starting and ending profiling respectively. An OS kernel, however, has multiple entry and exit points, making it necessary to create a mechanism to begin and end profiling.

Our system uses a special (new) system call for this. One of its arguments determines whether it starts profiling or ends it and writes out the results. Another argument determines what kind of profiling is carried out (basic block counts, edge counts, or hardware-counter profiles). The code to be profiled is bracketed with calls to this system call (currently these calls are inserted manually, but in principle this step is easily automated).

While this infrastructure has been used to instrument the kernel to track different kinds of control and data flows, it is unable to instrument the initialization code that sets up interrupt and fault handlers at boot time. This is because the profiling data structures can not be accessed until page tables have been initialized. However, in practice this region of code is small and this does not lead to significant loss of information for later optimizations.

2.5 FA Pointer Analysis

There is a large volume of literature on pointer alias analysis, with a variety of assumptions, goals, and trade-offs (see, for example, the discussions by Hind and Pioli [35]). In general, these analyses exhibit a trade-off between efficiency and precision: the greater
the precision, the greater the analysis cost, i.e., the lower the efficiency. FA analysis is a flow-insensitive, context-insensitive, and type-sensitive alias analysis, originally due to Zhang et al. [64, 65], that is at the efficiency end of this trade-off. This section gives a brief introduction to FA analysis.

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

Milanova et al. have observed, however, that despite its low precision for general-purpose pointer alias analysis, FA analysis turns out to be quite precise in practice for identifying the targets of indirect function calls [47]. The authors attribute this to the fact that programmers typically use function pointers in a few specific and relatively simple stylistic ways.

The FA analysis categorizes memory references into sets such that all the members in a given set may-alias one another. This has the side effect that if $A$ may point to $B$ then $B$ may point to $A$. In other words, FA analysis is symmetric.

The FA analysis does not take into account pointer arithmetic or array indexing. So a statement such as $* (p + 1)$ or $p[1]$ is simply treated as $*p$. The analysis does, however, distinguish between different fields of a structure, so $p.x$ is distinct from $p.y$.

The FA analysis constructs for each memory reference an object name. An object name is simply a variable name with a series of right-associative pointer dereferences ($*$) and address operators ($&$) as well as left-associative field accesses ($.$field). This construction mimics the use of memory references in C. For example, the reference in C, $& (t->y)$ would map to the object name $& (* (t).y)$. Since pointer arithmetic is ignored by FA analysis the reference $* (p+1)$ maps to the object name $*p$. 
The PE equivalence relation (Pointer-related Equality) is used to partition the set of all object names into equivalence classes. This relation is represented by a graph $GP_E$. Each vertex of the graph corresponds to an equivalence class of object names. These vertexes are connected via edges labeled as pointer-dereference or by a field name.

Here only a rough sketch of the construction of $GP_E$ is provided. The basic outline is to initially place every object name in its own equivalence class. Edges connect nodes as follows:

- If an equivalence class $e_1$ contains an object name $o.\ field$ where $o$ is an object name in equivalence class $e_2$, then there is an edge labeled $\ field$ from the vertex $v$ representing $e_2$ to the vertex $w$ representing $e_1$ in $GP_E$.

- If an equivalence class $e_1$ contains an object name $*o$ where $o$ is an object name in equivalence class $e_2$, then there is an edge labeled $*$ from the vertex $v$ representing $e_2$ to the vertex $w$ representing $e_1$ in $GP_E$.

- If an equivalence class $e_1$ contains an object name $\&o$ where $o$ is an object name in equivalence class $e_2$, then there is an edge labeled $*$ from the vertex $v$ representing $e_1$ to the vertex $w$ representing $e_2$ in $GP_E$.

For every assignment in a given program, the equivalence class of the object name representing the left hand side of the assignment is merged with the equivalence class of the object name of the right hand side. This involves identifying the vertexes representing these equivalence classes. If a merge results in a vertex $v$ being the source of two edges with the same label, then the equivalence classes represented by the destination of these two edges are also merged.

Direct calls are handled similarly to assignments. The object names for the formal parameters of function are merged with object names for the parameters of the function call. Indirect calls are handled somewhat differently. Suppose an indirect call is made using function pointer $p$. Whenever the object name of a function is merged with $p$ then the object names for the call sites parameters are merged with the formal parameters of the function.
After the $GP_E$ is constructed, determining the potential targets of an indirect call is straightforward. If an indirect call uses a function pointer $f$ then the possible targets are all the function object names in the same equivalence class as $f$. 
CHAPTER 3

CODE COMPACTION OF OS KERNELS

Embedded systems tend to have relatively static configurations: at the hardware end, they are limited in the set of devices with which they interact; at the software end, they usually support a fixed set of applications. This implies that an embedded system will typically use only some of the functionality offered by a general-purpose operating system. The code corresponding to the unused functionality is unnecessary overhead, and should be removed. Some of these overheads can be removed simply by configuring the kernel carefully so as to exclude as much unnecessary code as possible. However, not all overheads can be removed in this manner. For example, a given set of applications running on an embedded platform will typically use only a subset of the system calls supported by the operating system; the code for the unused system calls is then potentially unnecessary. Such unnecessary code typically cannot be eliminated simply by tweaking the configuration files; additional analysis is required. This section discusses how such analysis may be carried out in order to identify code that can be guaranteed to be unnecessary.

3.1 Challenges of Code Compaction of OS Kernel

An especially important issue for code compaction is that of control flow analysis, both intra-procedural and inter-procedural. This is because, in practice, most of the code size reduction arising from compaction comes from the detection and elimination of dead and unreachable code [19]. For soundness reasons, we have to ensure that we only eliminate code that can be guaranteed never to be needed during any future execution. This means that imprecision in control flow analysis directly affects the amount of code that can be eliminated.

Unfortunately, control flow analysis in operating system kernels is complicated by the interaction of two separate problems. First, there are significant amounts of hand-
written assembly code. Second, operating system kernels often make extensive use of indirect function calls in order to enhance maintainability and extensibility. This is a problem because static analyses are generally quite conservative in their treatment of indirect function calls. Each of these problems—hand-written assembly and indirect function calls—is nontrivial in its own right, and the situation is exacerbated further by the fact that they interact: the hand-written assembly code in an operating system kernel may itself contain indirect function calls, and identifying those targets requires pointer alias analysis of the assembly code.

A final problem in dealing with control flow in operating system kernels is that not all entry points into the kernel, and control flow within the kernel, are explicit. There are implicit entry points such as system calls and interrupt handlers as well as implicit control flow arising from interrupts; both have to be taken into account in order to guarantee soundness.

### 3.2 Pointer Analysis: Resolving Indirect Function Call Targets

Static analysis of OS kernel code is complicated by the presence of hand-written assembly code. On the one hand, dealing with hand-written assembly code at a source-level or intermediate-code-level analysis is messy and awkward because of the need to inject architecture-specific knowledge into the analysis—such as aliasing between registers (e.g., in the Intel x86 architecture, the register `%al` is an alias for the low byte of the

---

1In general, identifying the possible targets of indirect function calls is equivalent to pointer alias analysis, which is a hard problem both theoretically and in practice.
register %eax) and idiosyncrasies of various machine instructions. On the other hand, if the analysis is implemented at the assembly code or machine code level, much of the semantic information presenting at the source level is lost—in particular, information about types and pointer aliasing—resulting in an overly conservative analysis that loses a great deal of precision. Nor can such assembly code be ignored, since soundness demands that all possible execution behaviors of the program be taken into account.

One possible solution to this problem would be to decompile the hand-written assembly code back to equivalent C source code that could then be analyzed by source-level analysis. The problem with such an approach is that it is not obvious that all of the kernel assembly code can be reverse engineered back to equivalent C source code. For example, “system instructions” on the Intel x86 architecture, such as “load interrupt descriptor table register” and “invalidate TLB entry,” do not have obvious C-level counterparts. Moreover, even in situations where reverse engineering is possible, it can be complicated and involve a great deal of engineering effort. Instead, we deal with this problem using an approach we call “approximate decompilation,” which automatically maps hand-written assembly code back to C source files for analysis purposes. The idea, illustrated in Figure 3.1, is that given an assembly file foo.s and a program analysis \( A \), we create a source file foo\(^A\).c that has the property that an \( A \)-analysis of foo\(^A\).c is a safe approximation of the behavior of foo.s, even though foo\(^A\).c is not semantically equivalent to foo.s. For example, if \( A \) focuses on control flow analysis, then foo\(^A\).c may elide those parts of foo.s that are irrelevant to control flow. We have applied this approach to use a source-level pointer alias analysis technique called FA-analysis\(^2\) to identify the possible targets of indirect function calls.

3.2.1 Approximate Decompilation of Kernel Code for FA Analysis

As Figure 3.1 suggests, the way in which approximate decompilation is carried out depends in part on the source-level analysis that is applied to the resulting source files. This section discusses approximate decompilation of assembly code in the Linux kernel code.

\(^2\) A brief description of FA analysis is provided previously in Section 2.5.
for FA analysis. For concreteness, we discuss kernel assembly code on the Intel x86 architecture.

The hand-written assembly instructions in the Linux kernel falls into two broad groups: (1) general-purpose instructions that perform basic data movement, arithmetic, logic, and program control flow operations, and (2) system instructions that provide support for operating systems and executives [36]. We process these instructions as follows:

- System instructions (the second group above) manipulate only the hardware (or data related to the hardware) and have no effect on pointer aliasing in the kernel code. For pointer alias analysis, therefore, we simply ignore these instructions.

- Since FA analysis is flow-insensitive and context-insensitive, instructions whose only effect is on intra-procedural control flow, such as conditional and unconditional branches, have no effect on the analysis. Inter-procedural control flow cannot be ignored, however, since it induces aliasing between the actual parameters at the call site and the formal parameters at the callee. Our decompiler therefore ignores conditional and unconditional control flow instructions whose targets are within the same function, but translates inter-procedural control transfers.

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

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

Our decompiler maps registers in the assembly code to global variables of type `int` with 32-bit values; the 16-bit and 8-bit registers (which are aliases of parts of the 32-bit registers) are also mapped to the appropriate 32-bit global. Thus, the 8-bit register `%al` and the 16-bit register `%ax`, which refer to the low 8 bits and the low 16 bits of the 32-bit register `%eax` respectively, are both mapped to the variable `eax` denoting the 32-bit register `%eax`. 
Memory locations referenced as absolute addresses in the assembly code are also treated as global variables. Since there is little type information available at the assembly level, we declare memory locations as having type `MEMOBJ`, which denotes a word in memory. An object spanning a series of memory locations in the assembly code is treated as an array of `MEMOBJ` in the generated C code. This is illustrated in Figure 3.2(a). The segment of assembly code shown on the left side in Figure 3.2(a), taken from the file `entry.S` in the Linux kernel, defines the system call table that contains the function addresses of all system call handlers. Since `sys_call_table` spans a series of (initialized) memory locations in the assembly code, we map it to an (initialized) array in the

---

`typedef MEMOBJ int;`
generated C code shown on the right side in Figure 3.2(a). Moreover, since the symbols for the system call handlers are not themselves defined in entry.S, they are declared as extern objects in the generated C code. Before we start the actual pointer analysis, we scan the entire kernel source code and match memory objects to functions so that the source-level FA analysis can deal properly with function pointers in the assembly code.

Functions in the assembly code are identified from symbol table information and mapped to functions in the generated C code. Memory locations accessed through the stack pointer register %esp are assumed to be on the stack; these are mapped to local variables in the corresponding C function, with variables accessed via different displacements within the stack frame being mapped to different local variables in the generated C code. Actual parameters to a call are identified by similarly examining displacements in stack references, as illustrated in Figure 3.2(b). In this manner, by examining the references to actual parameters in the body of a function, we can determine the number of arguments it takes, and thereby generate a function prototype in the C code. Such prototypes are then used by the source-level analysis to identify aliasing between actual parameters and formal parameters. A control transfer to a symbol $S$ is translated as a function call if either the instruction is a call instruction, or if the target $S$ is a function, as illustrated in Figure 3.2(c).

3.3 Identifying Reachable Code

3.3.1 Reachable Analysis

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

Unlike ordinary applications, an operating system kernel contains multiple entry points. These entry points are the starting points for our reachable analysis. We classify kernel entry points into four categories: (1) the entry point for initializing the kernel (for the Linux kernel, this is the function startup_32), (2) system calls invoked during the kernel boot process, (3) interrupt handlers, and (4) system calls invoked by user appli-
cations. Once the entry points into the kernel have been identified, our reachable analysis performs a straightforward depth-first traversal of the program call graph to identify all the reachable functions in the kernel.

3.3.2 Improving the Analysis

During the initialization phase of kernel bootup (e.g., before the `init` program in Linux begins execution), execution is deterministic because there is only one active thread and execution depends only on the hardware configuration and the configuration options passed through boot command line. In other words, the initialization code can be considered to be “static” in the partial evaluation sense [37]. This means that if the configuration is not changed, we can safely remove any initialization code that is not executed. We use this idea to further improve our reachable analysis.

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

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

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

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

\[
\text{worklist} \leftarrow \text{functions that are entry points into the kernel}
\]

\[
\text{while } \text{worklist} \neq \emptyset \text{ do}
\]

\[
f \leftarrow \text{Remove a function from worklist}
\]

Mark \(f\) as reachable

\[
\text{for every indirect/direct call target } c \text{ of } f \text{ do}
\]

\[
\text{if } c \text{ is static } \land c \text{ is not executed based on profile then}
\]

\[
\text{continue}
\]

\[
\text{else if } c \text{ is not marked as reachable then}
\]

\[
\text{Add } c \text{ into worklist}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{end while}
\]

Figure 3.3: The improved reachable analysis algorithm

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

Once we have computed the set of functions that are considered to be static during kernel initialization, we use the results to improve our reachable analysis as shown in Figure 3.3. The improvement is that when a potentially reachable function is found, if the function is marked static and if, based on profile data, it was not called during kernel initialization, then we do not add it to the set of reachable functions.

### 3.4 Kernel Compaction

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

1. **Unreachable code elimination.** This identifies and deletes code that cannot be reached during execution [19].

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

3. Duplicate code elimination. This involves transformations (other than whole function abstraction) where duplicate code instances are identified and eliminated. We use two different code transformations for this: tail-merging and procedural abstraction [19].

