Varmosa: Just-in-time Binary Translation of Operating System Kernels

by

Perry L. Hung

Bachelor of Science in Electrical Engineering and Computer Science,
Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
[JUNE]
May 2009

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by .....
Larry Rudolph
Principal Research Scientist
M.I.T Thesis Supervisor

VI-A Company Thesis Supervisor

Accepted by .....
Arthur C. Smith
Professor of Electrical Engineerning
Chairman, Department Committee on Graduate Theses

Varmosa: Just-in-time Binary Translation of Operating System Kernels

by

Perry L. Hung

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a just-in-time binary translation scheme that dynamically switches between system emulation with a slower but more memory efficient instruction interpreter, and a faster, more memory intensive binary translator. In testing, this hybrid intepreter/translator scheme reduced the size of the binary translation cache by up to 99% with a slowdown less than a factor of 5x in the worst case, and less than a 2x in the best case compared to a pure binary translation scheme. With only a 10% decrease in performance, upwards of 49% memory reduction is demonstrated. Additionally, a technique of guest kernel introspection and profiling using binary translation is presented.

Thesis Supervisor: Larry Rudolph
Title: Principle Research Scientist, Computer Science and Artificial Intelligence Laboratory

Acknowledgments

First and foremost I would like to express my gratitude to my thesis advisor and friend Larry Rudolph. This thesis would not have been possible if not for his constant support, direction, and understanding over the last year.

I would also like to thank VMware's Mobile Virtualization Platform (MVP) technical team for their patience and tutelage in the science and practice of virtualization. They have without exception answered my questions and taken time out of their busy schedules to help me, even when asked nonsensical questions. Many thanks to my manager Julia Austin, for being so understanding of my unusual sleep and work schedules and accepting so many late timecards.

It would be remiss to forget Anne Hunter, course administrator and the subject of many acknowledgments over the years. She has provided constant reassurance and support throughout my undergraduate and graduate education, helped me understand the convoluted policies of the MIT EECS department, and fought for me when I needed it.

Thanks to my friends, for being there when I needed it and for constant entertainment: Mike Rieker, Alex Fainkichen, Thomas Evans, Bharadwaj Yadavalli, Harvey Tuch, Emil Sit, Viktor Gyuris, Prashanth Bungale, Scott Devine, Michael Lyons, Hubert Pham, Gerald Fry, Lucy Wang, Jon Okie Williams, Erica Griffith, Justen Calvrett Aninye, Charlie Agoos, and my cat, Tux.

Of course, I would not be where I am without my family Benny, Rosita, and Elliot, for their unerring support, encouragement. and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The design and implementation of a virtualization layer for mobile and embedded devices presents a number of interesting technical challenges not present in the desktop and server space. Compared to their more powerful desktop counterparts, embedded systems are typically heavily constrained in both computing power and memory size. Additionally, hardware support for virtualization has not yet been added to common embedded CPU architectures, complicating the development of any potential virtualization scheme for embedded systems. While some well-known techniques used in desktop virtualization systems such as binary translation can also be used to virtualize embedded systems, such systems often have far larger amounts of memory available to it. With significantly smaller amounts of memory available to embedded devices, any embedded virtualization system must be more conservative. Using a combination of an instruction interpreter and an instruction set binary translator, this thesis presents a just-in-time binary translation scheme for virtualizing unmodified guest operating systems for the ARM embedded microprocessor architecture. By dynamically switching between a slower, more memory efficient interpreter and a faster, more memory intensive binary translator, memory usage can be reduced significantly with only minor performance degradation. I characterize the time-space tradeoff introduced by such a scheme, and additionally introduce a method of virtual machine guest kernel introspection and profiling using binary translation.

While virtualization in the desktop and server space is a mature and well under-

stood topic, the prospect of virtualizing mobile and embedded systems is relatively new. Only recently have advancements in embedded hardware resulted in the possibility of efficiently virtualizing embedded systems. Today, powerful mobile devices such as cellular phones, internet tablets, netbooks, and personal digital assistants (PDAs) all offer sufficient speed and memory such that the prospect of virtualizing mobile devices is now possible. However, given the relative infancy of embedded virtualization, the overheads and techniques associated with virtualizing embedded systems are not as well-understood.

To this end, this thesis presents a just-in-time dynamic binary translation scheme for Varmosa (virtual ARM OS "A"), a research project started at MIT and funded by VMware aimed at investigating and implementing various virtualization techniques for the ARM microprocessor. By performing profiling of guest operating system code using an instruction interpreter, this system can identify frequently executed sections of guest code. Such sections are then translated by a binary translator for more efficient execution. This scheme is specifically designed for reducing the memory usage required by the Varmosa virtualization layer while still maintaining acceptable performance. In testing, this hybrid intepreter/translator scheme reduced the size of the binary translation cache by up to 99% with a slowdown less than a factor of 5x in the worst case, and less than a 2x in the best case compared to a pure binary translation scheme. With only a 10% decrease in performance, upwards of 49% memory reduction is demonstrated.

Additionally, a technique of guest kernel introspection and profiling using binary translation is presented.

The following chapter introduces the basic theory and implementation of virtualization systems, and outlines the advantages and disadvantages of each scheme. Chapter 3 begins with a discussion of the design motivations for Varmosa and outlines the techniques used for virtualizing the ARM CPU architecture. In the latter part of the chapter, a just-in-time binary translation method for reducing memory usage is described. Chapter 4 discusses a method of guest profiling and introspection techniques using binary translation. Chapter 5 presents experimental results with

analysis and discussion. Finally, chapter 6 concludes with a summary and outlines possible future work.

1.1 Related Work

Software dynamic binary translation is found in a variety of systems, including FX!32[10], Shade[5], PIN[7], DynamoRIO[3], and Embra[14], with many of them using similar techniques as Varmosa for translation cache performance optimization. These systems perform same-ISA translation and optimization using information discovered at runtime, but are not used for virtualization.

Binary translation in the context of virtualization has been used in a number of systems including Disco[4], VMware[1], and QEMU[2]. Each of these systems use binary translation to overcome non-virtualizability requirements but do not target memory usage and do not integrate interpretation as a significant part of the system.

Similary, just-in-time compilers translate from high-level languages to machine code and cache the results. Perhaps the most relevant example is Sun Microsystem's Java Hotspot[6] technology, used in the Java Virtual Machine. Hotspot performs profiling of commonly executed code blocks in an interpreter, then compiling those blocks into native machine code on-demand to speed-up performance. Hotspot, however, is restricted to user-mode applications–profiling and efficiently translating operating system code is a more involved process.

Mobile virtualization has been explored by Open Kernel Labs[13], TRANGO systems[12], and Xen. All three of these systems use paravirtualization as their primary method of virtualization and do not perform full-system emulation.

Chapter 2

Virtualizing the ARM Architecture

2.1 Overview

Varmosa is a hosted, full system virtualization platform for the ARM microprocessor architecture. It implements complete processor and memory virtualization for the ARMv4 instruction set, and presents a number of virtualized devices from the reference ARM Integrator Control Platform (ICP) to guest operating systems. Using this platform, we have demonstrated the execution of unmodified instances of Windows CE and Linux with network and graphics capabilities.

In this chapter, this thesis presents a brief background of virtualization techniques and relevant terminology, then describes the design and implementation challenges in virtualizing the ARM architecture. Additionally, the various design motivations for Varmosa are outlined, as well as the function and performance characteristics of system emulation and binary translation in Varmosa.

2.2 System Virtualization Background

Because the term "virtualization" is loosely defined and used broadly in many different contexts, we restrict our discussion to a form of virtualization known as *system virtualization,* or the virtualization of a complete system or platform environment. This type of environment (called a *host*) can support the concurrent execution of one

or more *virtual machines* (VM), each with a *guest* operating system along with its many user processes. In system virtualization, each virtual machine is presented a complete interface to a set of "virtual" hardware resources such as a processor, memory, and I/O devices. The virtual interface presented to the guest is often the same as that of the host, but this is not a necessary condition.

The concurrent execution of multiple operating systems immediately raises a problem: **modern operating systems *require* that they have privileged access to the hardware.** More concretely, most operating systems utilize the hardware features of modern processors such as privileged execution modes (e.g. user mode vs kernel mode) and virtual memory mechanisms to both protect itself from user processes, and provide isolation and security for user processes. Access to privileged hardware state is therefore necessary for their function. However, separate virtual machines can clearly not both be given access to real, privileged hardware state, as this would compromise the security of the system. For example, if two different guest operating systems were able to both modify page table permissions, a malicious guest could arbitrarily read or write the memory of the other, an undesirable security flaw.

The solution to this problem is to **deny guests from modifying or inspecting real machine state**. Instead, guests manage their own set of virtual hardware, while the real hardware is managed by inserting an intermediary layer of software known as a *Virtual Machine Monitor* (VMM) or *hypervisor* between the hardware and guest virtual machines. The VMM thus owns exclusive control of the hardware and is responsible for managing the state of the virtual hardware presented to each VM.

In practice, modern virtual machine monitors or hypervisors are typically implemented in three possible ways: hardware-assisted virtualization, paravirtualization, or software full-virtualization.

