Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

# Pre-Virtualization Compiler Enhancements

Raphael Neider

Diplomarbeit

| | |
|---|---|
| Verantwortlicher Betreuer: | Prof. Dr. Frank Bellosa |
| Betreuender Mitarbeiter: | Dipl.-Inf. Jan Stöss |

31.01.2006

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 31.01.2006

______________________________

Raphael Neider

## Abstract

One problem common to all virtualization techniques is the efficient injection of emulation code into the guest operating systems (guest OSs). Emulation is needed to limit the effects of *virtualization sensitive instructions* to the appropriate virtual machine. For instance, privilege mode changes, processor halting or resetting, and device accesses must be redirected to the executing *virtual* machine rather than affect the underlying *physical* machine, as the latter is probably shared among multiple virtual machines. The emulation code will—in many cases—overwrite the contents of general purpose registers that would remain unmodified by the emulated sensitive instruction. These additional side-effects must carefully be hidden from the guest OS to guarantee its correct execution.

The pre-virtualization approach uses a load-time rewriter to replace sensitive instructions with a sequence of code that saves all caller-saved registers, calls appropriate emulation code, and restores the saved registers afterwards. In many cases, this approach unnecessarily saves many registers whose content is afterwards discarded, thus increasing the virtualization overhead.

In this thesis we propose to use additional context-information for each sensitive instruction to facilitate generating more efficient replacement code: *Live registers* enable us to identify and discard irrelevant registers instead of preserving them across the emulation. Furthermore, literally known register content can help in statically selecting more specific emulation code, especially for device I/O. We also propose to use static rewriting techniques, such as register reallocation and rescheduling, to reduce the number of live caller-saved registers at the sensitive instructions.

All presented techniques will be integrated into an advanced rewriting system, which automatically extracts the required information from the pre-virtualized guest OS binary, performs the supporting static rewriting, and implements efficient load-time rewriting routines.

Our implementation for IA-32 shows that the use of context-information reduces the number of preserved registers during the emulation of sensitive instructions by more than 40 % for current Linux kernels.

# Contents

**1 Introduction** **1**

1.1 The Problem: Missing Context-Information . . . . . 2

1.2 Approach . . . . . 3

1.3 Structure of This Thesis . . . . . 4

**2 Background And Related Work** **5**

2.1 Virtualization . . . . . 5

2.1.1 Terminology . . . . . 6

2.1.2 Simulation and Interpretation . . . . . 6

2.1.3 Pure Virtualization . . . . . 7

2.1.4 Para-Virtualization . . . . . 7

2.1.5 Pre-Virtualization . . . . . 8

2.2 Binary Rewriting . . . . . 10

**3 Proposed Solution** **13**

3.1 Terminology . . . . . 13

3.2 Optimizations for Individual Patches . . . . . 14

3.2.1 Discarding Irrelevant Registers . . . . . 14

3.2.2 Reducing the Overhead of Callee-Saved Registers . . . . . 16

3.2.3 Statically Evaluating Dispatch Tables . . . . . 19

3.3 Optimizations for Successive Patches . . . . . 20

3.3.1 Deferring Register Restoration . . . . . 20

3.3.2 Rescheduling the Instruction Stream . . . . . 22

3.3.3 Compacting Basic Blocks . . . . . 22

3.4 Context-Aware Rewriting System . . . . . 23

3.4.1 Goals of Our Design . . . . . 24

3.4.2 System Structure . . . . . 25

3.4.3 Sources of Context-Information . . . . . 26

3.4.4 Analysis Component . . . . . 27

3.4.5 Rewriter Component . . . . . 28

**4 Implementation** **29**
4.1 Rewriter Component . . . . . 29
4.1.1 General Approach . . . . . 30
4.1.2 Efficient Handling of Caller-Saved Registers . . . . . 30
4.1.3 Static Dispatch . . . . . 33
4.1.4 Deferred Register Restoration . . . . . 34
4.1.5 Compaction of Basic Blocks . . . . . 35
4.2 Glue Code Generators . . . . . 36
4.2.1 Specification Language for Glue Code . . . . . 36
4.2.2 Generated Glue Code . . . . . 37
4.2.3 Locally Modified Calling-Conventions . . . . . 38
4.3 Analysis Component . . . . . 39
4.3.1 Overview . . . . . 39
4.3.2 Code Discovery . . . . . 41
4.3.3 Disassembly . . . . . 43
4.3.4 CFG Reconstruction . . . . . 44
4.3.5 Data-Flow Analysis . . . . . 45
4.3.6 Static Rewriter . . . . . 48