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

The first involves a small number of functions in the kernel bootup code that execute prior to page table initialization, and which are required to be at specific fixed addresses. Such functions therefore cannot be moved in memory during the code compaction process. Our current implementation uses a fixed list of functions that cannot be moved: there are some 71 such functions, out of a total of roughly 4,600 functions in the input kernel binary (see Table 3.1).

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

3.5 Experimental Results

We have implemented our ideas using the PLTO binary rewriting system for the Intel x86 architecture [56] and evaluated them using two different versions of the Linux kernel:
With networking & Without networking

<table>
<thead>
<tr>
<th></th>
<th>With networking</th>
<th>Without networking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>4,584</td>
<td>3,882</td>
</tr>
<tr>
<td>Basic blocks</td>
<td>72,951</td>
<td>55,708</td>
</tr>
<tr>
<td>Instructions</td>
<td>268,335</td>
<td>205,587</td>
</tr>
<tr>
<td>Code size (Kb)</td>
<td>836.70</td>
<td>641.61</td>
</tr>
</tbody>
</table>

(a) Linux 2.4.25

<table>
<thead>
<tr>
<th></th>
<th>With networking</th>
<th>Without networking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>4,388</td>
<td>3,914</td>
</tr>
<tr>
<td>Basic blocks</td>
<td>71,118</td>
<td>51,609</td>
</tr>
<tr>
<td>Instructions</td>
<td>261,188</td>
<td>209,352</td>
</tr>
<tr>
<td>Code size (Kb)</td>
<td>830.68</td>
<td>645.89</td>
</tr>
</tbody>
</table>

(b) Linux 2.4.31

Table 3.1: Static kernel characteristics

versions 2.4.25 and 2.4.31. To get an accurate evaluation of the efficacy of this system, we began with a minimally configured kernel where as much unnecessary code as possible has been eliminated by configuring the kernel carefully. For our experiments, we therefore configured the Linux kernel to remove modules, such as the sound card and video support, that are not required to run our benchmarks. We considered two configurations for each kernel: one with networking support and the other without. The kernel code was compiled with gcc version 3.4.4 using the compilation flags ‘-Os -fomit-frame-pointer’, which instructs the compiler to optimize for code size. Table 3.1 gives various size-related statistics for the resulting kernel images. In order to simplify the booting process of the Linux kernel, we modified the kernel boot up file inittab so that the Linux kernel will run in single user mode (level 1). Based on the profile data, there are 81 different system calls that are invoked during the booting process.

We used programs from the MiBench suite [31], a widely used and freely available collection of benchmark programs for embedded systems, to evaluate our approach. We augmented the original six program sets in MiBench with two additional sets: Entertainment, representing a multi-media consumer appliance for music and digital pictures; and
<table>
<thead>
<tr>
<th>Benchmark set</th>
<th>Programs</th>
<th>No. of unique system calls</th>
<th>No. of non-bootup system calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto./Industrial</td>
<td>basicmath, bitcount, qsort, susan</td>
<td>33</td>
<td>9</td>
</tr>
<tr>
<td>Consumer</td>
<td>jpeg, mad, lame, tiff2bw, tiff2rgba, tiffdither, tiffmedian, typeset</td>
<td>46</td>
<td>11</td>
</tr>
<tr>
<td>Network</td>
<td>dijkstra, patricia (blowfish, CRC32, sha)</td>
<td>43</td>
<td>12</td>
</tr>
<tr>
<td>Office</td>
<td>ghostscript, ispell, rsynth, stringsearch</td>
<td>57</td>
<td>15</td>
</tr>
<tr>
<td>Security</td>
<td>blowfish, pgp, rijndael, sha</td>
<td>49</td>
<td>10</td>
</tr>
<tr>
<td>Telecomm</td>
<td>adpcm, CRC32, FFT, gsm</td>
<td>39</td>
<td>11</td>
</tr>
<tr>
<td>Entertainment</td>
<td>jpeg, lame, mad</td>
<td>43</td>
<td>10</td>
</tr>
<tr>
<td>Cellphone</td>
<td>blowfish, sha, CRC32, FFT, gsm, typeset</td>
<td>45</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3.2: Characteristics of the benchmarks used (from the MiBench suite)

Cellphone, representing a cell phone with security features. Characteristics of these sets are shown in Table 3.2. We also considered the BusyBox embedded toolkit [61], which was used by Chanet et al. to evaluate their kernel compaction work [12].

Before we can carry out kernel compaction for any given benchmark set, we have to identify the system calls that can arise from programs in that set. It is not enough to examine their executions using tools such as strace, since this may not cover all the execution paths in the programs. Nor is it enough simply to examine the source code of the benchmarks for system calls, since these actually call library routines that may contain additional system calls not visible in the source code. We therefore analyze statically linked binaries of the programs to ensure that we find all the system calls that may be invoked. This, however, causes the entire C library to be linked in. We address this problem by first carrying out a reachable analysis on the application program binaries to identify and eliminate unreachable library routines (using a conservative approximation to deal with indirect function calls) and then traversing the resulting whole-program control flow graph to determine the set of possible system calls. These data are shown in Table 3.2: the third column of this table gives the number of different system calls across all of the programs in each set of benchmarks, while the fourth column gives, for each benchmark set, the number of system calls not occurring in the set of system calls invoked during the kernel bootup process. Once we have the system calls that may be invoked by a set of
Figure 3.4: Results of code size reduction programs, we use them to identify and eliminate unreachable code in the kernel.

3.5.1 Code Size Reduction

Figure 3.4 shows the effects of code compaction. Figure 3.4 a) shows the results of code size reduction for Linux 2.4.25 kernel and Figure 3.4 b) shows the results for Linux 2.4.31 kernel. The mean overall code size reduction achieved for the Linux 2.4.25 kernel is 19.3% for the version with networking code and 23.75% for that without networking; for the Linux 2.4.31 kernel, these numbers are 22.4% for the version with networking...
code and 22.6% for the version without networking code. The detail experimental data is shown in Section A.1.

The bars labeled “All system calls” in Figure 3.4 show how much code compaction is achieved if all system calls in the kernel are assumed to be invokable by the application code. There are two conclusions that can be drawn from this. First, it is evident that our optimizations are able to achieve significant code size reductions (around 12%–16%) on a carefully configured kernel, even if we make no assumptions about what system calls can be invoked by applications. Second, it can be seen that the ability to restrict the set of possible system calls, based on knowledge of the application code, can yield significant benefits, in our case giving an additional savings of 7%–9%.

3.5.2 Effects of Different Optimizations

Table 3.3 shows the effects of different optimizations on code size. For each optimization, we show both the incremental improvement obtained from adding that optimization as well as the cumulative benefit of these optimizations. It can be seen that the largest contribution, about 15%–21%, comes from unreachable code elimination. For the Linux 2.4.25 kernel, whole function abstraction gives an additional improvement of about 3%; the effects of duplicate code elimination, while noticeable, are much smaller (under 1%). For the Linux 2.4.31 kernel almost all of the code size reduction comes from unreachable code elimination: the effect of whole function abstraction is essentially negligible, while that of duplicate code elimination is only around 1%. The large difference in the effects

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Optimization</th>
<th>WITH NETWORKING</th>
<th>WITHOUT NETWORKING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Incremental benefit (%)</td>
<td>Cumulative benefit (%)</td>
</tr>
<tr>
<td>2.4.25</td>
<td>unreachable code elimination</td>
<td>14.56</td>
<td>14.56</td>
</tr>
<tr>
<td></td>
<td>whole function abstraction</td>
<td>3.16</td>
<td>17.72</td>
</tr>
<tr>
<td></td>
<td>duplicate code elimination</td>
<td>0.92</td>
<td>18.64</td>
</tr>
<tr>
<td>2.4.31</td>
<td>unreachable code elimination</td>
<td>21.32</td>
<td>21.32</td>
</tr>
<tr>
<td></td>
<td>whole function abstraction</td>
<td>0.03</td>
<td>21.35</td>
</tr>
<tr>
<td></td>
<td>duplicate code elimination</td>
<td>0.81</td>
<td>22.13</td>
</tr>
</tbody>
</table>

Table 3.3: Effects of different optimizations on code size
Table 3.4: Effects of different call target analyses on code size

of whole function abstraction between the two versions of the kernel arise from the fact that there are a number of distinct but identical functions in the 2.4.25 kernel that have been hand-optimized away in the 2.4.31 kernel. (Note that these optimizations can have overlapping effects, which means that these numbers can be different if the optimizations are considered in a different order. For example, we might see somewhat greater benefits from duplicate code elimination, and a little less from unreachable code elimination, if the former is carried out before the latter. This can happen because in such a situation, some of the duplication eliminated might be in code that turns out to be unreachable.)

3.5.3 Effects of Different Indirect Call Target Analyses

Since unreachable code elimination accounts for most of our code size savings, the precision of program analyses for identifying unreachable code plays a significant role in the amount of code that can be eliminated. Determining the possible targets of indirect function calls plays a crucial role in this. Table 3.4 shows the effects of different indirect call target analyses on both the amount of unreachable code identified as well as the overall code size reduction achieved. We consider three different scenarios:

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

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

- **FA analysis** refers to the source-level pointer alias analysis described earlier in Section 2.5.

Not surprisingly, since no function is identified as unreachable in the conservative case, unreachable code analysis yields no savings; all the code size reduction in this case comes from whole function abstraction and duplicate code elimination. This yields code size reductions of about 4.5%–5% on the 2.4.25 kernel, but (because of fewer opportunities for whole function abstraction) only about 1.3%–2.4% for the 2.4.31 kernel. The straightforward address-taken analysis does surprisingly well, identifying about 11%–12% of the code as unreachable in the 2.4.25 kernel and 12%–16% in the 2.4.31 kernel; the overall size reductions achieved range from 13% to about 18%. FA analysis identifies about 14.6%–20% of the code as unreachable in the 2.4.25 kernel and about 21% of the code in the 2.4.31 kernel; this yields overall improvements of about 19%–22%. Because of the availability of higher-level semantic information, the source-level FA analysis is able to attain improvements over the address-taken analysis ranging from 2.5%–3% for the 2.4.25 kernel to over 9% for the 2.4.31 kernel with networking.

### 3.5.4 System Call Popularity

The amount of compaction achieved for each set of applications depends on the particular set of system calls made by the set of application programs. To get an idea of the extent to which our results might generalize to other sets of embedded applications, we evaluated the “popularity” of different system calls across the MiBench suite. The popularity of a given system call $s$ in a set of programs $P$ is given by the fraction of programs in $P$ that use $s$. Intuitively, if different kinds of applications use very different sets of system calls, i.e., many system calls have low popularity, then our results may not generalize well; on the other hand, if different kinds of applications tend to have mostly-similar sets of system calls, then we can expect these results to generalize. The results are shown in Figure 3.5.
Figure 3.5: System call “popularity” in embedded applications

It can be seen that out of some 226 different possible system calls in our system, there is a small core of 32 system calls that are used by every program. Popularity drops off sharply outside this core set. All of our benchmark programs, taken together, refer to only 76 system calls, i.e., about a third of the total set of system calls.

This relative uniformity in system call usage across a wide variety of applications helps explain the surprising uniformity of our code compression results across all of the benchmark sets. While the applications themselves are very different in terms of their nature and code size, the popularity data shown in Figure 3.5 show that they do not differ from each other hugely in terms of their interactions with the operating system kernel: for example, they typically read data from some files, process that data, and write out the results. Moreover, in addition to the system calls made by application code, the kernel itself makes 81 different system calls during the bootup process. The overall result is that the set of “non-bootup” system calls arising in the application code is relatively small, and does not vary greatly from one benchmark set to another (Table 3.2, col. 4). Because of this, the compaction results for the different benchmark sets tend to be similar.

There were 259 syscall entries in our version of Linux kernel; of these, 33 were not implemented (“sys_not_syscall”), leaving a total of 226. Other sources put the number of Linux system calls much higher: e.g., Wikipedia mentions “almost 300” system calls for Linux.
3.6 Summary

This chapter describes a compaction-based approach to reduce the code size of an OS kernel. We begin with the observation that embedded systems typically run a small fixed set of applications. This knowledge can be used to identify the minimal functionality required by the kernel code to support those applications and then to discard unnecessary code. We discuss a number of technical challenges that have to be addressed in order to make this work; in particular, we describe “approximate decompilation,” which allows us to apply source-level program analyses to hand-written assembly code. Our ideas have been implemented in a prototype binary rewriting tool that is able to achieve a code size reduction of close to 24% on an already minimally configured Linux kernel.
In previous chapter, we show that compaction-based approach is able to accomplish significant code size reductions. However, the previous approach has the limitation that the entire kernel code, whether or not it is actually executed, is kept in memory. This can unnecessarily limit the amount of code size reduction achievable.

Keeping all the kernel code in memory might be reasonable if most or all the kernel code was actually executed. It turns out, however, that (at least for embedded applications) most of the kernel code is executed infrequently, if at all. When running the MiBench embedded application suite [31] on a minimally configured Linux kernel, for example, we found that out of a total of 213,862 instructions (occupying a total of 633.7 KB of memory), only 71,298 instructions (occupying 202.8 KB of memory)—i.e., about 32%—were actually executed. However, although about 68% of the kernel code was not executed, existing code optimization techniques are able to prove only about 20% of the code to be unreachable; the remaining code cannot be discarded because in theory it could be executed depending on other inputs to the applications running. However, keeping this code in memory uses up a scarce resource.

While embedded systems typically have a limited amount of main memory, they often have a considerably greater amount of secondary storage available, e.g., in the form of flash memory. This chapter describes an automated approach that takes advantage of this feature to reduce the main memory requirements of the OS kernel code. The essential idea behind our approach is to keep rarely used code on secondary storage, and load it into main memory if and when it is needed. We apply a clustering algorithm to the whole-program control flow graph of the kernel to group “related” code fragments together; this has the effect of dividing up the OS kernel code into partitions and allows us to reuse the main memory buffer, into which code is loaded on demand, for different partitions at different times. If necessary, a very minor modification of the kernel code
suffices to deal with multi-threading issues. The result is reminiscent of the old idea of code overlays, except that in our case the entire process of clustering the code into partitions and transforming it to manage the code overlays, is automatic. Our approach does not require hardware or operating system support for virtual memory. Therefore, it is also applicable to low-end embedded system without virtual memory or MMU hardware support. Experiments with the Linux kernel indicate that we are able to reduce the main memory requirements of the OS kernel code by around 53% with very little degradation in performance.