2.2.1 Hardware-Assisted Virtualization (Trap-and-emulate)

One possible resolution of the hardware multiplexing problem is by *de-privileging* the guests. Just as operating systems use hardware privilege levels to partition user

mode processes from kernel mode operation, one could run each guest in user-mode, ensuring that any privileged operations that modify or inspect privileged machine state trap or generate an exception. In this manner, the VMM would sit directly above the hardware, maintain a set of "virtual" hardware state for each guest, and wait for a guest to execute a privileged instruction. When a guest attempts to do so, the CPU traps, vectoring execution to the VMM. The VMM can then emulate that instruction's expected effect on the virtual hardware state, and return execution to the guest. This scheme is often called *trap-and-emulate.*

Trap-and-emulate schemes have strict requirements before they can be applied. In 1974, a classical paper by Popek and Goldberg[9] outlined the formal requirements for virtualizability using this scheme:

1. A processor architecture must have at least a privileged and an unprivileged mode

2. The set of sensitive instructions is a subset of the privileged instructions.

Under Popek and Goldberg's requirements, a *sensitive instruction* is defined as an instruction that needs to be executed in a privileged mode because they either modify or inspect hardware state (for example, change the MMU settings). A *privileged instruction* is an instruction that generates a processor trap or exception if executed in an unprivileged mode. In other words, if an instruction which is not allowed to execute in user mode is executed, it **must** trap to be handled by the VMM.

Trap-and-emulate requires that the Popek and Goldberg criteria are met. However, modern processor architectures such as the x86 and ARM do not meet these criteria, as there exist sensitive instructions in both architectures that do not trap. In order to address this shortcoming, hardware manufacturers such as Intel and AMD have recently introduced hardware-assist virtualization technologies in their x86 CPUs that introduce an additional privilege level in which the x86 instruction set is modified such that all instructions meet the Popek and Goldberg virtualization requirements.

The ARM architecture does not to date have hardware-assist capabilities. **It is not virtualizable under trap-and-emulate.**

2.2.2 Paravirtualization

In paravirtualization, the problem of keeping guests from executing privileged instructions and modifying or inspecting hardware state is solved simply by eliminating such instructions from the guest code. In this architecture, the source code of the guest is modified to remove all privileged instructions, and instead make *hypercalls* to the VMM, which then handles and emulates virtual hardware state.

The primary advantage of paravirtualization is lower virtualization overhead, since guest operating systems can be modified to communicate directly with the VMM instead of through a potentially large and complicated hardware interface. Additionally, hypervisors in these schemes tend to be easier to implement, frequently exhibiting smaller footprints than full virtualization solutions.

However, paravirtualization presents a number of disadvantages. While paravirtualization allows for a simpler VMM implementation, guest operating systems often require deep and invasive changes to their kernels in order to port them to a paravirtualized interface, effectively shifting the complexity of virtualization from the hypervisor to the guest. In cases where the source code of the guest is not available (as in many commercial operating systems such as Microsoft Windows or Mac OS X), it can not be run on such a system.

2.2.3 Software Full Virtualization

Full system virtualization provides a complete simulation of the underlying hardware so that operating systems and other guest software may run on the virtual hardware in the same manner as they would have if it were on real hardware. In full-system virtualization, the virtual hardware interface presented to the guest is for all intents and purposes, identical to the interface of the original hardware. Guest operating systems may freely execute privileged instructions, interfacing with virtual hardware with the full belief that it is actually interacting with real hardware. Because of this property, **full virtualization systems can run unmodified guest binaries.**

There are two main techniques for software based full virtualization: interpreta-

tion and binary translation. In interpretation, the code for a guest system is emulated instruction by instruction by a software emulation layer in the VMM called an interpreter. Because *every* single instruction must be fetched, decoded, and emulated by the VMM, emulation techniques typically exhibit unacceptable performance degradation–even an efficient emulator may easily experience several hundred times slowdown over native execution.

An alternative method is binary translation. In binary translation, guest code is fetched by the VMM in blocks of multiple instructions at a time. These instructions are analyzed for unsafe privileged instructions, then translated into an equivalent instruction sequence, albeit replaced with "safe" instructions. These translated, safe instructions are then saved away in memory in a *translation cache*, so that the next time they are executed, they do not need to be retranslated. Binary translation is significantly faster than emulation, and can exhibit near-native performance[1].

One particular concern in using binary translation is that translated blocks must be saved in memory, expanding the memory footprint the hypervisor.

2.3 Varmosa Design and Implementation

The Varmosa virtual machine monitor is a type-2 hypervisor[8], otherwise known as a *hosted* hypervisor. More concretely, Varmosa is installed as an application on an existing, host operating system as opposed to running *bare-metal*, or directly on the hardware. The benefits of such a model is that it runs like an application, allowing it to leverage the existing low-level capabilities of the host such as device drivers. This reduces development time, with the downside of reduced performance because resources need to be allocated to the host.

In the hosted model, our hypervisor is split up into two parts: a user-mode half called the VMX, and the kernel-mode half called the VMM. Whenever the host operating system schedules our application (the VMX), Varmosa seizes control of the hardware via a kernel module and installs itself (the VMM) onto the hardware. At this point, the Varmosa VMM is in total control of the system, completely replacing

the state of the host with its own state–the VMM has its own set of page tables, interrupt and exception handlers, etc. At this point, the VMM is running at the most privileged level, and can begin the process of running guest virtual machines.

In order to actually run guest code, however, it must virtualize the CPU of guest VMs. Varmosa is a full virtualization solution, and uses a combination of direct execution, interpretation, and binary translation to virtualize the ARM architecture.

2.3.1 ARM CPU Virtualization

Given the three possible architectures for virtualizing ARM processors, Varmosa implements software full virtualization with the following reasoning:

1. **Hardware-assist**: The ARM architecture does not meet the Popek and Goldberg virtualization requirements, as there exist sensitive instructions which do not trap. One such example is the `cps` instruction, which modifies privileged processor state in privileged mode, but fails silently in user mode. Hardware-assisted virtualization does not yet exist to fix this, so this method is impossible.

2. **Paravirtualization**: Source code for common embedded operating systems such as Windows CE, Symbian, and iPhone OS are not easily obtained, making paravirtualization impractical.

By process of elimination, the only choice remaining is software-based full virtualization. The remainder of this chapter describes the various techniques used by our VMM to virtualize the ARM architecture.

2.3.2 Direct Execution

In direct execution mode, the control is transferred directly to the unmodified guest. This mode is only suitable in limited cases–namely in the case when the guest wishes to run code at the lowest privilege level (user-mode). In other words, the direct execution mode is used for the user-mode processes of the guest. This mode effectively

side-steps the issue of non-virtualizability of the ARM architecture, as user-mode processes do not execute privileged instructions anyway.

Direct execution mode is extremely efficient, as the original guest code is running directly on the hardware. Just like an operating system, the VMM will seize back control of the processor at the next interrupt, or if the guest attempts to execute a privileged instruction or system call. In Varmosa, the VMM will execute in direct execution mode for all guest code that is meant to be run at user-level. In this manner, user-mode processes essentially experience native performance.

2.3.3 Interpretation

For kernel code, one way of overcoming the non-virtualizability of the ARM architecture is by using an interpreter. In interpretation, the hardware state of each virtual machine is fully represented by an in-memory data structure. Instead of directly executing guest code on the physical CPU, guest instructions are executed by an interpreter, effectively eliminating the semantic obstacles imposed by the non-virtualizability property. In interpretation mode, guest instructions are read by the VMM one at a time. The instruction is then decoded, and based off the opcode, the appropriate VMM emulation routine for that opcode is called. This is called a *fetch-decode-execute* cycle.

In pseudocode, a typical implementation might look like:

```
while (1) {
    instruction = guestRead(guestPC);
    opcode = decode(instruction);

    switch (opcode) {
    ...
    case ADD:
        rd = getRd(instruction);
        rn = getRn(instruction);
        rm = getRm(instruction);
        guestCPU->reg[rd] = guestCPU->reg[rn] + guestCPU->[rm];
        guestCPU->pc += 4;
        break;
    ...
```

```
    }
}
```

A full ARM instruction interpreter is implemented in Varmosa. While such a system has the advantage that it is quite easy to implement, it has the significant drawback of poor performance. Whereas in direct execution mode the guest code runs directly on the processor, in interpretation mode *every* single instruction must be fetched, decoded, and emulated, quickly increasing the time it takes to emulate the guest system. For each guest instruction, upwards of hundreds of physical instructions may be executed[1]. Interpretation also tends to exhibit poor branch predictor performance, due to the unpredictable branch that the switch statement introduces.

Nonetheless, the interpretation mode in Varmosa is quite powerful. Despite it's relatively poor performance, it is extremely memory efficient, requiring only a handful of in-memory data structures to completely encapsulate the state of the virtual hardware. It is this property that this thesis leverages to reduce the memory footprint of Varmosa.

2.3.4 Binary Translation

Binary translation is the general technique of converting a *source* binary program into a *target* binary program. In general, the source architecture need not be the same as the target architecture (for example, one could use a binary translator to convert ARM binaries to x86 binaries). This is called a cross-instruction set (cross-ISA) binary translator. When the source and target architectures are the same, this is called a same-ISA binary translator. In Varmosa, our binary translator is capable of performing both types of translation, for example from a ARMv4 guest to an ARMv6 host, or an ARMv4 guest to an ARMv4 host.

Varmosa uses binary translation to overcome two major obstacles, first, the non-virtualizability of the ARM instruction set, and second, the poor performance of interpretation.

Regarding the first obstacle, guest code can contain sensitive or privileged instructions that can not be allowed to execute directly on the physical hardware. With bi-

nary translation, guest code is analyzed on-demand, just before it is about to execute. Any sensitive instructions are recognized by the binary translator and translated into "safe" equivalents for execution on the host. This preserves the semantic intent of the original guest code, effectively overcoming the ARM architecture's failure to conform to the Popek and Goldberg criteria.

With regards to the second obstacle, once the binary translator has translated a section of guest code, the translated code is saved in memory in a *translation cache*, so that the next time that code is executed, the VMM can jump directly to the saved code, effectively amortizing the cost of translation over many executions.

In this manner, binary translation allows for efficient software emulation of an entire virtual machine while still preserving the semantic correctness of an interpreted approach. There are drawbacks, however, associated with binary translation. First, binary translation is difficult to implement, requiring complicated data structures and rules for correct translation. Second, it takes time to translate instruction blocks—translating a block that is going to be executed only a handful of times may not be worth the time it takes to translate. Third, because each translated block must be saved in the translation cache, binary translation consumes an increasing amount of memory as more code is translated.

Nonetheless, because of its comparatively high-performance, all guest kernel code in Varmosa is executed through the binary translator whenever possible. In the next chapter, this thesis presents an alternative scheme which performs profiling of guest kernel code using interpretation, switching to the binary translator only for frequently executed code. Binary translation in Varmosa is also discussed in more depth in chapter 4, where a method of using the Varmosa binary translator for guest kernel introspection and profiling is presented.

Chapter 3

Just-In-Time Binary Translation

The driving motivation behind this optimization is to reduce the memory footprint incurred by the binary translation system in Varmosa. In order to maximize virtualization performance, the default behavior in Varmosa is to directly execute user code whenever possible, and use the binary translator for all guest kernel code[1]. Because all guest kernel code is translated, the size of the translation cache increases as more and more code is executed. However, we begin by assuming the following postulate: **workloads are typically dominated by a very small percentage of the code.** In other words, given a large program, most of the time is spent in only a very small portion of it. This thesis demonstrates that such an assumption is often a reasonable one to make.

Assuming this proposition, we present the following idea: because a small percentage of code used in a program is responsible for most of its execution, Varmosa by default interprets all guest code, performing execution profiling of the guest. When it sees that a certain section of code is executed more than a threshold T number of times, it performs *just-in-time* (JIT) binary translation on that section of code, so that all future invocations of that code execute faster. In this manner, uncommonly executed code is not translated, saving space in the translation cache. However, because the commonly executed code dominates the execution time, the extra cost of

[1]This is not technically true–Varmosa's binary translator is not complete and falls back to the interpreter for a few limited cases of instructions in which a suitable translation is very difficult, or if a translation rule has not yet been implemented.

interpretation is somewhat mitigated. Clearly, this method introduces a time-memory tradeoff.

The remainder of this chapter presents the design and implementation of this method. The time-memory tradeoff characteristics are described, and a heuristic for selecting a threshold T is proposed. In benchmarks this hybrid intepreter/translator scheme reduced the size of the binary translation cache by up to 99% with a slowdown less than a factor of 5x in the worst case, and less than a 2x in the best case compared to a pure binary translation scheme. With only a 10% decrease in performance, upwards of 49% memory reduction is demonstrated.

3.1 Design

The central idea behind the JIT method is that guest code is evaluated on-demand as it is executed, recording a history of execution and branching to the binary translator when appropriate. In order to implement this scheme, we first define the following:

Definition: A *basic block* is a contiguous sequence of instructions up the next control-flow changing instruction.

An example basic block of guest code might look like (presented in a (*guestAddress, instruction, disassembly*) tuple format):

```
C0076064: [E28C0064] < add   r0, r12, #100   ; 0x64
C0076068: [E280000C] < add   r0, r0, #12 ; 0xC
C007606C: [E5903000] < ldr   r3, [r0]
C0076070: [E1520003] < cmp   r2, r3
C0076074: [8AFFFFFB] < bhi   C0076068
```

Definition: A *control-flow changing* instruction is any instruction that may potentially cause a branch in control-flow, excepting instructions that may cause exceptions (e.g. a data-abort, or an invalid memory access).

An example of a control-flow changing instruction is any branch instruction or a load or data processing instruction for which the destination register `Rd` is the program

counter (PC) or (R15) register. See Appendix B for a list of control-flow changing instructions.

With these defined, basic algorithm can be outlined.

3.1.1 Implementation

Execution of all guest code begins in the VMM. At this point, the VMM is running in a privileged mode, and guest virtual machine state is stored in an in-memory data structure. Emulation of guest code begins at the virtual CPU's current program counter. First, Varmosa determined the current privilege level of the guest, if it is attempting to run guest user-mode code, it switches into user-mode and vectors operation directly to the guest (Direct Execution mode). Otherwise, it

proceeds in units of basic blocks with the following algorithm in C-pseudocode:

```
while (1) {
    block = getBlock(guestCPU->pc);

    if (getExecutionCount(block) > threshold) {
        enterBinaryTranslation(block);
        /* Never returns, execution has been vectored to the guest */
    }

    /* Otherwise, scan forward for the next control-flow changer,
       grabbing the number of instructions in the block */
    numInstructions = getBlockLength(block);

    /* Emulate all the instructions in the block  */
    for (i = 0; i < numInstructions; i++) {
        guestInterpret(guestCPU);

        /* Check if the guest PC has jumped outside the block,
           if so, this indicates an interrupt or exception  */
        if (checkPC(block))
           break;
    }

    /* Increment our stored count for this particular block */
    incrementExecutionCount(base);
}
```

For each basic block, a mapping is contructed from the base address of each guest basic block to a saved counter in memory. This is stored in a lightweight hash table. Every time a given block is interpreted, the counter is incremented. When the counter exceeds a given threshold, the VMM will enter binary translation mode, jumping out of the main loop, translating the block, and vectoring control to the guest.

Once binary translation mode is entered, the VMM will not regain control again until the next CPU interrupt, exception, or trap. Once the VMM does regain control, our main loop is once again entered. In this manner, every time the VMM runs, it makes a decision based off the execution count of the current guest program counter about whether to enter binary translation mode or continue the interpretation loop.

3.1.2 Practical Complications

This algorithm, while simple in theory, presents a number of complications. First, we have adopted the conventional notion of a basic block as our fundamental unit. In conventional binary translators, basic blocks are used because they represent a unit of straight-line code. That is, one can be certain that each instruction of the block is executed given an entry point into the beginning of the block, because blocks only contain control-flow changing instructions as terminators. This obviates the expensive accounting of keeping a counter for each instruction in memory. While this is true for user-code, we are attempting to emulate an entire kernel. This necessitates the correct emulation of all exception handlers, interrupts, memory access violations, etc. The fundamental result is that **program flow can be interrupted at any point in time**. As a result, a check is performed after each interpreted instruction to ensure that the current program counter is still running within the bounds of the basic block. If it is not, it indicates that an exception such as a data-abort (accessing invalid memory) or interrupt has occurred, so execution must resume at the guest's exception handlers. In these instances, it breaks out of the basic block emulation loop and starts over, since the guest is now executing in a different block.

Another complication is what happens when the guest wants to jump to an address that is in the middle of another block.

For example, consider the example basic block from above:

```
C0076064: [E28C0064] < add   r0, r12, #100   ; 0x64
C0076068: [E280000C] < add   r0, r0, #12 ; 0xC
C007606C: [E5903000] < ldr   r3, [r0]
C0076070: [E1520003] < cmp   r2, r3
C0076074: [8AFFFFFB] < bhi   C0076068
```

In this example, the block beginning at guest address 0xC0076064 ends with a conditional branch to address 0xC0076068, which is the second instruction in the block. Unfortunately counters are indexed in a hash table by the addresses of the beginning of each block. While one could perform the appropriate accounting by search to see if the target address lies within an already emulated block, this dramatically increases the lookup time for each control-flow change. Instead of these more expensive accounting techniques, Varmosa simply begins another block, associating a new counter with that address. If the branch is commonly taken, the target address block will quickly exceed the threshold.