4.1 On-Demand Code Loading

As mentioned earlier, the main idea behind our approach is to keep the commonly executed OS kernel code (the “hot” code) in main memory while moving the rarely executed code (the “cold” code) to secondary storage, and with appropriate adjustments to load cold code into memory when (if) it is needed. The code transformation needed to accomplish this basic functionality is conceptually straightforward: on each control flow edge that goes from the hot code to the cold code, insert some code to load the cold code from secondary storage into memory; and adjust addresses to reflect the fact that the cold code may execute at a different location in memory than its original address when it is loaded. Note that while code for on-demand loading is inserted along control flow edges from the hot code in memory to the cold code in secondary storage, no corresponding code is needed on edges from the cold code to the memory-resident code. This is because the memory resident code always stays in memory, while the code loaded from secondary storage is not modified and so does not need to be written back to secondary storage.

This straightforward idea is much too simplistic, however. We have to reserve enough space in memory to accommodate any code that may have to be loaded; we call this region of memory the code buffer. If all we do is that described above, then the code buffer has to be large enough to accommodate all of the cold code, resulting in no net space savings (it would actually make things a little worse because of the additional code needed for on-demand loading of code from secondary storage).
It is evident from this discussion that we have to do three things in order to make our approach profitable. First, we have to determine how much code is to be memory-resident and how much is to be kept in secondary storage: increasing the amount of code moved to secondary storage leads to a reduction in the memory footprint of the kernel, but can potentially lead to runtime overhead increased because of the need to load code into memory during execution. Second, we have to load the cold code in smaller chunks so as to reduce the time and energy cost of loading code. Finally, the memory region where these code chunks are loaded has to be reused, i.e., the code buffer needs to be able to hold different chunks of code at different times. If the code loaded from secondary storage is partitioned into $k$ chunks, of sizes $n_1, \ldots, n_k$, then this code buffer reuse makes it possible to use a buffer of size $\max_{i=1}^{k} n_i$ instead of $\sum_{i=1}^{k} n_i$ as in the case with no reuse. However, these requirements raise some technical issues, which we discuss in the remainder of this section. We deal with first issue via a user-specifiable threshold that controls the amount of code that is memory-resident; this issue is discussed in Section 4.1.1. The second requirement implies that we have to be able to divide the cold code into different chunks in some reasonable way, so as to minimize the total number of loads from secondary storage; we discuss this in Section 4.1.2. The third requirement introduces some complications into the handling of control transfer between two different dynamically loaded code chunks; this issue is discussed in Section 4.1.3.

4.1.1 Upper-Bound of Memory Requirement for Kernel Code

The amount of memory required for kernel code is determined by two factors: the size of the code that is always kept in memory and the size of code buffer, i.e., the memory region that is used to keep the code that is loaded dynamically. The sum of these two values gives an upper bound on the memory usage for the kernel code.

In our approach, the code that is always kept in memory (memory-resident) consists of two categories:

- The core code, which is the code that has to be in memory to make the kernel work correctly. This includes the scheduler, the memory management, the trap and
interrupt handling code in the kernel, and—in the final overlay-based version of our
system—the code that manages overlays. Our current implementation identifies
such code by having the user designate a specific set of kernel functions as core
code.

- The “hot” code, i.e., frequently executed code that is kept in memory in addition to
the core code for performance reasons. Our approach uses a user-specified parameter $\gamma$ that determines how much code is kept in memory. In our current implementation, $\gamma$ is specified as the percentage of extra code allowable as memory-resident with respect to the size of the core code. For example, if the core code occupies 100 KB in the input kernel binary and $\gamma = 10\% = 0.1$, then the memory-resident code in the transformed kernel is allowed to be up to 10% larger than the core code in the input binary, i.e., up to 110KB.

We assume that the size of code buffer is given as an input parameter (the experimental results reported in this work were based on the size of code buffer $= 2$ KB). The size of code buffer, denoted as $BufSz$, limits how large each code chunk(cluster) can be in order to be accommodated in the code buffer. With parameter $\gamma$ and $BufSz$, the upper-bound of memory usage for kernel code equals to

$$size(\text{core code}) \times (1 + \gamma) + BufSz.$$

4.1.2 Code Clustering

In order to minimize the number of reads for code from secondary storage, we use a greedy node-coalescing algorithm for clustering. We begin with an edge-weighted whole-program control flow graph for the kernel. The edge weights, representing execution frequency counts, are obtained via edge profiling. Since there is a significant amount of indirect control transfers through function pointers in the kernel, we also perform target profiling for all indirect control transfer instructions to collect the weights for indirect edges and add them into the whole-program control flow graph.
Input: A basic block \( bbl \)

Output: The estimated final size of \( bbl \)

Method:

\[
N = \text{the total memory size of original instructions in } bbl.
\]

if \( bbl \) contains a control flow edge to a different cluster or an indirect edge (means control target is unknown) do

return \( N + \) size increased due to code transformation. (see Section 4.1.3).

else

return \( N \).

fi

Figure 4.1: The \( \text{BlockSize} \) function

From the edge-weighted whole-program control flow graph, we construct an edge-weighted graph, the cluster graph, whose nodes represent the code clusters and whose edges represent control transfers between clusters.

The details of our clustering algorithm are shown in Figure 4.2. The algorithm operates on chains of basic blocks that must be contiguous in memory, e.g., due to fall-through edges or the return from a function call. There are four major steps in the algorithm:

1. \text{Create a cluster graph}

Initially, each chain is assigned to a separate cluster; there is an edge \( e \) between two clusters \( a \) and \( b \) if there are any control flow edge between \( a \) and \( b \), and the weight of the edge \( e \) is computed as the total weight of all the control flow edges between the blocks in \( a \) and \( b \).

2. \text{Compute the code size and cluster size for each node}

The cluster size of each node determines how much code a cluster can hold. For each node other than the node \( C \) for the core code, the cluster size is given by \( BufSz \), the size of the code buffer. The cluster size of \( C \) is the final size of code that
Input:

1. An edge-weighted control flow graph for a program, together with a function \( \text{BlockSize} \) that gives an estimate of total memory size of each basic block.
2. A set of functions \( \mathcal{F} \) that must reside in memory. The code for these functions comprises the core code.
3. A bound \( \gamma \) on the final size of the memory-resident code.
4. An integer \( \text{BufSz} > 0 \) giving the size of code buffer.

Output: A cluster graph for the program.

Method:

1. Create a cluster graph \( G = (V, E) \) as follows:
   - \( V \) contains a single node \( C \) corresponding to the core code, as well as a node for each basic block chain \( B \) such that \( B \neq C \).
   - There is an edge \((a, b)\) between nodes \( a \) and \( b \) in the cluster graph if there is a control transfer edge between some basic block in \( a \) and some basic block in \( b \). The weight \( w(e) \) of an edge \( e = (a, b) \) is given by the total edge weight of all control flow edges between blocks in \( a \) and those in \( b \).

2. Compute the code size and cluster size for each node:
   - For each node \( a \) in the cluster graph, let \( a.\text{code size} = \sum \{ \text{BlockSize}(bbl) \mid bbl \in a \} \).
   - Let \( C.\text{cluster size} = C.\text{code size} \times (1 + \gamma) \).
   - For each node \( a \neq C \), let \( a.\text{cluster size} = \text{BufSz} \).

3. [Node coalescing] Process the edges of the cluster graph \( G \) in descending order of weight, iteratively coalescing nodes:

   ```plaintext
   while \( \exists (a, b) \in E \text{ s.t. } (a.\text{code size} + b.\text{code size}) \leq \max(a.\text{cluster size}, b.\text{cluster size}) \) do
     Coalesce the endpoints \( a \) and \( b \) and merge \( b \) with \( a \), setting:
     \[ a.\text{code size} = a.\text{code size} + b.\text{code size}; \]
     \[ a.\text{cluster size} = \max(a.\text{cluster size}, b.\text{cluster size}) \).
     Update edge weights for all clusters adjacent to \( a \) and \( b \) appropriately.
   od
   ```

4. [Defragmentation] Coalesce small clusters into larger ones where possible:

   ```plaintext
   while \( \exists a, b \in V \text{ s.t. } (a.\text{code size} + b.\text{code size}) \leq \max(a.\text{cluster size}, b.\text{cluster size}) \) do
     Merge \( b \) with \( a \), setting:
     \[ a.\text{code size} = a.\text{code size} + b.\text{code size}; \]
     \[ a.\text{cluster size} = \max(a.\text{cluster size}, b.\text{cluster size}) \).
   od
   ```

5. Return the resulting cluster graph.

Figure 4.2: The Clustering Algorithm
is always in memory. The upper-bound is determined by the size of core code and the code-size bound specified by the parameter $\gamma$.

The code size for a cluster is computed as the total size of all of the basic blocks in that cluster. The function $\text{BlockSize}$, shown in Figure 4.1, is used to compute the memory size of a basic block. This is given by the total size of the instructions in the basic block together with the size of any additional code that is inserted to support overlays. The reason for the latter code is that if there is an inter-cluster control transfer in a basic block, the control transfer instructions in the basic block have to be modified to deal with overlays (see Section 4.1.3).

3. **Node coalescing**

The algorithm then processes the cluster edges in descending order of weight, iteratively coalescing the end-points of edges whenever possible until no further coalescing can be carried out. Two nodes $a$ and $b$ can be coalesced if doing so will not cause the size of the resulting (coalesced) node to violate the following conditions:

- If neither $a$ nor $b$ is the core node $C$, then the size of resulting node must not exceed the cluster size of either node, which is equal to $\text{BufSz}$.
- If either of $a, b$ is $C$, the size of resulting node should not exceed $\max(C.\text{cluster\_size}, \text{BufSz})$. In this case, the code of the other node becomes “hot” code and memory-resident as well.

4. **Defragmentation**

At the end of this step, there are usually some small clusters left over; a defragmenting step is carried out at the end to merge such clusters with larger ones where possible.

In this algorithm, the bound parameter $\gamma$ controls the final size of memory-resident code. If $\gamma = 0$, only the core code will be kept in memory. All other code, including even hot code, will be kept in secondary storage, which likely results in a large number of reads from secondary storage and a concomitant high runtime overhead. Larger values
of $\gamma$ mean that some additional code can be kept memory-resident. Since we process the cluster edges in descending order of weight, this will cause some of the frequently executed code (which must be small enough to fit into the additional memory space that is now available) to be coalesced with the cluster corresponding to the core code. This can result in reducing runtime overheads because less code will have to be loaded at runtime. Our experimental results, reported in Section 4.2, confirm this.

4.1.3 Code Transformation

Once clustering has been done, the next step is to transform the kernel code to support overlays. We add a small amount of code into the core cluster (i.e., the cluster that contains the core code) for managing code loading at runtime. We call this code as overlay manager. The overlay manager consists of:

- A *dynamic loader*, which is given an address that is the target of a control transfer instruction into the code that needs to be loaded into the memory. The dynamic loader looks up this address in the cluster address table\(^1\) to identify the cluster that it belongs to, then loads the code for that cluster from secondary storage into the code buffer.

- Two *control transfer routines*: \_\texttt{dynamic\_call} and \_\texttt{dynamic\_jmp}. The first of these, \_\texttt{dynamic\_call}, handles the case where control transfer into the target cluster is a function call, while the second routine, \_\texttt{dynamic\_jmp}, handles the case where the control transfer is a jump instruction. Conceptually, these two routines are very similar in their essential functionality: they invoke the dynamic loader to load the target cluster into memory, translate the target address into the appropriate offset within the code buffer, then branch to that location. The only difference between

\(^1\)The cluster address table stores the starting address of each dynamically loaded clusters (since dynamically loaded clusters are placed in contiguous address space, it is enough to keep only the starting address of each dynamically loaded cluster in the table). The table is loaded into the memory at the very beginning when kernel starts.
them is that when the control transfer is a function call, the return from that call
continues execution at the instruction after the call instruction, and some extra
book-keeping is necessary to handle this, as described below.

The inter-cluster control flow edges where the target cluster is not (or, in the case
of indirect control transfer, may not be) the core cluster need to be changed so that the
overlay manager can take over the control of the execution. The transformation is rather
straightforward. We transform the code to push the target address of inter-cluster control
flow edges on the stack and then branch to the appropriate control transfer routines:

- A direct unconditional jump \( \text{jmp } \ell \) is transformed to code that simply pushes the
target address and jumps to \_dynamic\_jmp:

  \[
  \begin{align*}
  &\text{push } \ell \\
  &\text{jmp } \_\text{dynamic\_jmp()}
  \end{align*}
  \]

- The code for an indirect jump \( \text{jmp } *r \) is similar as a direct jump:

  \[
  \begin{align*}
  &\text{push } *r \\
  &\text{jmp } \_\text{dynamic\_jmp()}
  \end{align*}
  \]

- A conditional jump instruction \( \text{J}_{cc} \ell \) is transformed to code of the form

  \[
  \begin{align*}
  &\text{J}_{cc} A \\
  &\text{...}
  \end{align*}
  \]

  A: push \( \ell \\
  \text{jmp } \_\text{dynamic\_jmp()}
  \]

The transformation of function calls is analogous to that shown above for uncon-
ditional jumps and indirect jump, except that it branches to \_dynamic\_call. Since the
transformation makes changes to the stack, the control transfer routines need to clean up
the stack before it branches to the code buffer.

The control transfer routines \_dynamic\_call and \_dynamic\_jmp are similar except that there is one important difference that arises from a subtlety in dealing with function
calls from one dynamically loaded cluster into another. This is illustrated in Figure 4.3. Suppose we have a function call from a dynamically loaded cluster $A$ to a function $f$ in a different dynamically loaded cluster $B$. Figure 4.3(a) shows the machine state at the point of the call: the code buffer contains cluster $A$, and the return address, at some offset $m$ in the code buffer, is the instruction following the call instruction. Since cluster $B$ is also dynamically loaded, this call causes the code for $B$ to be loaded into the code buffer, thereby overwriting the code for $A$ that has been there. When control returns from the function $f$, therefore, the return address—which is simply the address of offset $m$ in the code buffer—actually points to an instruction in cluster $B$ rather than one in cluster $A$.

We deal with this problem as follows. The control transfer routine \texttt{dynamic\_call} checks whether the return address is within the code buffer. If it is, it creates a restore stub routine dynamically (using heap memory) for the return address. The purpose of the restore stub routine is to reload the cluster $A$ from which the call is originated, then jump to the appropriate offset within the code buffer. The return address passed to the callee is modified to point to the restore stub. The restore stub simply invokes the dynamic loader with the appropriate address to cause the required cluster (cluster $A$ in this case) to

\footnote{In an architecture with variable-length instructions, such as the Intel x86, the return address may not even refer to a valid instruction.}
be loaded,\(^3\) then cleans up the stack and branches to the proper location within the code buffer (i.e., the instruction that follows the original call instruction).

The mechanism is able to handle a chain of calls among different clusters properly as well. For example, suppose that there is a call chain, \(a \rightarrow b \rightarrow c \rightarrow d\), where \(a, b, c,\) and \(d\) are functions belonging to different clusters. There are 3 different restore stubs created for this call chain—one for each call-site. When a function, say \(d\), returns, the program control first jumps to the corresponding restore stub for call \(c \rightarrow d\) and the restore stub calls dynamic loader to load the cluster of which \(c\) belongs to into the code buffer. Then the program counter is set at the appropriate location in the code buffer so that the execution can be continued in function \(c\). The other two restore stubs act in a similar way when the function \(c\) and \(b\) returns.

The size of a restore stub is small and consumes little amount of memory. For efficient purpose, a multiple instances of restore stubs are created initially and during the execution, additional restore stubs can be created if necessary. In practice, we choose to create 20 restore stubs initially (during the kernel initialization) and reuse them. Because the Linux kernel has a small fixed-size stack (only 4KB in our tested kernel), the call stack of the Linux kernel is usually not very deep. This number is large enough for all the experiments we tested.

4.1.4 Context Switches and Interrupts

There are two issues that have not been addressed in the discussion so far. The first is that of multi-threading, which is a typical feature of modern general-purpose operating system kernels. The second is that of control transfers due to interrupts.

We extend our approach to handle multi-threading via minor (manual) modifications to the kernel code, as follows. We add a single new field to the thread state (in the Linux kernel, the structure `task_struct`) to identify the cluster whose code is being executed by that thread. The additional memory requirements for this are small, just 4 bytes per

---

\(^3\)Since the return address is actually within the code buffer in this case, the routine `_dynamic_call` maps it back to an address that, when passed to the dynamic loader by the restore stub, causes it to load the appropriate cluster into memory.
thread. This field is initialized and updated by the dynamic loader as needed during execution. The code for the scheduler is modified to check the program counter value for a thread that is about to run, and to invoke the dynamic loader to load the appropriate cluster into the code buffer if this address is within the code buffer. The code changes necessary are shown in Figure 4.4, where the additional code that has to be introduced in our approach has been highlighted.

The second issue is that of dealing with interrupts. When the kernel is executing a code cluster $A$ in the code buffer and an interrupt happens, if the interrupt handler brings a different code cluster $B$ into the code buffer, $B$ will overwrite cluster $A$ in the code buffer. Once the interrupt handler finishes and the kernel execution returns back to its previous normal execution, if $A$ is not reloaded into the code buffer, an error will happen. In our current implementation, we handle this issue by making sure that interrupt handlers are part of the core code and therefore always remain in memory so that interrupt handler will not load new code cluster into the code buffer. This, however, will obviously increase the size of core code. There are possible other solutions to handle this. One possible approach is to have two code buffers: one is dedicated to the normal kernel execution other than interrupt handling; while the other is dedicated only to the interrupt handling. Another approach is to modify the return process of interrupt handler so that the required code cluster is loaded into the code buffer before the interrupt handler returns. However,
both approaches require substantial modifications in the original kernel code.

4.2 Experimental Results

We have implemented our ideas using the PLTO binary rewriting system. PLTO is also used to collect profiles for the kernel as described previously in Section 2.4.1. In our current implementation, the dynamically loaded code is still stored in memory, but in a separate section in the kernel binary. The code is loaded into the code buffer through `memcpy` function call. All the experiments are conducted using a Intel Pentium 4 3 GHz desktop machine with 2GB memory installed.

We evaluated our prototype using Linux version 2.4.31. Similar to the experiments conducted in Section 3.5, the Linux kernel was first minimally configured and compiled. Three sets of benchmarks were used for our experiments: MiBench [31]; MediaBench, a suite of programs used for evaluating multimedia and communications systems [43]; and `httpd`, the Apache HTTP server (version 2.0.50).

We considered two kernels: one with networking support and one without. The kernel without networking was used for the experiments with MiBench and MediaBench since they do not actually require networking. The kernel with networking was used for the experiments with `httpd`.

4.2.1 Memory Reduction Results

Figure 4.5 shows the results of memory reduction of the Linux kernel code for all three benchmarks. The data presented corresponds to a code buffer size of 2 KB. This value was chosen because it is the page size on flash memory chip considered in Section 4.2.2. The detailed experimental data is shown in Table A.2 in Section A.2.

Since the code buffer size is being held constant in our experiments (2 KB), the total memory size reduction achieved decreases as γ—and therefore the amount of memory-resident code—increases. The decreasing is relatively small as the amount of hot code added in memory is not big (even for γ = 0.1). The memory size reductions achieved are fairly consistent across our benchmarks, and range from about 56%–58% for γ = 0 to
Figure 4.5: Code size reductions for different core code growth bounds (BufSz=2KB) about 53%–55% for $\gamma = 0.1$.

4.2.2 Cost of Code Loading

The benchmarks were run as follows: for the MiBench suite, we ran both the small and large input sets that came as part of the suite; for MediaBench, we used the run scripts provided with the suite; for httpd, we used the following command $^4$

$$\text{ab} -n 5000 -c 2 \text{http://test_addr},$$

which sends a total of 5000 requests, 2 at a time, to the test machine whose IP address is given by test_addr.

$^4$ab is the Apache HTTP server benchmarking tool.
Since our experiments were done on a relatively fast desktop environment, the small amount of time spent in the operating system kernel, together with the granularity of the system clock, made it difficult to reliably measure the effect of our dynamic code loading scheme on the total time spent within the kernel. Instead, we give a rough estimate of the effect of such a scheme in an embedded context.

First, we estimate the time taken for dynamic code loading out of flash memory secondary storage using manufacturer’s data sheets for a typical commercial flash memory currently in use. For this, we (quite arbitrarily) chose the Micron MT29f2G08AAAb NAND flash memory [46]. This is a 2 GB flash memory unit where data is stored in 2 KB pages. Data reads are done a page at a time (i.e., the smallest unit of data read is 2 KB), and it takes 130.9 microseconds to read each page. We estimate the cost of code loading as

$$\text{Est. Cost} = \sum_i \left(\frac{\text{size}(i)}{2048}\right) \times \text{access}(i) \times 130.9 \mu s,$$

where $\text{size}(i)$ is the size of cluster $i$ and $\text{access}(i)$ is the total number of times of which cluster $i$ was loaded into code buffer (The estimated cost is shown in columns 5 and 8 in Table A.3.).

Secondly, we tried to evaluate the impact of dynamic code loading on the performance of the application programs running on the kernel. We estimate this by considering the time taken to run each of the three benchmarks on an unmodified kernel (i.e., the cost of code loading is zero). On average, the total time for running each benchmark on an unmodified kernel is shown in Table 4.1.

Figure 4.6 shows the effect of different core code growth bounds on the runtime cost of on-demand code loading (The detail experimental results are shown in Table A.3 in Section A.2.). Let us consider MiBench as an example: by choosing $\gamma = 0$, it yields

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Running time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiBench</td>
<td>18.82</td>
</tr>
<tr>
<td>MediaBench</td>
<td>3.53</td>
</tr>
<tr>
<td>httpd</td>
<td>3.82</td>
</tr>
</tbody>
</table>

Table 4.1: Running time of all benchmarks on an unmodified kernel
a 56.7% reduction in code size, but also leads to an overhead that is almost 6 times of the runtime on an unmodified kernel. On the other hand, if we choose \( \gamma = 10\% \), there is 52.9% of reduction of code size while the overhead is smaller than 1%. For all three benchmarks, the overheads of dynamic code loading reduce significantly when the code growth bound is increased from 0% to 4%. The reason for this dramatic performance improvement is not hard to see: if the frequently executed parts of the kernel are kept in memory, they will not have to be loaded repeatedly from secondary storage.

It is important to note that these numbers are a conservative upper bound on the runtime overhead that would actually be incurred on an embedded platform. The embedded platforms our technique is aimed at are likely to be considerably slower than the desktop used for our experiments, which means that the time taken to run the whole MiBench
would be correspondingly much greater than the 18.82 seconds used for these calculations. Since the flash memory characteristics remain the same, it is reasonable to conclude that the actual runtime overheads experienced on an actual embedded system would be even lower.

Just as most profiled-based optimizations, our approach can suffer performance degradation if the actual execution run differs from the profiling runs. The results in Figure 4.6 show such an effect as the hot code were purposely removed from the memory. Such expected behavior of this approach may not be favorable for certain system designs, especially for those that require fast responding time, such as network communication and real-time environment. However, if there are such constraints for certain code in an OS, we can always put those code in the core code (i.e., always in the memory) to avoid dynamic code loading at runtime.

4.3 Summary

This chapter describes an approach that reduces the memory requirements of the OS kernel code via on-demand code loading. It uses edge profile information to carry out code clustering in order to reduce the cost of code loading. Experiments with the Linux kernel show that we are able to reduce the memory requirements of the kernel code to 53% with little degradation in performance.
The work described in previous two chapters has focused on reducing the code size of an OS kernel. We can consider the memory consumed by the code and global data as static since they can be examined by looking at the program binary image. However, the static components of an OS kernel—its code and global data—account for only a portion of its total memory footprint. Just as significant are the dynamic data, namely, the stack and heap memory, which can easily exceed the size of static memory.

It turns out that, in practice, there is quite often room to reduce the memory requirements for dynamic data. For example, programmers—possibly for convenience or portability reasons—very often allocate more memory to data structures than what are required by the values stored in them in specific environments or executions. Thus, scalar numerical values are often stored in integer-valued variables, which typically occupy a 32-bit machine word on a conventional 32-bit architecture. However, Brooks and Martonosi have shown that over half of the integer operations in SPECint95 can be represented with 16 bits or less [9]. Other opportunities for memory usage reduction arise from redundancy in the values of sets of pointer values, whose high-order bits typically share a common prefix.

The problem we are confronted with when trying to reduce a program’s dynamic memory requirements is that such opportunities for data compression are usually not obvious statically. For example, it may not always be readily apparent from examining the source code for a program, that a variable usually takes on only small-sized values or that a group of pointers all point to “nearby” addresses. Even if we could somehow identify such variables, it would be difficult to determine the set of possible values for a scalar variable or the common prefixes of addresses for a set of pointers. Furthermore, any change to the concrete representation of a variable, to reduce the amount of memory it occupies, will impact every reference to that variable: in order to preserve program semantics, it is
necessary to ensure that such impacts are properly handled. For example, if the value of a pointer variable is represented in a compressed form, then every assignment to that variable has to be changed to compress the value prior to writing it to the (compressed) variable, and every use of that variable must first decompress the value. Finally—and again for correctness reasons—any scheme to reduce the size of data representations must be robust enough to deal with all possible runtime values that a “compressed” variable may take on.

This chapter presents a technique called *dynamic data structure compression* that aims to address the problem of reducing the dynamic memory requirements of an OS kernel (although it is applicable to regular programs as well). Our technique uses profiling to detect opportunities for dynamic data size reduction, then transforms the code so that data values are maintained using a smaller amount of memory. The technique is *safe*: if a runtime value is beyond the value range that can be accommodated in the compressed representation of some variable, our approach automatically “expands” that variable to its original (un-optimized) size. Our experiments show that this approach can be quite effective in reducing the dynamic memory footprint of an OS kernel: applying our technique to the slab allocator in Linux kernel reduces the dynamic memory consumption of the slab allocator by about 17.5% when running the MediaBench suite, while incurring only a 5.4% increase in code size and a 1.9% increase in execution time.

5.1 Linux Kernel Slab Allocator

Slab allocation forms the core of dynamic memory allocation in the Linux kernel: the kernel memory allocation routines *kmalloc* and *kfree* (the kernel-level analogs of *malloc* and *free*) are built atop the slab allocator. For this reason, our prototype implementation targets the Linux slab allocator. This section gives a brief introduction to the slab allocator in the Linux kernel.

Slab allocation was adopted for the first time in the SunOS 5.4 kernel and introduced in Linux kernel since Linux 2.2 [25]. It provides an efficient mechanism to speed up dynamic memory allocation and reduce internal fragmentation in the kernel. The slab
allocator groups objects into *caches* where each cache stores objects of the same type. The caches are then divided into *slabs* (hence the name of this system). Each slab consists of one or more physically contiguous pages but typically consists of only a single page. To avoid initializing objects repeatedly, the slab allocator does not discard the objects that have been allocated and then released but instead saves them in memory. When a new object is then requested, it can be taken from memory without having to be reinitialized. Our approach does not need to modify the internal implementation of the slab allocator in the Linux kernel. Instead, the memory consumption of the slab allocator is reduced by compressing only the data structures that are used by the slab caches.

5.2 Data Structure Compression

The section describes the details of our approach for data structure compression. While we focus primarily on compressing dynamic data, our technique is more general and is applicable to any data structure.

A data structure is transformed into a more space-efficient structure by statically compressing its compressible fields, which include scalars and pointers, based on profile information. Our approach consists of the following steps:

1. **Data structure profiling.** We instrument a program, then run the instrumented code on training inputs to obtain profiling information about the values stored in all the compressible fields. This profiling information is called data value characteristics (the exact definition is given later in Section 5.2.1).

2. **Identifying candidate structures and choosing compression schemes.** Based on the data value characteristics obtained from profiling, we choose a compression scheme for each compressible field. In each case, the goal is to represent all profiled data values using as few bits as possible. To ensure that our optimization is safe, and does not change the observable behavior of a program, we exclude from consideration structure fields with certain properties; this is discussed in more detail in Section 5.3.3.
3. **Code modification.** The original program is then modified to support compressed data structures. Code that writes to a compressed field is modified so that the value being written is compressed appropriately before being stored into the compressed representation of the structure. Code that reads from a compressed field is modified to extract the value of the field from the compressed representation and decompress it appropriately.