One final and related problem occurs once binary translation mode has been entered. The binary translator uses the same basic block definition as defined above. It begins by translating a block instruction-by-instruction into a functionally equivalent set of safe instructions, replacing sensitive instructions with an equivalent set that emulates their function. It performs this translation with *pattern-based substition*, a technique which looks at each original instruction and replaces it with its translated equivalent by means of a simple pre-defined substitution rule. This technique is not particularly efficient, but is significantly easier to implement than alternative implementations. Even given its relative simplicity, the translation of certain instructions can prove to be difficult. Consider the ARM `ldrbt` instruction, or Load Register Byte with Translation. This instruction loads a byte from a memory using a post-indexed addressing mode and writes the value to a register. When executed in a privileged mode, the memory system treats the access as it were accessed in user mode, so a memory access which would be invalid in user-mode should trap to a privileged mode, allowing handling by a kernel exception handler. A translation of this instruction involves a complicated series of page table switches and cache flushes, so in cases such

as these, Varmosa will fall back to interpretation mode by replacing the instruction with an undefined instruction to ensure that the instruction traps and the VMM regains control. As above, the current guest program counter will lie in the middle of a block, with no counter associated with it. In these cases, we adopt the same scheme as above.

3.1.3 A simple optimization

Clearly, it would be desirable to be able to predict that a certain block is going to be executed frequently and avoid interpreting it.

Consider the following block:

```
C001E784: [E4D13001] < ldrb  r3, [r1], #1
C001E788: [E2522001] < subs  r2, r2, #1  ; 0x1
C001E78C: [E4C03001] < strb  r3, [r0], #1
C001E790: [1AFFFFFB] < bne   C001E784
```

This block ends with a conditional branch backwards to the beginning of the block, representing a small, classic while-loop. Looping blocks are extremely likely to be executed many times, so reasoning in classic branch-prediction fashion, we preemptively translate any block that executes twice in a row, skipping the interpretation stage. Using data gathered from profiling the guest, this prediction turns out to be true most of time. In an analysis of approximately 14,400 different basic blocks executed during the boot of a Linux guest, there were 118 blocks of this type. Of these, over 85% of were executed over 10 times, with many of them executing hundreds or thousands of times.

In the next chapter, a method of using the binary translator for doing guest kernel profiling is presented, along with discussion and analysis of the just-in-time scheme. The data gathered with the profiler is used to understand algorithmic performance.

Chapter 4

Guest Kernel Instrospection and Profiling

One final remaining point is considering how to determine what the threshold T should be set to while still maintaining "reasonable" performance. This section presents a method of using the binary translator to profile the operation of guest kernel code, enabling an accurate and complete recording of an execution trace for guest code.

4.1 Kernel Profiling with Binary Translation

This section presents a method of using Varmosa's binary translator to construct a complete *control-flow graph* (CFG) of the execution of a guest. Unlike some other kernel profiling methods, this method is *complete* and does not rely on statistical sampling techniques to obtain a graph. It is also dynamic in that it can be turned on and off selectively, and has the additional advantage that it does not require instrumentation or modification of the kernel to be profiled. Figure 3-1 shows a sample of the type of traces that the profiler can be used to obtain.

This method is typically not used simultaneously with the just-in-time algorithm, but aids in understanding the performance characteristics of the former. Using this method, the binary translator can track the number of times every block is executed, as well as construct an entire control-flow graph for a kernel.

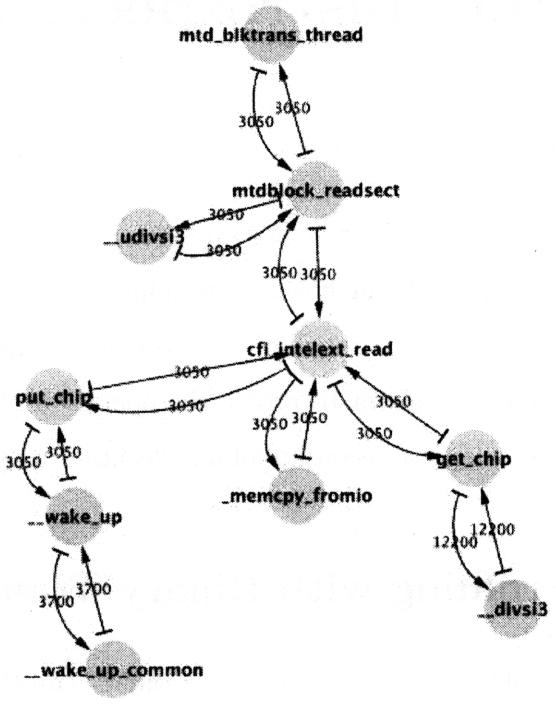

Figure 4-1: A sample Linux kernel call-trace
A sample Linux kernel call-trace, tracking function call and returns within the Common Flash Interface (CFI) subsystem

As previously outlined, the binary translator in Varmosa strictly controls the execution of the guest by translating blocks as they are encounted in guest code. As each block is translated, it is saved in the translation cache, with the intention of amortizing the cost over time if blocks are executed many times. Execution within the translation cache is controlled via hardware mechanisms, ensuring that the guest can not arbitrarily exit the translation cache. However, once execution has been vectored to the guest in binary translation mode, it is possible for it to remain within the translation cache for many clock cycles. The basic idea of using this method is to force the guest to exit at every control flow change. This is implemented using a combination of the hardware exception mechanisms and a simple modification of the binary translator.

Basic profiling is performed with a choice of two methods, the first of which allows an efficient way of gathering the number of times a block was executed within the system. The second, slower method enables the gathering of control-flow.

4.1.1 Execution counters

First, consider the our canonical example block from the last chapter:

```
GUEST INSTRUCTIONS
[GUEST]    C001E784: [E4D13001] < ldrb  r3, [r1], #1
[GUEST]    C001E788: [E2522001] < subs  r2, r2, #1  ; 0x1
[GUEST]    C001E78C: [E4C03001] < strb  r3, [r0], #1
[GUEST]    C001E790: [1AFFFFFB] < bne   C001E784
```

As outlined before, Varmosa's binary translation system translates guest binaries using a pattern-based substitution system, replacing each instruction with a santized equivalent. For example, the basic block above is translated into the following, described as a (*guestAddress, translationCacheAddress, translatedInstruction, disassembly*) tuple:

```
TRANSLATION Start addr : E83ED82C
[C001E784] E83ED82C: [E4F13001] > ldrbt r3, [r1], #1
[C001E788] E83ED830: [E2522001] > subs  r2, r2, #1  ; 0x1
[C001E78C] E83ED834: [E4E03001] > strbt r3, [r0], #1
[C001E790] E83ED838: [1F000000] > svcne 0x00000000
           E83ED83C: [EF000000] > TERMINATOR
```

The basic method of keeping a block-level counter for each translated block is to simply insert straight-line, rudimentary counter code directly into the beginning of each translated block:

```
TRANSLATION Start addr : E83ED82C
BEGIN Counter code:
E83ED82C: [E58F0018]          // Spill r0
E83ED830: [E59F0010]          // Load COUNTER to r0
E83ED834: [E2800001]          // Add one to r0
E83ED838: [E58F0008]          // Store r0 back to COUNTER
E83ED83C: [E59F0008]          // Fill r0
E83ED840: [EA000002]          // Jump over the following data
E83ED844: [STAT_MARKER]       // Magic constant for finding it later
E83ED848: [COUNTER]           // Counter location for this block
E83ED84C: [SPILL]             // Spill Location
END Counter Code

TRANSLATION
[C001E784] E83ED850: [E4F13001] > ldrbt r3, [r1], #1
[C001E788] E83ED854: [E2522001] > subs  r2, r2, #1  ; 0x1
[C001E78C] E83ED858: [E4E03001] > strbt r3, [r0], #1
[C001E790] E83ED85C: [1F000000] > svcne 0x00000000
           E83ED860: [EF000000] > TERMINATOR
```

By doing this for every translated block (or a subset if doing targeted profiling), it can be guaranteed that the counters are incremented every time each block is executed. After the end of a profiling run, the translation cache can then be scanned for the magic `STAT_MARKER` constant and the counters for each block extracted.

Obviously, there is a necessary and significant increase in the size of the translation cache. However, for the purposes of profiling this is a significantly more efficient method than using the interpreter to obtain the equivalent data, as the binary translator is considerably more efficient than the interpreter.

4.1.2 Constructing control-flow graphs

The second method presented here allows the construction of an execution trace for a guest kernel. Consider again the block above (counter code removed for brevity):

```
GUEST INSTRUCTIONS
[GUEST]     C001E784: [E4D13001] < ldrb  r3, [r1], #1
[GUEST]     C001E788: [E2522001] < subs  r2, r2, #1  ; 0x1
[GUEST]     C001E78C: [E4C03001] < strb  r3, [r0], #1
[GUEST]     C001E790: [1AFFFFFB] < bne   C001E784

TRANSLATION
[C001E784] E83ED82C: [E4F13001] > ldrbt r3, [r1], #1
[C001E788] E83ED830: [E2522001] > subs  r2, r2, #1  ; 0x1
[C001E78C] E83ED834: [E4E03001] > strbt r3, [r0], #1
[C001E790] E83ED838: [1F000000] > svcne 0x00000000
           E83ED83C: [EF000000] > TERMINATOR
```

The first three instructions in the guest block are translated more or less identically, with the addition of the `T-bit` set in the translated version, signifying a load/store *as-user* instruction. This means the translated versions will perform the access with the guest page tables, ensuring that a data-abort (memory-access violation) will cause an exception to be handled by the VMM. These are more-or-less uninteresting. The last two translated instructions are considerably more interesting. The `bne` guest instruction is a conditional branch, branching to its target address depending on the state of the Z-bit processor condition code. In this case, the original instruction has been replaced with two instructions, a `SWI` software-interrupt instruction, and a magic `TERMINATOR`, another SWI.

The `SWI` is placed in the translation so that the first time the block is translated, the execution of the `SWI` guarantees a trap, because the target address of the branch may not yet have been translated. This is to ensure that execution doesn't jump to some unpredictable location in memory. In this particular example, this is a redundant step, because the target is itself (definitely already translated!), but this is not always the case. However, once the `SWI` has been executed and the target block has been translated, the binary translator then performs an optimization step called

linking, which hot-patches the translated code to point directly to the target block, so that future executions will jump directly to the target without an expensive trap. The `TERMINATOR` SWI instruction is for the fall-through case, again ensuring a trap. This can also be patched once the target has been translated.

Indirect branches such as a branch through a register are handled differently, because the target addresses are unpredictable. Code that executes within the translation cache is clearly located at a different section of memory, but guest code assumes that it is running code at its original location in guest memory (e.g. around 0xC000000 for a Linux kernel), so indirect branches must look up the target address in a hash-table that maps guest virtual addresses to translation cache addresses. As an optimization, the binary translator will avoid an expensive trap into the VMM and instead insert a small trampoline to a snippet of code that performs a hand-optimized fast hash table lookup, only jumping back to the VMM if the lookup fails (again because the target has not yet been translated).

In order to track execution flow, it must be guaranteed that execution returns to the VMM at the end of every block. In order to do this, we deliberately break both optimization schemes—first by disabling the linking phase from happening, and second by bypassing the hash table lookup. Both schemes guarantee that execution returns to the VMM at the end of every block, allowing us to compute and record the next address, forming a control-flow graph. Because of limited memory and the large number of edges in the graph, we cache the recorded data in memory and periodically flush them over a network connection to a PC.

This method as well as the previous method can be applied to either every single block in the translation cache, or selectively to perform targeted profiling, for example to see how often a certain system call is performed, or to track inefficiencies or control flow in a running guest kernel.

In the next chapter, experimental results from the just-in-time scheme are presented, and data gathered from profiling using the binary translator is used to understand the results, and select a suitable threshold T.

Chapter 5

Experimental Results

This chapter presents the benchmarks and experimental results used for evaluting the just-in-time binary translation method, along with analysis and discussion of the results. Profiling data gathered from the techniques outlined in the previous chapter is used to understand and select recommended general thresholds depending on desired memory usage and performance.

5.1 Benchmarking Framework

For testing, Varmosa is run on a standard Nokia N810 internet tablet, running the OS2008 Diablo 4.1 release of Maemo Linux. The host kernel is a 2.6.21-omap1 kernel compiled for the ARMv6 architecture. The benchmarking target is an unmodified est virtual machine running a 2.6.21.5 Linux kernel compiled for the reference ARM Integrator Control Platform with the ARMv4 instruction set. Timing is performed via the `mrc` instruction to access the `CP15` coprocessor cycle counter.

This thesis uses three different benchmarks to measure the performance of Varmosa:

- **boothalt32**: This benchmark measures time from the beginning of boot of the virtual machine to entry into userspace, executing a special backdoor to the VMM to mark the end of the benchmark. `boothalt32` is primarily composed

of initialization code, composed of many different blocks that are run only a small number of times.

- **dd32**: Use the standard UNIX program `dd` to copy a 2MB file from the Linux device node `/dev/urandom` to a temporary file mounted on the emulated filesystem. The benchmark measures the time it takes to begin and complete the `dd` command. The `dd` benchmark utilizes the special kernel-emulated `/dev/urandom` device node frequently, intended to exercise a small portion of kernel code, but executed a large number of times. The translation cache and just-in-time hash table counters are reset just before the benchmark begins, ensuring that no blocks have yet been translated.

- **gzip32**: Use the standard UNIX compression program `gzip` to compress a 3MB file read from /dev/urandom to a tempory file on the file sytem. The benchmark measures the time it takes to begin and complete the `gzip` command. The `gzip` benchmark is primarily run in user-space, executing a small section of kernel-code to write its results to the emulated disk. The translation cache and just-in-time hash table counters are reset just before the benchmark begins, ensuring that no blocks have yet been translated.

Benchmarks were performed in the following manner:
For each benchmark in `boothalt32, dd32, gzip32`:

- For threshold $T = 0 \ldots 5, 10, 20, 50, 200, 500, 1000, 10000$:
  - For $i = 1 \ldots 3$:
    - Measure and record execution time t_i
    - Measure and record translation cache size s_i
  - Average recorded values for t_i, s_i

$T = 0 = BT_{ref}$ is defined as the fully binary translated scheme. This number is used as the reference value for comparison to the just-in-time algorithm.

5.2 Experimental Results

This section presents the results of each benchmark. Figures and tables are presented at the end of the chapter. Each benchmark is illustrated with two plots, one on a standard linear graph and another on a semilog graph, illustrating the translation cache size and execution time while varying the JIT threshold. Additionally, a table of execution times, translation cache sizes, and performance versus the reference binary translation scheme is presented.

In experimental tests, each benchmark exhibited a classic time-memory tradeoff, with the size of the translation cache decreasing as execution time increases. In particular, the figures illustrate an illuminating result: the size of the translation cache appears to exhibit exponential decay as the threshold increases. In fact, in all three tests, choosing a threshold of $T = 10000$ reduced the size of the translation cache to nearly zero, an approximately 99% reduction! In contrast, the slowdown did not exceed a factor of 5x in the worst case (`boothalt32`), and was as low as 1.65x in the best case (`gzip32`). This is a startling result.

Consider Table 5.3, where an extra threshold of $T = \infty$ is presented. $T = \infty$ is defined as the time it takes to do a pure interpretation scheme. Performing each run purely interpreted for the `dd32` run took well over an hour, an over 200x slowdown over the reference time. Although it consumed zero memory compared to our just-in-time scheme, it took nearly a hundred times longer than the $T = 10000$ scheme, which used only 10KB in the translation cache, a trivial amount! Moreover, when viewed on a semilog plot, we can see that choosing a threshold of just $T = 10$ reduces the size of the translation cache by 59% in the best case (Table 5.2, `boothalt32`), and 49% in the worst (Table 5.4, `gzip32`). Yet, the execution time only increased by a factor of 1.18x and 1.03x, respectively. In fact, even in the `boothalt32` (Table 5.2) benchmark, which exhibits the poorest performance in time overall, a 79% reduction in the size of the translation cache can be achieved with only a 2x slowdown.

Function	Block Size	Execution Count	Insts. Executed	% Execution
get_chip	82	3050	50100	0.54
__divsi3	22	12227	268994	0.58
memmap_init_zone	67	4095	274365	0.60
__const_udelay	15	20000	300000	0.65
__memzero	23	13654	314042	0.68
kmem_cache_create	49	6903	338247	0.74
do_timer	133	2620	348460	0.76
v4wbi_flush_kern_tlb_range	24	32778	786672	1.72
arm920_flush_user_cache_range	7	148736	1041152	2.28
_memcpy_fromio	5	1561600	7808000	17.11

Table 5.1: Linux Profile, Top 10 Blocks (`boothalt32`)

5.3 Discussion

The results reveal a staggering disparity in performance between a purely interpreted mode and our just-in-time scheme. In the `dd32` benchmark, we achieved a 100% reduction with the interpreter, but the run took over an hour and a half, more than two hundred times slower. With the just-in-time scheme, the same benchmark ran in one minute, achieving a 98% reduction, a difference of only 10KB. Yet, the slowdown was only a factor of 2.56x. In Chapter 3, we posited that this scheme would perform well if most of the time spent running a program is contained to a small portion of the code. We believe these results demonstrate that such an assumption is a reasonable one to make.

In fact, consider the `boothalt32` test. Using the binary translation profiling technique described in Chapter 4, we profiled `boothalt32`, measuring where the guest spent most of its time and sorting it by block, and correlating it with a symbol table from Linux to retrieve the original function name. Table 5.1 lists the top 10 most commonly run blocks out of 14,400 used during `boothalt32`, and their percentage of overall execution. The most frequently executed block in `boothalt32` is `__memcpy_fromio`, responsible for nearly 20% of the instructions executed on boot! In fact, out of 14,400 blocks executed in `boothalt32`, the top 1,000 blocks are responsible for over 85% of the instructions executed.