It is important to note that since this compression is based on data obtained from profiling runs, it can happen that our scheme compresses a data field to $k$ bits but some other runs of the same program can produce a value for that field that requires more than $k$ bits to represent it. To handle this, our approach uses a scheme to expand the compressed data field with additional storage, as necessary, to hold the incompressible data.

### 5.2.1 Data Structure Profiling

The goal of data structure profiling is to obtain information about the values stored in variables and in the fields of aggregate data structures, in particular structs (i.e., records), in order to determine whether and how to compress them. Data structure profiling is done by instrumenting all field reference expressions in the source code of a program. In the C programming language, the targets for profiling are fields that are referenced through the operators ‘.’ and ‘->’.

Three kinds of data are collected for a compressible field: 1) **value range**, i.e., the minimal and maximum values that are encountered during program execution; 2) **distinct values**, which record the top $N$ distinct values presented in a profiled field [10, 60]; and 3) the number of references.\(^1\) Based on the characteristics of the data obtained from profiling, the values taken on by a field in a data structure can be classified into the following categories:

**Narrow width.** This refers to a set of values that can be represented by a small number of bits. For example, values from the set \{0, 2, 3, 5\} can be represented using only 3 bits.

\(^1\)This number is used later in our experimental evaluation to avoid compressing frequently-used fields and thereby reduce the runtime overhead.
### Table 5.1: Profiling data of dentry structure (value table size = 4096)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Min</th>
<th>Max</th>
<th># of distinct values</th>
<th>pg. prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_mounted</td>
<td>Integer</td>
<td>32</td>
<td>0x0</td>
<td>0x1</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>d_op</td>
<td>Pointer</td>
<td>32</td>
<td>0x0820b78c</td>
<td>0x0820ccd4</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>d_inode</td>
<td>Pointer</td>
<td>32</td>
<td>0x086480b4</td>
<td>0x09c82e34</td>
<td>4096</td>
<td>737</td>
</tr>
<tr>
<td>d_parent</td>
<td>Pointer</td>
<td>32</td>
<td>0x084fb1a4</td>
<td>0x0903cf74</td>
<td>349</td>
<td>99</td>
</tr>
</tbody>
</table>

**Common prefix.** This refers to a set of values whose binary representations have some number of high-order bits in common. In other words, there exists some $k > 0$ such that the top $k$ bits of these binary representations are the same for all of these values. This situation is encountered primarily with pointers.

**Small set.** This refers to a set of values of “sufficiently small” cardinality. The idea here is that a value can then be referred to by its index in the set, and the index can be represented using only a small number of bits.

### 5.2.2 Data Compression Techniques

Based on the characteristics of profiling data, as described above, we consider four kinds of compression techniques:

1. compression with narrow-width data ($NW$);
2. compression with common-prefix data ($CP$);
3. compression with compression table, which is used for small-set data ($CT$); and
4. a combination of 2 and 3 ($CT+CP$).

The first scheme is mainly used to compress non-pointer scalar type fields and the other three schemes are mainly used to compress pointer fields.

Figure 5.1 illustrates one of the Linux kernel data structure called dentry (directory entry), which is used to describe the name of a file in the Linux file system. For simplicity, we only list four compressible fields in dentry structure and use them as examples.
Table 5.1 shows the profiling data collected for these four fields. In Table 5.1, the second column is the type of each field and the third column is the bit width of each field. The fourth and fifth columns indicate the value range of each field. The fifth column is the number of distinct values of a field (a value table with size \(4K\) is used in experiments). The last column shows the data for “page prefixes,” a special case of value profiling in which the values are prefixes of the addresses of memory pages. Page prefix information is used for a particular type of data compression that combines the common-prefix and compression-table techniques; this is discussed in more detail in Section 5.2.2.

**Compression with narrow width data**

Narrow width data is common in many non-pointer scalar type fields. For instance, programmers often use a whole machine word to represent a single Boolean variable, even though a single bit suffices. The field `d_mounted` in structure `dentry` in Figure 5.1 is another example: it is defined to be of type `int`, which has value range \([-2^{31}, 2^{31} - 1]\) in a 32-bit machine. `d_mounted` is used to represent the number of file systems that are
mounted on one particular directory. Table 5.1 shows that the value range of $d_{\text{mounted}}$ is $[0, 1]$. This is because normally there is either no or at most one file system that is mounted on one directory. It is unlikely that there are $2^{31} - 1$ file systems mounted on a single directory.

A field with narrow width data is compressed by selecting the least possible bit width to represent the value range. In the above example, a single bit is enough to represent the value range of field $d_{\text{mounted}}$. Compression with narrow width data is similar to the use of bitfields in the C programming language. However, using bitfields usually involves manual declaration and is not seen often in typical programs. Also, bitfields can not be applied to pointer fields, which are discussed next.

**Compression with common prefix**

If a pointer points to a set of addresses that share a common prefix, the pointer can be compressed by eliminating the common prefix and only preserving the remaining bits as compressed value. To decompress, the common prefix of the pointer is simply added back to the compressed value. For example, consider the value range of field $d_{\text{op}}$ in Table 5.1. The minimum and maximum addresses of field $d_{\text{op}}$ share a 17-bit common prefix (0x08208000), which means all the addresses presented in $d_{\text{op}}$ during profiling also share 17 bit common prefix. Therefore the first 17 bits of this field can be “factored out,” and need not be represented explicitly with each address. Thus, we represent each value for this field using $32 - 17 = 15$ bits; the 17-bit prefix is stored separately. Whenever the field is used, the 17-bit prefix and 15-bit compressed representation are concatenated to obtain the original 32-bit representation.

**Compression with compression table**

The basic idea of this compression scheme is to keep the actual data value of a field in a table, which we call the *compression table*, and use the index into the table as the compressed value in the field. Let $V_f$ be the set of distinct values of a field $f$ from profiling. The number of bits required to represent the index for $V_f$ is defined as $\lceil \log_2|V_f| \rceil$. For
instance, in Table 5.1, the number of distinct values of $d_{op}$ is six, which means that only three bits are needed to represent the indexes for the compression table.

$d_{op}$ is a pointer to global data objects that describe how different file systems can overload the standard dentry operations. In our configured kernel, there are in fact seven such global objects, which can potentially be the target of $d_{op}$. Six of these that are present in the profiling data are highlighted in Figure 5.1. The missing one is pipefs_dentry_operations because the pipe file system was not used during our profiling run. However, it is interesting to see that, even if pipe file system is used for some execution run, the address of pipefs_dentry_operations can still be compressed because there are two unused slots in the compression table of $d_{op}$.

Using a compression table can achieve better compression results, in terms of saving bit width, than using a common prefix. However, maintaining the compression table may introduce considerable overhead at runtime, especially when the size of the compression table is large.

To limit the cost of using a compression table, our implementation limits the size of the table to at most 256 values; thus, a value needs at most eight bits for the index. When the size of the compression table is larger than this limit, the common-prefix scheme, described in Section 5.2.2 above, is used instead. For example, consider the profiling data of field $d_{inode}$ in Table 5.1. The number of distinct values (addresses) of $d_{inode}$ is 4096,\(^2\) which is larger than the limit. Therefore, compression with compression table is not applied to $d_{inode}$.

**Combining common prefix and compression table**

There are situations where the compression results can be further improved by combining the common-prefix and compression-table approaches. An example of such can be seen in the slab objects in the Linux kernel. The addresses of all slab objects inside of a

\(^2\)In fact, the total number of distinct values that appeared in $d_{inode}$ is larger than 4096. However, the table used for value profiling is set to hold a maximum of 4096 values. Note that even though the number of distinct values recorded saturates at 4096, this does not compromise soundness because in this case, saturation simply means compression-table scheme is not applied.
memory page only differ in the lower 12 bits—the page offset\(^3\). The top 20 bits of these addresses are therefore common—the page prefix. The page prefixes themselves can be further compressed by keeping them in a compression table and using the index into the table plus page offset as compressed value.

For example, consider field \(d\_parent\) in \(dentry\) structure. Field \(d\_parent\) is a pointer that points to objects in the \(dentry\) slab cache, as shown in Figure 5.1. The number of distinct page prefixes of \(d\_parent\) (shown in the last column in Table 5.1) is 99, which is less than our 256 value limit for compression table size. Therefore, field \(d\_parent\) can be compressed to use \(\lceil \log_2 99 \rceil = 7\) bits for the index and 12 bits for page offset, for a total of 19 bits. This is illustrated in Figure 5.1.

However, this compression scheme does not apply to field \(d\_inode\) since the number of distinct page prefixes is still too large.

### Choosing the compression scheme

To determine which compression scheme to use for each compressible field, we compute, for each compression scheme, the bit width that is required to represent the profiled data values.

Table 5.2 shows the bit widths of all the compression schemes computed for the compressible fields in \(dentry\) structure. For each compressible field, the compression scheme that achieves the smallest bit width is the one that is selected for that field (indicated with ‘\(\ast\)’). For field \(d\_mounted\), the first three compression schemes get the same

\(^3\)The size of a memory page in Linux is 4096 bytes, which requires \(\lceil \log_2 4096 \rceil = 12\) bits to represent.
bit width. In this case, compression with narrow width data is preferred because it does not need any extra space to store common prefix or compression table like the other two.

Once the compression scheme for a compressible field is decided, the representation for data values of that field is also determined. The values that can be accommodated within the chosen representation are said to be *compressible*, while values that cannot be accommodated within that representation are said to be *incompressible*. For example, suppose that a field $f$ for a structure is chosen to be represented using 6 bits. Then, the runtime value 58 for $f$ is compressible, but the value 68 is incompressible.

5.2.3 Compression and Decompression Procedures

The actual compression and decompression are performed by three procedures. Given one of the compression schemes, $S$,

- $test\_compress(S, v)$ (Figure 5.2 a) checks the compressibility of a given value $v$ with scheme $S$.

- $compress(S, v)$ (Figure 5.2 b) does the actual compression of given value $v$ with scheme $S$ and returns the compressed value.

- $decompress(S, v)$ (Figure 5.2 c) decompresses the given compressed value $v$ with scheme $S$ and returns the result.

Each type of compression scheme has several attributes, which contain necessary information for compression and decompression. The detailed descriptions of these attributes and corresponding examples are shown in Table 5.3. As we can see, scheme $CT$, i.e., compression table and scheme $CT+CP$, i.e., compression table and common prefix, share the same attributes. This is because $CT$ can be considered to be a special case of $CT+CP$, where the common prefix is the entire data value.

5.3 Data Structure and Source Code Transformation

After the compression scheme for each compressible field in a data structure has been determined, all the compressed fields are packed into an array $cdata$ that is just large
a) **Procedure** `test_compress(S, v)`

```plaintext
if S.type = NW /* narrow width */
    then
        if v ∉ [S.min, S.max] then
            return Fail
        end if
    else if S.type = CP then
        prefix ← v & S.prefix_mask
        if S.prefix = uninitialized then
            S.prefix ← prefix
        else if S.prefix ≠ prefix then
            return Fail
        end if
    else
        // compression-table + common prefix
        prefix ← v & S.prefix_mask
        if prefix ∉ S.compress_table then
            if S.compress_table is full then
                return Fail
            else
                insert prefix in S.compress_table
            end if
        end if
    end if
return Succ
```

b) **Procedure** `compress(S, v)`

```plaintext
// v' is the compressed representation of v
if S.type = NW /* narrow width */
    then
        v' ← v
    else if S.type = CP /* common prefix */
    then
        v' ← v & S.offset_mask
    else
        // compression-table + common prefix
        prefix ← v & S.prefix_mask
        idx ← index of prefix in S.compress_table
        v' ← (idx ◦ S.offset_bits) | (v & S.offset_mask)
end if
return v'
```

c) **Procedure** `decompress(S, v)`

```plaintext
// v' is the decompressed representation of v
if S.type = NW /* narrow width */
    then
        v' ← v
    else if S.type = CP /* common prefix */
    then
        v' ← S.prefix | v
    else
        // compression-table + common prefix
        idx ← extra index from v
        prefix ← S.compress_table[idx]
        v' ← S.prefix | v
end if
return v'
```

Note: The operators &, |, and ◦ denote bitwise-and, bitwise-or, and left-shift operations.
<table>
<thead>
<tr>
<th>Type</th>
<th>Attributes</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>NW</td>
<td>min and max</td>
<td>Value range</td>
<td>\texttt{d_mounted:{min=0, max=2}}</td>
</tr>
<tr>
<td>CP</td>
<td>prefix_mask</td>
<td>Bit mask used to extract common prefix from value</td>
<td>\texttt{d_inode:{prefix_mask = 0xfe000000, offset_mask = 0x1fffff}}</td>
</tr>
<tr>
<td></td>
<td>offset_mask</td>
<td>Bit mask used to extract offset from value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>prefix</td>
<td>The common prefix; initially set to \textit{uninitialized}</td>
<td></td>
</tr>
<tr>
<td>CT or</td>
<td>prefix_mask</td>
<td>Same as for CP</td>
<td>\texttt{d_op:{prefix_mask=0xffffffff, offset_mask=0x0, offset_bits=0, idx_mask=0x7}}</td>
</tr>
<tr>
<td>CT+CP</td>
<td>offset_mask</td>
<td>Same as for CP</td>
<td>\texttt{d_parent:{prefix_mask=0xffff0000, offset_mask=0xfff, offset_bits =12, idx_mask=0x7000 }}</td>
</tr>
<tr>
<td></td>
<td>offset_bits</td>
<td>Number of bits used for offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>idx_mask</td>
<td>Bit mask used to extract index from compressed value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>compress_table</td>
<td>Compression table; initially is empty</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Attributes of different compression schemes.

![Memory layout of compressed fields in cdata array and the compression access table.](image)

![Memory layout of cdata array and extra allocated space after expansion and the expansion access table.](image)

Figure 5.3: Memory layout of compressed data structure before and after expansion.
enough to accommodate the total number of bits in the compressed data structure. In the example of dentry structure in Figure 5.1, the corresponding compressed structure type is defined as

```c
struct dentry {
    ...
    char cdata[N];
};
```

We use `char` as the type of array `cdata` because `char` is the smallest data type in C.

Figure 5.3 a) shows the memory layout of `cdata` array in the compressed dentry structure. We use one bit in each `cdata` array to indicate whether it contains compressed data (i.e., all the values encountered during execution, for fields within that structure, were compressible) or whether `cdata` contains a forwarding pointer to an uncompressed representation of that structure (i.e., some field value was incompressible). This is done using the lowest bit in the first byte of `cdata`, which we call the `compress bit`. Initially, this bit is set to 1, which means `cdata` contains compressed values. Later, if an incompressible value is encountered for any compressed field of that structure, this bit is set to 0 and the first word of `cdata` is set to point to an uncompressed representation of the structure. This is discussed in more detail in Section 5.3.1.