From these results, we can explain the characteristics of the JIT benchmark results. For a given program, even though interpretation is significantly slower than

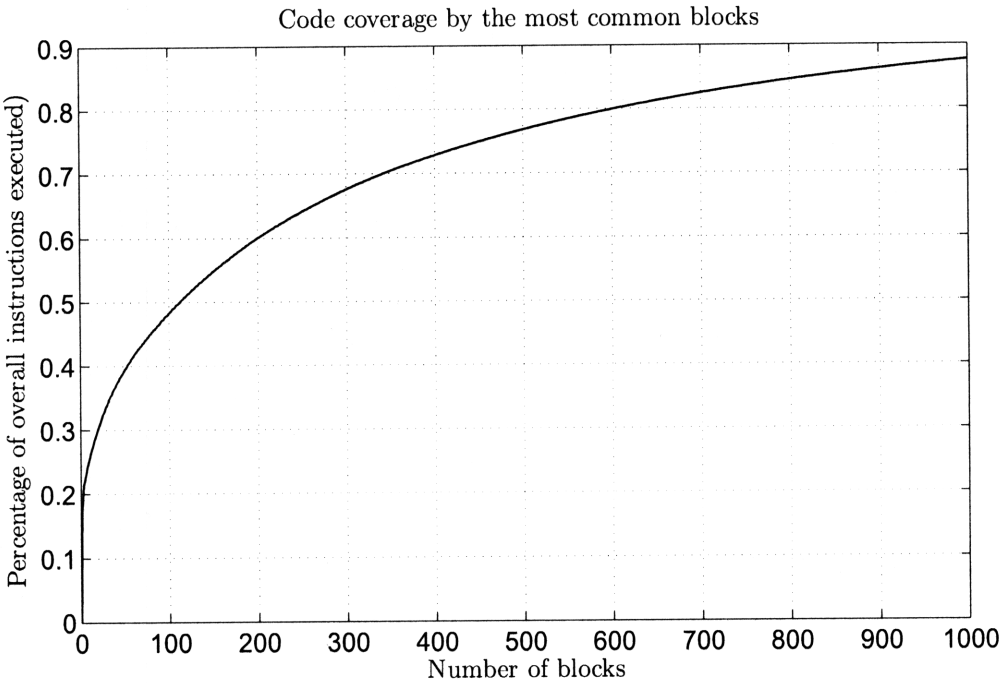

Figure 5-1: Percent code coverage, top 1,000 blocks out of 14,400 (`boothalt32`)

binary translation, the majority of guest kernel code is run infrequently, with only small portions of code being frequently exercised. These sections hit the JIT threshold very quickly, so they are binary translated and execute more efficiently. Although most of the code is interpreted, the execution time of each benchmark is so dominated by a small working set which has been binary translated.

Finally, we conclude this section with a brief discussion of the limitations of the JIT technique. Our tests were run on very specific cases intended to benchmark only specific processes. In a more typical scenario, many different processes are being run simultaneously. Like other JIT framesworks our technique would exhibit incur significant overhead in what can be considered as the "priming" stage or initial stage of a program execution. Furthermore, under the given scheme, a real-world process that is run for long enough will eventually push many blocks over the JIT threshold, increasing memory usage. In these cases, if continual interpretation of infrequently run code is desired, a slight modification would involve periodically resetting the JIT counters for those blocks. We have not explored the heuristics and performance for

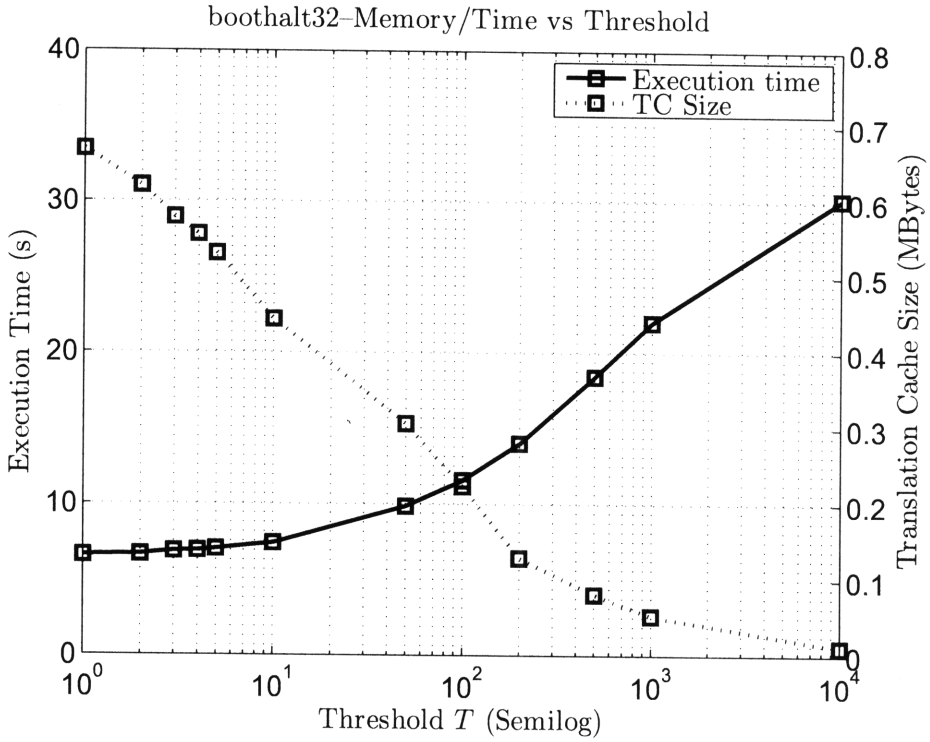

Figure 5-2: `boothalt32` Benchmark Result, Semilog Plot

such a modfification.

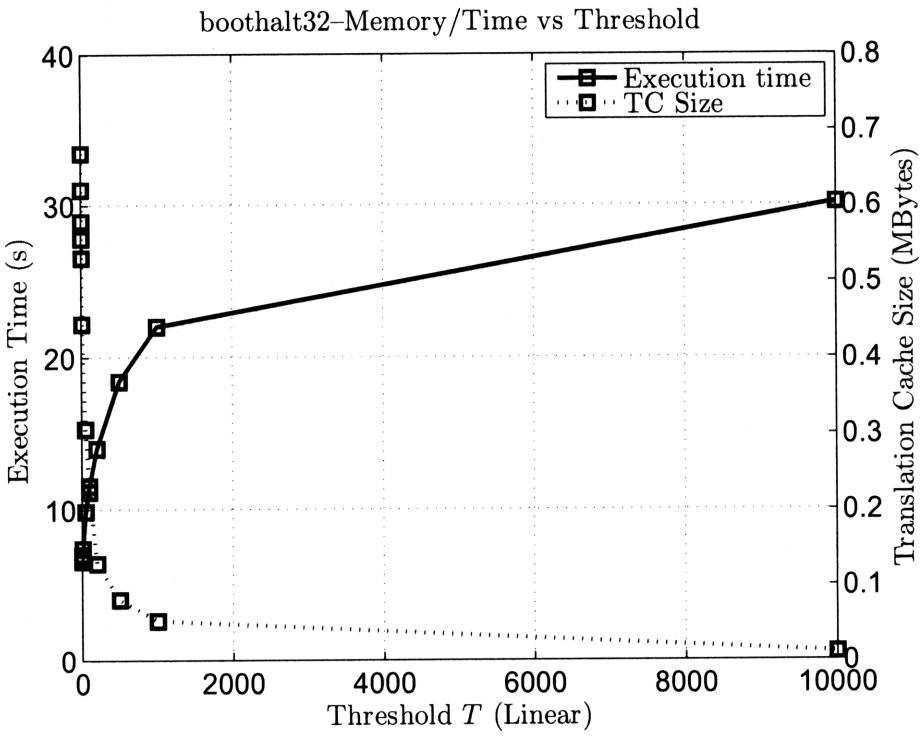

Figure 5-3: `boothalt32` Benchmark Result, Linear Plot

T	Time (s)	Slowdown	TC Size (MB)	% Size Reduction
BT_{ref}	6.25	1.00x	1.10	0
1	6.58	1.05x	0.66	39
2	6.65	1.06x	0.62	43
3	6.87	1.09x	0.57	47
4	6.92	1.10x	0.55	49
5	7.02	1.12x	0.53	51
10	7.40	1.18x	0.44	59
50	9.85	1.57x	0.30	72
100	11.57	1.85x	0.22	79
200	14.02	2.24x	0.12	88
500	18.42	2.94x	0.08	92
1000	22.01	3.51x	0.05	95
10000	30.17	4.82x	0.01	99

Table 5.2: `boothalt32` Benchmark Results

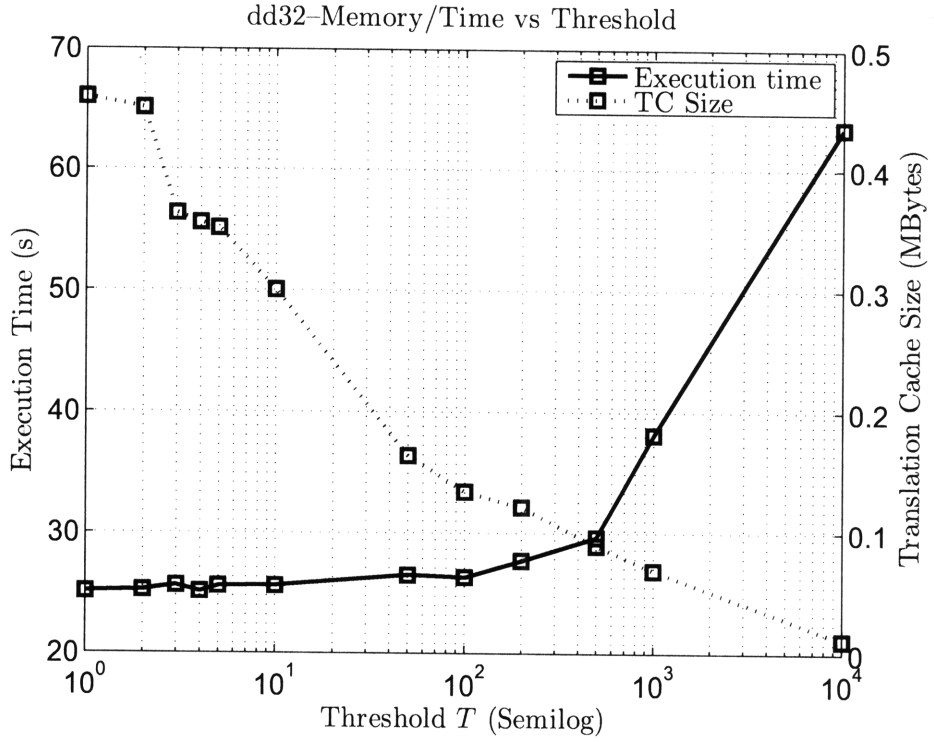

Figure 5-4: `dd32` Benchmark Result, semilog plot

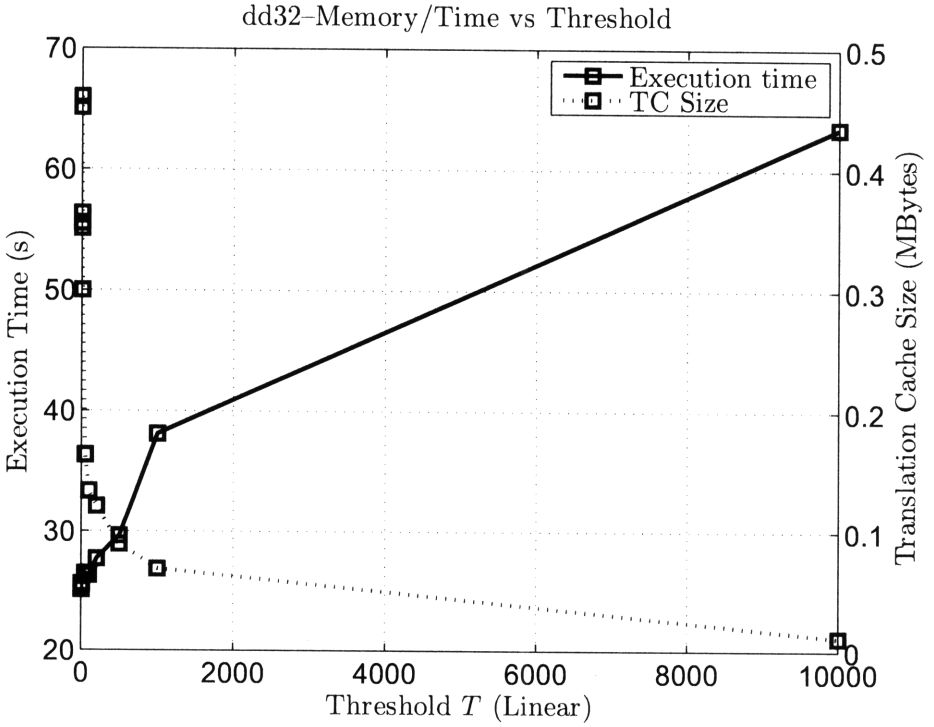

Figure 5-5: `dd32` Benchmark Result, Linear plot

T	Time (s)	Slowdown	TC Size (MB)	% Size Reduction
BT_{ref}	24.73	1.00x	0.51	0
1	25.09	1.01x	0.45	10
2	25.22	1.01x	0.45	12
3	25.61	1.03x	0.36	29
4	25.10	1.01x	0.35	31
5	25.56	1.03x	0.35	32
10	25.58	1.03x	0.30	42
50	26.48	1.07x	0.16	68
100	26.30	1.06x	0.13	74
200	27.69	1.11x	0.12	76
500	29.64	1.19x	0.08	83
1000	38.12	1.54x	0.06	87
10000	63.46	2.56x	0.01	98
∞	568 7	229x	0	100

Table 5.3: `dd32` Benchmark Results

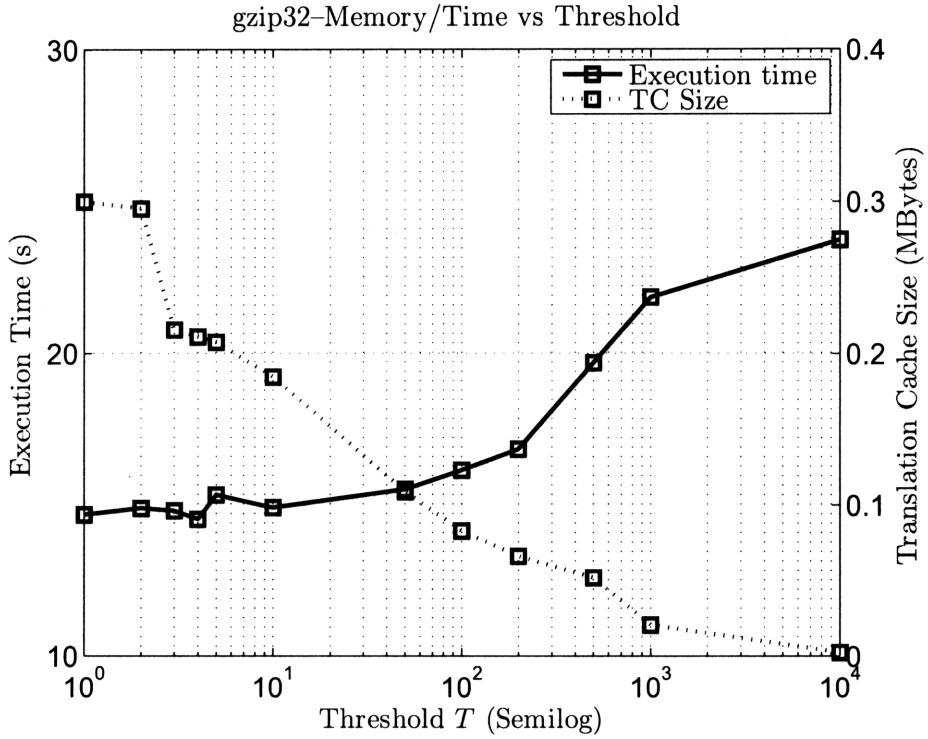

Figure 5-6: `gzip32` Benchmark Results, Semilog Plot

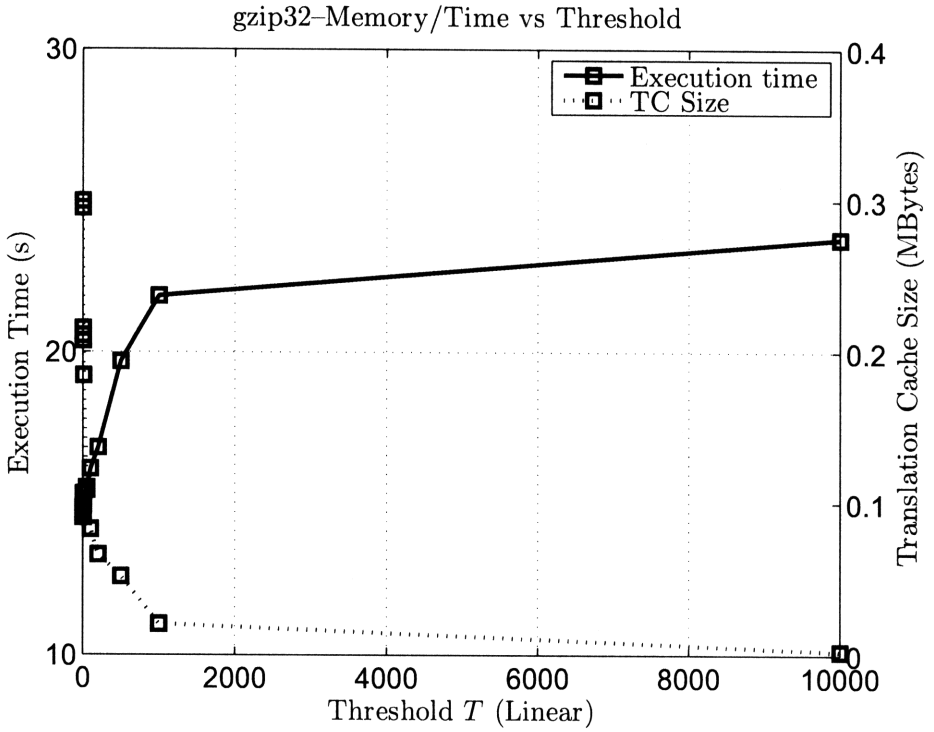

Figure 5-7: `gzip32` Benchmark Results, Linear Plot

T	Time (s)	Slowdown	TC Size (MB)	% Size Reduction
BT_{ref}	14.37	1.00	0.36	0
1	14.66	1.01	0.29	18
2	14.87	1.03	0.29	19
3	14.78	1.02	0.21	41
4	14.51	1.00	0.21	42
5	15.31	1.06	0.20	43
10	14.90	1.03	0.18	49
50	15.50	1.07	0.10	70
100	16.13	1.12	0.08	77
200	16.83	1.17	0.06	82
500	19.68	1.36	0.05	86
1000	21.85	1.51	0.02	94
10000	23.74	1.65	2KB	99

Table 5.4: `gzip32` Benchmark Results

Chapter 6

Conclusion and Future Work

By combining interpretation and binary translation with a simple just-in-time heuristic, this thesis has demonstrated significant reductions in the memory footprint of the Varmosa virtualization system while minimizing performance degradation. In testing, this hybrid intepreter/translator scheme reduced the size of the binary translation cache over a fully binary translated scheme by up to 99% with a slowdown of a factor of between 1.65x and 5x, depending on workload. With only a 10% decrease in performance, upwards of 49% memory reduction was demonstrated.