All the compressed fields are packed into the remaining space in `cdata`. The size of array `cdata` is equal to \(\lceil \sum \frac{\text{Bits}_f + 1}{8} \rceil\), where `Bits_f` is the bit width of a compressed field `f`. For example, the size of `cdata` for compressed dentry structure is \(\lceil \frac{1+3+25+19+1}{8} \rceil = 7\).

5.3.1 Compression and Expansion

For a compressed data structure, two tables are created automatically. The first table, `compress access table`, stores the information about how to handle compressed values for each compressed field. The second table, `expansion access table`, contains information about how to handle a value once an instance of a compressed data structure is expanded to store an incompressible value.

Given an instance, `A`, of a compressed data structure, the compressed access table is
used to access \textit{cdata} as long as all the data values in A are compressible. Consider the compress access table shown in Figure 5.3 a). \textit{fid} is a unique id assigned to each compressed field for fast look-up into the table. \textit{start} is the bit location where a compressed field starts in \textit{cdata} and \textit{bits} is the bit width of a compressed field. Lastly, \textit{scheme} is the compression scheme of a compressed field as we discussed in Section 5.2.2.

If an incompressible value is encountered during execution, all the compressed data in \textit{A.cdata} are expanded to their original size according to the expansion access table. Consider the example shown in Figure 5.3 b), which illustrates how an instance of compressed \textit{dentry} structure is expanded. First, extra space is allocated to hold decompressed values. The first four bytes of \textit{cdata} are used to store the address of the extra allocated space. The remaining space in \textit{cdata} is reused as much as possible, but in the example of \textit{dentry}, the remaining three bytes are left unused. The compressed values stored in \textit{cdata} are decompressed and stored in new locations according to the expansion access table. By assigning the address of extra space to the first four bytes in \textit{cdata}, the compress bit in \textit{cdata} is set to 0 as well.\footnote{This assumes that dynamic memory allocation routines such as \textit{malloc} return addresses that are at least even-address aligned (common implementations of \textit{malloc} satisfy this requirement, e.g., the GNU C library returns blocks that are at least 8-byte aligned).}

Table 5.4 lists the set of procedures that are used to access data in the compressed space and the expanded space of an instance of a compressed data structure. Once an incompressible value is encountered, all the compressed fields in a compressed instance are expanded. Because all compressed fields are expanded at once, there is at most one expansion operation for each instance and there is only one pointer needed to keep the addresses of extra space. It is possible to expand compressed fields separately, but that can lead to the multiple allocations of extra space (which is expensive) and also require multiple pointers (consuming more space).

Once an instance of a compressed data structure is expanded, it is always maintained in uncompressed form. This is to simplify the maintainance of an expanded instance, and also to avoid repeating expansion and converting back to compressed form.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bit_write(cdata, start, n, v)</code></td>
<td>Writes up the lower ( n ) bits of ( v ) to array ( cdata ) starting at bit location ( start ).</td>
</tr>
<tr>
<td><code>bit_read(cdata, start, n)</code></td>
<td>Reads ( n ) bits of data from array ( cdata ) starting from location bit ( start ) and return the data.</td>
</tr>
<tr>
<td><code>expand_write(cdata, E_{access}, v)</code></td>
<td>if ( E_{access}.extra = True ) then () ( extra = bit_read(cdata, 0, 32) ); get address of extra space () ( bit_write(extra, E_{access}.start, E_{access}.bits, v); ) () else () ( bit_write(cdata, E_{access}.start, E_{access}.bits, v); ) endif</td>
</tr>
<tr>
<td><code>expand_read(cdata, E_{access})</code></td>
<td>if ( E_{access}.extra = True ) then () ( extra = bit_read(cdata, 0, 32) ); get address of extra space () ( return bit_write(extra, E_{access}.start, E_{access}.bits); ) () else () ( return bit_write(cdata, E_{access}.start, E_{access}.bits); ) endif</td>
</tr>
<tr>
<td><code>expand_space(cdata)</code></td>
<td>copy ( cdata ) into buffer () ( extra \leftarrow allocate extra space. ) ( bit_write(cdata, 0, 32, extra); ) set extra pointer () for each ( fid ) of compressed data in ( buffer ) do () ( C_{access} \leftarrow CompressedAccessTable[fid]; E_{access} \leftarrow ExpAccessTable[fid]; ) ( data \leftarrow bit_read(buffer, C_{access}.start, C_{access}.bits); ) ( v \leftarrow decompress(data, C_{access}.scheme); ) ( expand_write(cdata, E_{access}, v); ) endfor</td>
</tr>
</tbody>
</table>

Table 5.4: Procedures for accessing data in compressed space and expanded space.
5.3.2 Source Code Transformation

Table 5.5 Shows the rewriting rules to transform source code to support dynamic data structure compression. In the rewriting rules, $T$ is the original data structure type and $T'$ is the compressed data structure type. Once an instance of $T'$ is allocated, the compress bit in the cdata of this instance needs to be set to 1 first. Once an instance of $T'$ is freed, the extra allocated space is also freed if the instance is expanded. A statement that accesses a compressed field in $T'$ is transformed to corresponding compression/decompression function calls. If a statement stores a new value into a compressed field in $T'$, the statement is transformed to a function call to compress_store and if a statement is to load a value from a compressed field in $T'$, it is transformed to a function call to compress_load.

5.3.3 Soundness Considerations

Data structure compression changes the way in which structure fields are represented and accessed. To preserve safety, we have to make sure that such changes do not affect the observable behavior of a program. Intuitively, the requirement for this is that a field $f$ of a structure $S$ is considered for compression only if the only way in which $f$ can be accessed in the program is via expressions of the form ‘$S.f$’ or ‘$p->f$’, where $p$ is a pointer to $S$. This property ensures that we can use type information to ensure that all accesses to a compressed field have the appropriate compression/decompression code added to them.

We enforce this requirement as follows: a field $f$ of a structure $S$ is excluded from compression if any of the following hold: (1) the address of $f$ is taken, e.g., via an expression of the form &$(S.f)$; (2) a pointer $p$ to the structure $S$ or to the field $f$ is cast to some other type (in either case, it would be possible to bypass the compression/decompression code when accessing $f$); or (3) an offset is used to access a field within $S$, e.g., $(&S) + 4$. These restrictions exclude from compression any field of a structure that can have a pointer to it. This is important because the process of comparison can change the relative order of fields that are compressed and fields that are not compressed: the former get pulled into the cdata array, the latter do not. The conditions given above ensure that code that may be sensitive to the layout of fields within the structure are precluded from compression.
<table>
<thead>
<tr>
<th>Original statement</th>
<th>Transformed statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T * p = \text{alloc}(T)$</td>
<td>$T' * p = \text{alloc}(T')$</td>
</tr>
<tr>
<td></td>
<td>$p-&gt;\text{cdata}[0] = 0x1$ // initialize cdata</td>
</tr>
<tr>
<td>\texttt{free}(p)</td>
<td>if $(p-&gt;\text{cdata}[0] &amp; 0x1)$ // test compress bit</td>
</tr>
<tr>
<td></td>
<td>\texttt{extra = bit\textunderscore read}(p-&gt;\text{cdata}, 0, 32)</td>
</tr>
<tr>
<td></td>
<td>\texttt{free(extra)}</td>
</tr>
<tr>
<td></td>
<td>\texttt{endif}</td>
</tr>
<tr>
<td></td>
<td>\texttt{free(p)}</td>
</tr>
<tr>
<td>$p-&gt;\text{field} = v$</td>
<td>\texttt{compress_store}(p-&gt;\text{cdata}, \textit{fid}, v)</td>
</tr>
<tr>
<td></td>
<td>$\texttt{cdata} \leftarrow p-&gt;\text{cdata}$</td>
</tr>
<tr>
<td></td>
<td>$C \leftarrow \text{CompAccessTable}[\textit{fid}]$</td>
</tr>
<tr>
<td></td>
<td>$E \leftarrow \text{ExpAccessTable}[\textit{fid}]$</td>
</tr>
<tr>
<td></td>
<td>if $(\text{cdata}[0] &amp; 0x1)$ // test compress bit</td>
</tr>
<tr>
<td></td>
<td>if \texttt{!test_compress}(C.scheme, v)</td>
</tr>
<tr>
<td></td>
<td>\texttt{expand_space}(\textit{cdata}, E)</td>
</tr>
<tr>
<td></td>
<td>\texttt{else}</td>
</tr>
<tr>
<td></td>
<td>$v \leftarrow \text{compress}(C.scheme, v)$</td>
</tr>
<tr>
<td></td>
<td>\texttt{bit_write}(\textit{cdata}, C.start, C.bits, v)</td>
</tr>
<tr>
<td></td>
<td>\texttt{return}</td>
</tr>
<tr>
<td></td>
<td>\texttt{endif}</td>
</tr>
<tr>
<td></td>
<td>\texttt{endif}</td>
</tr>
<tr>
<td></td>
<td>\texttt{expand_write}(\textit{cdata}, E, v)</td>
</tr>
<tr>
<td>$v = p-&gt;\text{field}$</td>
<td>\texttt{compress_load}(p-&gt;\text{cdata}, \textit{fid})</td>
</tr>
<tr>
<td></td>
<td>$\texttt{cdata} \leftarrow p-&gt;\text{cdata}$</td>
</tr>
<tr>
<td></td>
<td>$C \leftarrow \text{CompAccessTable}[\textit{fid}]$</td>
</tr>
<tr>
<td></td>
<td>$E \leftarrow \text{ExpAccessTable}[\textit{fid}]$</td>
</tr>
<tr>
<td></td>
<td>if $(\text{cdata}[0] &amp; 0x1)$ // test compress bit</td>
</tr>
<tr>
<td></td>
<td>$v\leftarrow \texttt{bit_read}(\textit{cdata}, \text{C.start}, \text{C.bits})$</td>
</tr>
<tr>
<td></td>
<td>\texttt{return} \texttt{decompress}(C.scheme, v)</td>
</tr>
<tr>
<td></td>
<td>\texttt{else}</td>
</tr>
<tr>
<td></td>
<td>\texttt{return} \texttt{expand_read}(\textit{cdata}, E)</td>
</tr>
</tbody>
</table>

Table 5.5: Rewrite rules for dynamic data structure compression
### 5.4 Experimental Evaluation

We evaluated our ideas using the Linux kernel version 2.6.19. In order to emulate an embedded system environment, the experiments were conducted on an old laptop machine with Intel Pentium III 667MHZ processor and 128MB of memory. The data structure profiling is done by modifying GCC (4.2.1) to insert profiling code for every statement that contains field referencing expression in a program. The source code transformations are done manually at present. However, it is possible to automate the process using a source-to-source transformation tool, such as CIL \cite{CIL}, by using the rewriting rules given in Table 5.5.

#### 5.4.1 Select the Dynamic Data Structures to Compress

In our current implementation, we compress a subset of the slab caches in the Linux slab allocator (recall that, as discussed in Section 5.1, this forms the core of dynamic memory management in the Linux kernel). The compressed slab caches are listed in Table 5.6. Column 1 gives the names of compressed slab caches; column 2 is the data structure type used by each slab cache; column 3 shows the size of each slab cache based on profiling; and column 4 is the ratio of the size of each slab cache to the total memory space in the slab allocator. Overall, the slab caches in Table 5.6 account for over 81% of all memory space used by the Linux slab allocator. We ignore the remaining slab caches for two reasons: first, most of the remaining slab caches consume only small amount of memory even though they can be compressed; second, there are several slab

<table>
<thead>
<tr>
<th>Cache Name</th>
<th>Object type</th>
<th>Size(KB)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext2_inode_cache</td>
<td>ext2_inode_info</td>
<td>1034</td>
<td>46.8%</td>
</tr>
<tr>
<td>dentry_cache</td>
<td>dentry</td>
<td>450</td>
<td>20.4%</td>
</tr>
<tr>
<td>proc_inode_cache</td>
<td>proc_inode</td>
<td>137</td>
<td>6.2%</td>
</tr>
<tr>
<td>buffer_head</td>
<td>buffer_head</td>
<td>67</td>
<td>3.0%</td>
</tr>
<tr>
<td>inode_cache</td>
<td>inode</td>
<td>51</td>
<td>2.3%</td>
</tr>
<tr>
<td>sysfs_dir_cache</td>
<td>sysfsDirent</td>
<td>37</td>
<td>1.8%</td>
</tr>
<tr>
<td>bio</td>
<td>bio</td>
<td>18</td>
<td>0.8%</td>
</tr>
</tbody>
</table>

Table 5.6: Compressed slab caches (sorted by cache size in non-increasing order)
caches (account for about 14% of the total memory space) that can not be compressed with our current implementation, e.g., we currently do not compress array fields, and the main data field in the slab cache `radix_tree_node`, which is also the largest one that we do not compress, is an array.

Data structure compression can save memory space but it can also bring cost—compression/decompression of a compressed field are much more expensive than the store/load operations of an uncompressed field. Based on the profile information, all the fields can be classified into two categories: hot fields, which are the fields used frequently, and cold fields, which are the fields used infrequently. Conceptually, we should not compress hot fields because it may cause significant amount of overheads even though more memory space can be saved. To control this cost-benefit tradeoff, we use a user-specified threshold $r \in [0.0, 1.0]$ to determine the fraction of compressible fields (i.e., hot fields) that should not be compressed.

Let $cost$ be the number of load/store of a compressible field based on profiling. Let $C$ be the total number of load/store of all compressible fields in a program.\(^5\) Given a value of $r$, we consider all the compressible fields $f$ in the program in decreasing order of $cost(f)$ and determine the smallest value of $N$ such that

$$\sum_{f : cost(f) > N} cost(f) \leq C \cdot r.$$  

Any compressible field whose number of load/store is larger than $N$ is avoided from compression. For example, $r=0.0$ means all compressible fields are compressed; and $r=1.0$ means nothing is compressed.