On current embedded systems, where system memory is likely to fall in the sub 100 megabyte range as opposed to many gigabytes on typical desktop systems, we believe this reduction to be a useful technique, for example in allowing the use of multiple virtual machines per host or reducing the amount of memory required for live migration of virtual machines.

There are many directions to explore with this technique. At the expense of more complicated initial profiling with the interpreter, one could perform a number of optimizations in conjunction with the binary translator. For example, the interpreter could perform register liveness analysis, reducing the number of register spills and fills used by the binary translator, currently a significant source of overhead. The interpreter could also be used to track the number of traps from each block, for example identifying frequently trapping blocks to I/O emulation routines, and inserting automatic hypercalls or backdoors to the appropriate emulation routine. Another class

of possible optimizations are similar to the DynamoRIO type optimizations. The interpreter could again track control flow, giving hints to the binary translator about which blocks are likely to follow each other, and which blocks are likely to be executed often. The binary translator can then perform additional compiler-like optimizations such as rearranging or *chaining* blocks next to each other in memory to improve spatial locality and maximize cache performance. Furthermore, additional exploration into dynamic thresholds for each block or types of instructions in each block should be explored, as well as more extensive testing with a larger set of real-world applications and typical mobile workloads.

Finally, this thesis would like to conclude with an art piece generated with data gathered from a profiling run of Linux. Figure 5-1 represents a control-flow graph of the boot sequence of a Linux guest virtual machine, from the bootloader until it jumps into userspace to a shell prompt. The long tail at the top represents the bootloader. The main mass is the kernel. The little "star" or lobe on the left is made up of interrupt vectors and exception handlers. The graph is constructed by representing each node as a function in Linux, with each node having electrically repulsive charge. Edges represent function calls, and are modeled by springs. The edges are unweighted.

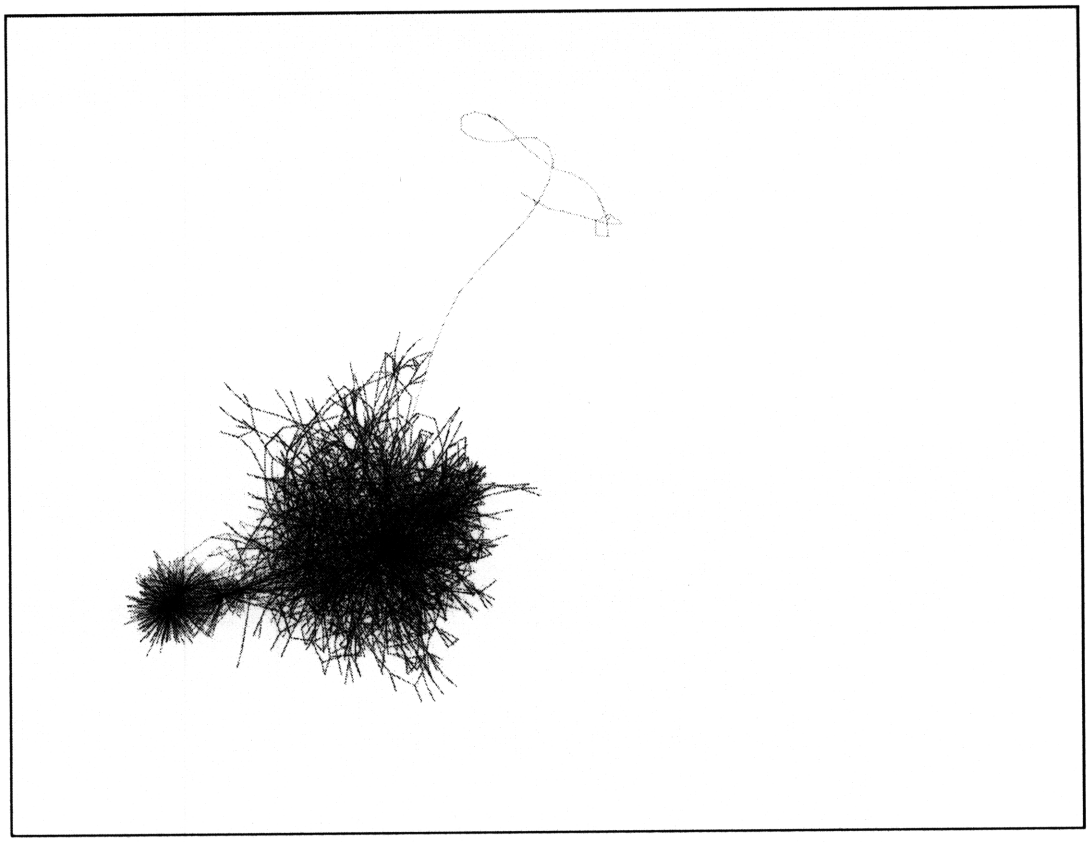

Figure 6-1: Graph of a Linux boot sequence

Appendix A

List of Privileged and Sensitive ARM Instructions

Privileged Instructions

1. `MRS/MSR`: (Write/Read) to/from Coprocessor Registers from/to ARM Registers
2. `MCRR and MRRC`: (Write/Read) to/from Coprocessor Registers from/to two ARM registers.
3. `LDC and STC`: Load/Store from Coprocessor registers from/to memory
4. `CDP`: Coprocessor data operations on coprocessor registers.

Sensitive Instructions

1. `MRS`: Read CPSR/SPSR register
2. `MSR`: Write CPSR/SPSR register (2 subtypes: MSR register and MSR immediate)
3. `CPS`: Change Processor State
4. `DPSPC`: 12 of the 16 Data Processing instructions with the 'S' bit set (MOVS, MVNS, ADDS, ADCS, SUBS, SBCS, RSBS, RSCS, ANDS, BICS, ORRS, and EORS) and writing in to PC

5. `LDM-caret/STM-caret`: Load Multiple / Store Multiple instructions with the 'S' bit set

6. `MOVS PC, ADDS PC, SUBS PC`: arithmetic with PC as target and the 'S' bit set

7. `MRC/MCR User`: Subset of MCR/MRC instructions that are non-privileged, and have different semantics when executed from user-mode directly

Appendix B

List of Control-Flow Changing Instructions

This is the list of ARM instructions that can change the control flow of a program. For reference, see the ARM Architecture Reference Manual[11]

- `B`: Branch to a target address
- `BL`: Branch to a target addres and store the return address in the link register (`R14`)
- `BLX`: Branch to a target address, optionally switching to Thumb mode, preserving the return address in the link register (`R14`)
- `BX`: Branch to a target address held in a register, with an optional switch to Thumb mode
- `LDR`: Load a word from memory into a register, control-changing only if the load is into the program counter (`PC`) or `R15`
- `LDM`: Load multiple words in memory into a set of registers, control-changing only if there is a load is into the program counter (`PC`) or `R15`
- `MOV`: Move a value into a register. Control-changing if the destination register `RD` is the program-counter

– `Data Processing Instructions`: Any data processing instruction with which the destination register `Rd` is the program counter (`R15`). These include `ADD`, `SUB`, `RSB`, `ADC`, `SBC`, `RSC`, `AND`, `BIC`, `EOR`, `ORR`.:w

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. pages 41–46.

[3] Derek L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Cambridge, MA, USA, 2004. Supervisor-Amarasinghe, Saman.

[4] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *ACM Transactions on Computer Systems*, pages 143–156, 1997.

[5] Robert Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[6] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot^{TM}client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[8] International Business Machines. IBM Systems Virtualization, Version 2 Release 1. http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf, 2005.

[9] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[10] Anton Chernoff Ray and Ray Hookway. Digital fx!32 running 32-bit x86 applications on alpha nt. In *in Proceedings of the USENIX Windows NT Workshop, USENIX Association*, pages 37–42, 1997.

[11] David Seal. *ARM Architecture Reference Manual.* Addison-Wesley, second edition, 2001.

[12] Trango Systems. TRANGO The Real-Time Embedded Hypervisor. http://www.trango-systems.com/.

[13] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30, New York, NY, USA, 2008. ACM.

[14] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *In Measurement and Modeling of Computer Systems*, pages 68–79, 1996.