5.4.2 Experimental Results

We used two sets of benchmarks to evaluate the runtime impacts of our approach: (1) a set of kernel-intensive benchmarks: `find`, which runs the `find` command at the root directory to scan all files in the file system, `copy.small`, which makes a copy of 5MB

\(^5\)In case of our experiment in the Linux kernel, all compressible fields are defined as the compressible fields in the data structures in Table 5.6.
file and *copy.large*, which makes a copy of 20MB file; and (2) a collection of eight application programs from the MediaBench suite [43], used for evaluating multimedia and communications systems. These two sets of benchmarks were tested separately while all programs in each set were executed sequentially (for instance, the execution sequence of kernel-intensive benchmarks is *find*, *copy.small*, *copy.large*).

Table 5.7 shows the average percentage of memory reduction in Linux kernel slab allocator and average percentage of performance overheads for kernel-intensive benchmarks and MediaBench benchmarks. The values of $r$ considered in our experiments are \{0.0, 0.2, 0.4, 0.6, 0.8\}. As we can see in Table 5.7, both the average memory reduction and average performance overheads decrease when the value of $r$ increases. When $r=0.0$, there is about 18% memory reduction for both benchmarks while the overhead of MediaBench (1.9%) is much lower than the overhead of kernel-intensive benchmarks (7.3%). However, even for kernel-intensive benchmarks, the overhead reduces to only 1.1% when $r=0.6$ and there is still about 14% of memory reduction.

The results of memory reduction percentage and performance overhead percentage for each program are shown in Figure 5.4 a) and b) respectively. The detailed experimental results are shown in Table A.4 and Table A.5 in Section A.3.

Generally, there are more overheads for kernel-intensive benchmarks than MediaBench benchmarks. *copy.large* has the largest overhead (over 15%) among all programs when $r=0.0$. But its overhead reduces to below 5% when $r \geq 6$. The results shown in Figure 5.4 a) also include the extra allocated space from expansion and the size of extra allocated space is relative small (less than a page (4KB)) for both sets of benchmarks.
a) Memory reduction percentage of Linux slab allocator.

b) Performance overhead percentage of benchmark programs.

keys: $r=0.0$, $r=0.2$, $r=0.4$, $r=0.6$, $r=0.8$

Figure 5.4: Runtime impacts of dynamic data structures compression

Table 5.8 shows the static impacts of our approach on size reduction of compressed kernel data structure and kernel code size. On average, there is 28.7% size reduction of all the compressed data structures in the Linux kernel when $r=0.0$. The number drops to 21.5% when $r=0.8$. The increase of code size caused by code transformation is about 5.4% when $r=0.0$ and the increase reduces to about 2.5% when $r=0.8$.

5.4.3 Discussion

The reason for low overall performance overhead in the Linux kernel is that, even though data structure compression causes a performance overhead within the Linux kernel, the
Table 5.8: Static impacts on data structure size and code size.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>r</th>
<th>0.0</th>
<th>0.2</th>
<th>0.4</th>
<th>0.6</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. data structure size reduction</td>
<td></td>
<td>28.7%</td>
<td>27.0%</td>
<td>25.9%</td>
<td>24.6%</td>
<td>21.5%</td>
</tr>
<tr>
<td>Increase of code size</td>
<td></td>
<td>5.4%</td>
<td>4.9%</td>
<td>3.9%</td>
<td>3.6%</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

actual time spent within the kernel is only a small portion of the total running time of the whole system. For programs in MediaBench, most of the running time is spent in the user processes. Even for the kernel-intensive benchmarks, a significant amount time is spent in data copying between the kernel and the user applications. In order to evaluate how data structure compression can affect computation-intensive applications, we applied our approach to several benchmarks in the Olden test suite [11], which consists of a set of pointer intensive programs that make extensive use of dynamic data structures.

When compression is applied to all fields regardless of how heavily they are used, we see significant performance degradation. The average reduction in memory consumption ranges from a little over 30%, for small inputs, to about 23% for large inputs. This is accompanied by slowdowns ranging, on average, from about 183% for small inputs to 194% for large inputs. When the most frequently accessed field is excluded from compression, there is a drop in the amount of memory usage reduction obtained, as one would expect: about 20% on average for small inputs and 6% for large inputs. However, because the remaining fields that are compressed are still accessed quite frequently, this still incurs significant runtime overhead, averaging about 88% for small inputs and 116% on large inputs. Interestingly, in the second case one of the benchmarks consumed more space with data compression than the original version: the reason for this is that the set of values encountered in the profiling data (small inputs) did not reflect the range of values encountered in the large inputs, leading to many compressed representations having to be expanded at runtime, thereby incurring an extra cost of 4 bytes per expanded representation to hold a pointer to the expanded representation.

The performance results for the Olden benchmarks illustrate that data structure compression can lead to significant performance overheads if applied to heavily used data. In

---

6The Olden benchmarks we considered are perimeter, treeadd, and tsp.
order to be a good candidate for data structure compression, a program should use plenty of compressible data (to obtain significant memory size reductions) which are used relatively infrequently (to keep the runtime performance overhead low). An OS kernel on embedded systems typically meets these requirements. Our experiments with the Linux kernel bear this out, with good reductions in dynamic memory usage with only a very small performance overhead.

5.5 Summary

This chapter describes an approach that reduces the memory requirements of the dynamic data structures within an OS kernel. We use value profiling information to transform the OS kernel code to reduce the size of dynamic kernel data structures. Experimental results show that, on average, our approach reduces the memory consumption of the slab allocators in the Linux kernel by about 17.5% when running the MediaBench suite, while incurring only minimal increases in code size (5.4%) and execution time (1.9%).
6.1 Program Size Reduction

There are typically two different ways to reduce the program size—compaction-based techniques and compression-based techniques. Compaction-based techniques generate a smaller program, which can be executed directly; while compression-based techniques transform the original representation of a program into a compressed form, which usually requires a decompression step before the compressed code can be executed.

6.1.1 Compaction-based Techniques

The most commonly used code compaction techniques are compiler optimizations for reducing program size. Although most compiler optimizations are meant to speed up program execution, some of the compiler optimizations, such as redundant-code elimination, dead code elimination, and unreachable code elimination, also have the effects of reducing program code size. For example, the GCC compiler provides a command-line parameter `-Os`, which enables optimizations for size, but not at the expense of speed. It is a difficult task for compiler writer to determine the best compile sequences due to the interactions between different optimizations. Cooper et al. [15] applied a genetic algorithm to find a good compiler optimization sequences that generate small program object code. Their results showed that the ideal optimization sequences differ for each individual program.

Researchers have also developed compaction techniques that are dedicated to reduce the program size. Optimizations, such as code factoring, eliminate repeated code sequences in a program using procedure abstraction to yield smaller executable. Early work by Fraser et al. [29], by Baker [7], by Cooper and McIntosh [14], and by Zastre [63]
treated an executable program as a simple linear sequence of instructions and used pro-
cedure abstraction to extract repeated instruction sequences into separate functions. The
size reductions they reported were modest, averaging about 4-7%.

Debray et al. [19] described a code compaction approach based on “whole-program”
analysis. Their approach used aggressive inter-procedural optimization together with pro-
cedure abstraction of repeated code fragments to achieve significant reduction in code
size. A binary-rewriting tool called SQUEEZE was implemented based on the idea. Ap-
p lied to SEPC-95 integer benchmarks, their technique managed to reduce the size of
executable by about 30% on the average.

Lau et al. [42] proposed a compaction approach using echo instructions. With echo
instructions, similar but not necessarily identical section of code can be reduced to a
single copy of the repeated code. Their results revealed that an additional 10% code size
reduction can be achieved with the use of echo instructions in SQUEEZE.

Many of modern-day embedded processors use RISC technology (e.g., ARM and
MIPS). RISC architectures generally have fixed-size instructions (usually 32 bits) that
only encode relatively simple operations. As a consequence, a RISC processor typically
needs more memory than a CISC does to store the same program. To alleviate the burden
of memory cost, a compact instruction set (e.g., Thumb and MIPS-16) is introduced as
a subset of standard RISC instruction set. Translating ARM code to Thumb instructions
on average reduces the code size with 30% but the resulting code will execute on average
40% slower [30]. Krishnaswamy and Gupta [40] proposed technique that generates mixed
ARM and Thumb code for application programs to achieve significant code size reduction
without loss in performance.

6.1.2 Compression-based Techniques

There is a rich body of work on code compression to reduce program size by exploring
the compressibility of a wide range of program representations: source code, intermediate
representations, machine code, etc [62]. Compression-based techniques generally focus
on compressing the executable as much as possible. However the drawback is that the
compressed representation of the original program must either be decompressed [23, 27,
Evans and Fraser [24] proposed a method for producing compact and interpretable bytecode by applying grammar rewriting. Based on a set of representative bytecoded programs used as the training inputs, their system searches for repeated patterns in the parse trees of initial grammar and creates an expanded grammar. This expanded grammar still accepts the same language as the initial grammar but permits a more succinct derivation (compressed bytecode) for a sampled program.

Debray and Evans [18] described a method that only compresses the infrequently executed code of a program and divides compressed code into pieces so that each piece of compressed code is decompressed only when it is needed.

Compression-based technique has also been developed to reduce heap memory usage of application programs. Zhang and Gupta [66] use a hardware-based scheme to reduce the memory usage of dynamic data by compressing a 32-bit integer and a 32-bit pointer into 15-bit values, which then are packed into a single 32-bit field. Since their approach compresses each field in a uniform way, it can potentially cause two problems: first, space is wasted if compressible fields require few bits than 15; and second, more expansions happen at runtime if compressible fields require more bits than 15. Also, it requires specialized hardware to improve performance.

The introduction of 64-bit architectures has led a number of researchers to investigate the problem of compressing pointers into 32 bits [55, 59, 41]. Our work on dynamic data structure compression in Chapter 5 differs from these in that we can compress both pointer data and non-pointer scalar data, so a wider set of applications can be benefit from our approach.

6.2 Operating Systems Optimization and Specialization

The work closely related to this dissertation is that on optimization/specialization of operating systems kernels [45, 52, 53]. These generally focus on improving execution speed rather than reducing size of memory footprint and therefore use techniques very different
The Synthesis kernel [53] introduced the idea of dynamically generating specialized code for system calls within a customized kernel. A follow-on project, Synthetix [52], extends the ideas in Synthesis with incremental and optimistic specialization. Incremental specialization allows specialized code modules to be generated as information becomes available. Optimistic specialization allows specialized modules to depend on system states that are likely to occur but not certain.

Flower et al. [26] described the use of Spike, a binary optimizer for Compaq Alpha, to optimize the Unix kernel. Their work focused in particular on profile-guided code layout but also conducted optimization such as unreachable code elimination.

The work that is closest to our work on OS kernel code compaction in Chapter 3 is that of Chanet et al., who describe a system for code compaction of the Linux kernel to reduce its memory footprint [13, 12]. The overall compaction results achieved by the two approaches are quite similar: on the one data point where the two systems can be directly compared (BusyBox with networking, identified with ‘†’ in Table A.1), Chanet et al. do somewhat better than us (18.9% reduction in text size, compared to the 18.0% we get). The techniques used by the two systems are quite different, however, and a detailed comparison of the two indicates different strengths for each. For example, in the context of unreachable code elimination, Chanet et al. use more detailed structural information about subsections in the input binary to do a better job of analyzing pointers between the code and data sections; our system, on the other hand, uses the source-level FA analysis to obtain more precise information about indirect call targets. Subsequent work by Chanet et al. obtains additional code size reductions by compressing infrequently executed code [13]. These techniques use Huffman compression to store infrequently executed code in a non-executable format and rely on runtime decompression to restore them to an executable format when necessary [18].

6.3 Memory Footprint Reduction for Embedded Systems

Recently, some researchers have begun exploring the use of overlays out of flash memory to reduce memory requirements in embedded systems. Park et al. describe a scheme for
generating dynamic code overlays for programs that can be modeled using synchronous data flow, which makes it possible to determine a static schedule for the program’s code [50]. Park et al. describe an application-specific demand paging mechanism for low-end embedded systems that do not have virtual memory [49]. Both works limit their focus on application programs and do not address the numerous issues peculiar to operating system kernels that arise in this dissertation.

Egger et al. describe dynamic code placement techniques and memory management strategies for scratchpad memory in embedded systems [21] [22]. Their interest focuses on improving the overall performance of system instead of reducing the memory requirement. They also apply their work only on application programs.

Cooprider and Regehr [16] apply static whole-program analysis to reduce a program’s data size including statically allocated scalars, pointers, structures, and arrays. However, their work targets small systems on microcontrollers that do not support dynamic memory allocation.
CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

7.1 Conclusions

As the complexity of embedded systems grows, there is an increasing use of general-purpose operating systems in embedded devices. Despite their convenience and flexibility, such operating systems can be overly general and contain features and code that are not needed in every application context, which incurs unnecessary performance overheads. In most embedded systems, resources such as processor processing, available memory, and power consumption, are strictly constrained. Even though memory costs have been dropping steadily due to advances in technology, there has been a concomitant growth in expectations of sophistication and functionality provided by embedded systems. This, together with the popular usage of operating systems in embedded devices, makes it important to reduce the memory footprint of operating systems for embedded systems. This dissertation tackles this problem by developing three techniques—code compaction, on-demand code loading, and dynamic data structure compression.

7.1.1 Static Code Compaction

Embedded systems tend to have relatively static configurations—both at the hardware end and at the software end. This implies that an embedded system will typically use only some of the functionality offered by a general-purpose operating system.

We begin with the observation that embedded systems typically run a small fixed set of applications. The code corresponding to the unused functionality is unnecessary overhead, and should be removed. This knowledge can be used to identify the minimal functionality required by the kernel code to support those applications and then to discard unnecessary code. We discuss a number of technical challenges that have to be addressed
in order to make this work; in particular, we describe a novel “approximate decompilation” technique, which allows us to apply source-level program analyses to hand-written assembly code. Our ideas have been implemented in a prototype binary rewriting tool that is able to achieve a code size reduction of close to 24% on an already minimally-configured Linux kernel.

7.1.2 On-Demand Code Loading

Even though code compaction can remove a portion of the entire OS kernel code, when exercised with typical embedded benchmarks, such as MiBench, most kernel code is executed infrequently if at all. While embedded systems typically have a limited amount of main memory, they often have a considerably greater amount of secondary storage available (e.g., flash memory). Based on these observations, we developed an automatic approach that keeps the rarely used code on secondary storage, and load it into main memory only when it is needed. In order to minimize the overhead of code loading, a greedy node-coalescing algorithm is proposed to group closely related code together. The experimental results show that this approach can reduce memory requirements for the Linux kernel code by about 53% with little degradation in performance.

7.1.3 Dynamic Data Structure Compression

Code compaction and on-demand code loading, have focused on reducing the code size of an OS kernel. Just as important are dynamic data, namely, the execution stack and the heap memory, which also consume a significant amount of memory. In practice, there is quite often room to reduce the memory footprint of dynamic data in an OS kernel using compression. For example, integer-value variables often do not require the full 32 bits allocated to them; sets of pointer values share redundant common prefix. We use value profiling information to transform the OS kernel code to reduce the size of dynamic kernel data structures. The technique is safe: if a runtime value is beyond the profile range, this approach automatically expand the compressed representation to its original size. Experimental results show that, on average, our approach reduces the memory consumption
of the slab allocators in the Linux kernel by about 17.5% when running the MediaBench suite, while incurring only minimal increases in code size (5.4%) and execution time (1.9%).

7.2 Future Directions

**Whole-software-system optimization** Operating system is just one, although an important one, of the components in a complete software system. A complete software system also consists of other components, such as user applications and libraries. Currently, most optimizations, such as code compaction, are usually done within individual software component and independent from others. It can be ideal to apply optimizations to the whole software systems so that all software components can be integrated in a more efficient way. For example, data transmission between the user applications and operating systems costs considerable overheads. Such overheads can potentially be mitigated by merging the spaces between user applications and operating systems. Not only embedded systems can benefit from whole-system optimization. Whole-system optimizations can also make other systems, such as newly emerging virtual appliances [3], more compact and more efficient.

**Hardware-aware optimization and specialization** Our approaches for operating systems specialization use high-level information from the user applications but they do not fully utilize the low-level information from hardware specifications. It turns out that, because of the end of frequency scaling as the dominant cause of processor performance, an increasing popular way to provide better performance is using special-purpose hardware (e.g., graphic processing unit). However, using special-purpose hardware/processors imposes a great challenges on software development—lacking support from compilers and libraries for new hardware. A possible direction is to incorporate learning and reasoning method to develop efficient approaches to drive program optimizations for special-purposed hardware.
BENCHMARK DATA

A.1 Experimental Results of Code Compaction

Table A.1 shows the effects of code compaction as described in Chapter 3. For each benchmark set, we present three sets of numbers; these give the amount of compaction achieved for the .text.init section (the code used for kernel bootup), the .text section (the kernel code used during steady-state execution), and the total amount of code (.text.init and .text together).
<table>
<thead>
<tr>
<th>Kernel config.</th>
<th>Application set</th>
<th>text.init section</th>
<th>text section</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size (Kb)</td>
<td>reduction (%)</td>
<td>size (Kb)</td>
<td>reduction (%)</td>
</tr>
<tr>
<td>Original kernel</td>
<td>48.54</td>
<td>–</td>
<td>788.16</td>
<td>–</td>
</tr>
<tr>
<td>All system calls</td>
<td>37.15</td>
<td>23.46</td>
<td>698.97</td>
<td>11.32</td>
</tr>
<tr>
<td>Busybox</td>
<td>37.15</td>
<td>23.46</td>
<td>648.95</td>
<td>17.06</td>
</tr>
<tr>
<td>Automotive</td>
<td>37.15</td>
<td>23.46</td>
<td>633.64</td>
<td>19.60</td>
</tr>
<tr>
<td>Cellphone</td>
<td>37.15</td>
<td>23.46</td>
<td>637.71</td>
<td>19.09</td>
</tr>
<tr>
<td>Consumer</td>
<td>37.15</td>
<td>23.46</td>
<td>639.48</td>
<td>18.86</td>
</tr>
<tr>
<td>Entertainment</td>
<td>37.15</td>
<td>23.46</td>
<td>636.12</td>
<td>19.29</td>
</tr>
<tr>
<td>Network</td>
<td>37.15</td>
<td>23.46</td>
<td>633.74</td>
<td>19.59</td>
</tr>
<tr>
<td>Office</td>
<td>37.15</td>
<td>23.46</td>
<td>640.39</td>
<td>18.75</td>
</tr>
<tr>
<td>Security</td>
<td>37.15</td>
<td>23.46</td>
<td>638.77</td>
<td>18.95</td>
</tr>
<tr>
<td>Telecom</td>
<td>37.15</td>
<td>23.46</td>
<td>634.23</td>
<td>19.33</td>
</tr>
<tr>
<td>Geom. mean:</td>
<td>23.46</td>
<td>–</td>
<td>19.03</td>
<td>–</td>
</tr>
</tbody>
</table>

† Corresponds to the experiments of Chanet et al. [12].

Table A.1: Code compaction results
A.2 Experimental Results of On-Demand Code Loading

Table A.2 shows the behavior of our clustering algorithm and the memory reduction results for the different benchmarks as described in Chapter 4.

The first column indicates the benchmark suite being considered. The second column gives the core code size bound $\gamma$ indicating how much the core code is allowed to grow in size. The third column gives the number of clusters formed. The fourth and fifth columns give the average and maximum cluster size, respectively. The sixth column gives the total memory size, computed as the sum of the sizes of the memory-resident code, the code buffer and the memory allocated by overlay manager, which includes the restore stubs (600 bytes) and the cluster address table ($= \text{no. of clusters } \times 4 \text{ bytes}$). The size of the memory-resident code is obtained as the size of the .text section in the kernel with on-demand code loading.\(^1\) The final column gives the percentage reduction size, measured relative to the size of the original kernel (Recall that the MiBench and MediaBench programs were evaluated on a kernel without networking support, with original size 590,022 bytes, while the httpd benchmark was evaluated on a kernel with networking support, with original size 890,793 bytes).

It can be seen from Table A.2 that, as expected, increasing the value of the code size bound $\gamma$ leads to a decrease in the total number of clusters. The average cluster size remains almost the same, while the maximum cluster size varies for all different $\gamma$.

Table A.3 shows the effect of different core code growth bounds on the runtime cost of on-demand code loading. We show data for two different costs: the first set of data (columns 3–5) show the cost of booting the kernel and starting a shell (for running httpd, the booting process also includes starting network and httpd server); while the second set (columns 6–8) show the kernel-level cost of running the benchmark applications. Columns 3 and 6 give the total number of accesses for code loading while columns 4 and 7 give the total amount of code loaded into the code buffer.

---
\(^1\)There is some code in the .init.text section used during the kernel bootup process, but we did not include this in our size computations because this section is deallocated, and its memory freed up, after the initial portion of booting.
<table>
<thead>
<tr>
<th>Core code size bound $\gamma$ (%)</th>
<th>No. of Clusters</th>
<th>Ave. cluster size (bytes)</th>
<th>Max. cluster size (bytes)</th>
<th>Total memory size (bytes)</th>
<th>Memory size reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MiBench</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>250</td>
<td>1634</td>
<td>2023</td>
<td>255,566</td>
<td>56.7</td>
</tr>
<tr>
<td>2</td>
<td>247</td>
<td>1635</td>
<td>2041</td>
<td>260,068</td>
<td>55.9</td>
</tr>
<tr>
<td>4</td>
<td>244</td>
<td>1634</td>
<td>2025</td>
<td>264,639</td>
<td>55.1</td>
</tr>
<tr>
<td>6</td>
<td>241</td>
<td>1635</td>
<td>2032</td>
<td>269,288</td>
<td>54.4</td>
</tr>
<tr>
<td>8</td>
<td>238</td>
<td>1636</td>
<td>2039</td>
<td>273,748</td>
<td>53.6</td>
</tr>
<tr>
<td>10</td>
<td>235</td>
<td>1638</td>
<td>2045</td>
<td>277,988</td>
<td>52.9</td>
</tr>
<tr>
<td><strong>MediaBench</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>250</td>
<td>1639</td>
<td>2045</td>
<td>254,573</td>
<td>56.9</td>
</tr>
<tr>
<td>2</td>
<td>248</td>
<td>1632</td>
<td>2025</td>
<td>259,041</td>
<td>56.1</td>
</tr>
<tr>
<td>4</td>
<td>245</td>
<td>1633</td>
<td>2022</td>
<td>263,629</td>
<td>55.3</td>
</tr>
<tr>
<td>6</td>
<td>242</td>
<td>1633</td>
<td>2025</td>
<td>268,332</td>
<td>54.5</td>
</tr>
<tr>
<td>8</td>
<td>239</td>
<td>1634</td>
<td>2026</td>
<td>272,762</td>
<td>53.8</td>
</tr>
<tr>
<td>10</td>
<td>236</td>
<td>1635</td>
<td>2000</td>
<td>277,073</td>
<td>53.0</td>
</tr>
<tr>
<td><strong>httpd</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>385</td>
<td>1650</td>
<td>2039</td>
<td>368,600</td>
<td>58.6</td>
</tr>
<tr>
<td>2</td>
<td>381</td>
<td>1649</td>
<td>2042</td>
<td>374,948</td>
<td>57.9</td>
</tr>
<tr>
<td>4</td>
<td>377</td>
<td>1650</td>
<td>2043</td>
<td>381,289</td>
<td>57.2</td>
</tr>
<tr>
<td>6</td>
<td>373</td>
<td>1650</td>
<td>2044</td>
<td>387,557</td>
<td>56.5</td>
</tr>
<tr>
<td>8</td>
<td>368</td>
<td>1654</td>
<td>2042</td>
<td>394,121</td>
<td>55.8</td>
</tr>
<tr>
<td>10</td>
<td>364</td>
<td>1653</td>
<td>2045</td>
<td>400,620</td>
<td>55.0</td>
</tr>
</tbody>
</table>

Table A.2: Clustering and memory reduction for different core code growth bounds
<table>
<thead>
<tr>
<th>γ(%)</th>
<th>Kernel boot data</th>
<th>Application execution data</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of access</td>
<td>Total code loaded (KB)</td>
<td>Est. load cost (sec)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MiBench</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>66,736</td>
<td>111,842</td>
<td>8.68</td>
</tr>
<tr>
<td>2</td>
<td>43,933</td>
<td>73,817</td>
<td>5.71</td>
</tr>
<tr>
<td>4</td>
<td>16,964</td>
<td>28,537</td>
<td>2.21</td>
</tr>
<tr>
<td>6</td>
<td>8,647</td>
<td>14,080</td>
<td>1.12</td>
</tr>
<tr>
<td>8</td>
<td>3,412</td>
<td>5,644</td>
<td>0.44</td>
</tr>
<tr>
<td>10</td>
<td>1,700</td>
<td>2,787</td>
<td>0.22</td>
</tr>
<tr>
<td>MediaBench</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>65,964</td>
<td>109,222</td>
<td>8.58</td>
</tr>
<tr>
<td>2</td>
<td>27,802</td>
<td>46,825</td>
<td>3.61</td>
</tr>
<tr>
<td>4</td>
<td>11,389</td>
<td>19,037</td>
<td>1.48</td>
</tr>
<tr>
<td>6</td>
<td>5,831</td>
<td>9,472</td>
<td>0.76</td>
</tr>
<tr>
<td>8</td>
<td>3,130</td>
<td>5,009</td>
<td>0.41</td>
</tr>
<tr>
<td>10</td>
<td>1,723</td>
<td>2,861</td>
<td>0.22</td>
</tr>
<tr>
<td>httpd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>96,111</td>
<td>163,719</td>
<td>12.49</td>
</tr>
<tr>
<td>2</td>
<td>23,735</td>
<td>39,375</td>
<td>3.09</td>
</tr>
<tr>
<td>4</td>
<td>11,078</td>
<td>18,326</td>
<td>1.44</td>
</tr>
<tr>
<td>6</td>
<td>2,861</td>
<td>4,657</td>
<td>0.37</td>
</tr>
<tr>
<td>8</td>
<td>1,620</td>
<td>2,708</td>
<td>0.21</td>
</tr>
<tr>
<td>10</td>
<td>1,030</td>
<td>1,696</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Table A.3: Cost of dynamic code loading for different core code growth bounds
A.3 Experimental Results of Dynamic Data Structure Compression

This section shows the detail experimental data of dynamic data structure compression, which is described in Chapter 5. Table A.4 shows the detail results of memory usage and reduction of Linux slab allocators with different compression ratios comparing with the original kernel. Table A.5 shows the detail results of corresponding performance overhead with different compression ratios.

### Table A.4: Memory pages usage and size reduction of Linux slab allocators

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Orig. pg</th>
<th>r=0.0 pg</th>
<th>r=0.2 pg</th>
<th>r=0.4 pg</th>
<th>r=0.6 pg</th>
<th>r=0.8 pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>476</td>
<td>404</td>
<td>15.1</td>
<td>402</td>
<td>15.5</td>
<td>408</td>
</tr>
<tr>
<td>copy 5MB</td>
<td>558</td>
<td>465</td>
<td>17.0</td>
<td>468</td>
<td>16.1</td>
<td>479</td>
</tr>
<tr>
<td>copy 20MB</td>
<td>884</td>
<td>691</td>
<td>21.8</td>
<td>717</td>
<td>18.9</td>
<td>755</td>
</tr>
<tr>
<td>Avg. Reduce</td>
<td></td>
<td>18.0</td>
<td>16.9</td>
<td>14.3</td>
<td>14.3</td>
<td>11.2</td>
</tr>
</tbody>
</table>

### Table A.5: Running time and overhead of benchmark programs

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Orig. sec</th>
<th>r=0.0 sec</th>
<th>r=0.2 sec</th>
<th>r=0.4 sec</th>
<th>r=0.6 sec</th>
<th>r=0.8 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>5.948</td>
<td>6.004</td>
<td>0.9</td>
<td>6.001</td>
<td>0.9</td>
<td>6.020</td>
</tr>
<tr>
<td>copy 5MB</td>
<td>1.625</td>
<td>1.715</td>
<td>5.5</td>
<td>1.683</td>
<td>3.6</td>
<td>1.653</td>
</tr>
<tr>
<td>copy 20MB</td>
<td>8.835</td>
<td>10.207</td>
<td>15.5</td>
<td>10.022</td>
<td>15.4</td>
<td>9.280</td>
</tr>
<tr>
<td>Avg. Overhead</td>
<td>7.3</td>
<td>6.0</td>
<td>1.1</td>
<td>6.0</td>
<td>1.1</td>
<td>6.0</td>
</tr>
</tbody>
</table>

### Table A.5: Running time and overhead of benchmark programs
REFERENCES