**5 Evaluation** **53**
5.1 Test Environment . . . . . 53
5.2 Microbenchmarks . . . . . 54
5.2.1 General Virtualization Code . . . . . 54
5.2.2 Unused Scratch Space . . . . . 56
5.3 Applicability . . . . . 57
5.4 Minibenchmarks . . . . . 61
5.4.1 Latency Tests . . . . . 61
5.4.2 Bandwidth Tests . . . . . 63
5.5 Execution Profiles . . . . . 64

**6 Conclusion** **67**
6.1 Summary . . . . . 67
6.2 Open Issues . . . . . 68
6.3 Outlook . . . . . 68

**A Examples of IA-32 Glue Code** **69**
A.1 Original Calling Conventions . . . . . 69
A.2 Efficient Calling Conventions Using Stubs . . . . . 71
A.3 Inlined Stubs . . . . . 72

# Chapter 1

# Introduction

Virtualization technology enables the concurrent execution of multiple operating systems (OSs) on a single physical machine. To safely isolate the operating systems from each other, virtualization environments confine their guest OSs in virtual machines and control their execution with virtual machine monitors (VMMs). Virtual machines can be hosted on general purpose operating systems or on specialized *hypervisors*, which provide services such as physical memory management and scheduling for their clients. The basic structure of a hypervisor-based system is outlined in Figure 1.1.

Most of the instructions of the virtual machines are passed through to the physical machine for direct execution. Some instructions, however, access the hardware in a way that might affect other concurrently running virtual machines or even break the control of the VMM over the guest OS. The effects of these *sensitive instructions* must therefore be redirected to software-emulated or virtualized hardware [27], which is usually provided by the VMM or the hypervisor. The approaches to discovering sensitive instructions and redirecting their effects vary between virtualization techniques.

One approach is to use *pre-virtualization*: In a preparation phase the sensitive instructions in the guest OS binary are first padded to provide scratch space for emulation code and then annotated to ease the process of finding them during later rewriting. While loading the guest OS into a virtual machine, a rewriter replaces all sensitive instructions with appropriate emulation code or a call to an emulation routine in the VMM or the hypervisor.

Care must be taken not to expose side-effects of the emulation to the guest OS. This is important, because the emulation code is executed on the same processor and in the same context as the guest OS, which implies shared registers between guest OS and emulation code. If the content of a register is still required after the sensitive instruction (respectively its emulation) has been executed, the rewriter must guarantee that the emulation does not overwrite this particular register. If the rewriter cannot guarantee this, it must save and restore the content of the register as part of the emulation.

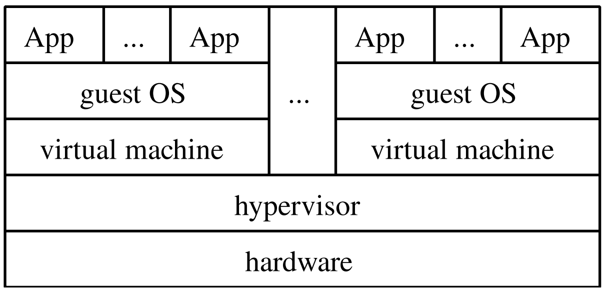

Figure 1.1: Structure of a virtualization environment. The operating systems and their applications are executed and confined in virtual machines. The virtual machines can access critical system hardware only via the hypervisor, which also provides physical memory management and basic scheduling for its client virtual machines. (system structure adopted from [1])

## 1.1 The Problem: Missing Context-Information

For convenience, the emulation routines provided by the VMM are usually compiled from a high-level programming language, such as C or C++. Thus we have only limited influence on the registers that are used and modified, but can rely on the calling-conventions of the compiler. These include a logical split of the register set into caller-saved and callee-saved registers: The *callee-saved* registers are guaranteed to retain their value during execution of the called subroutine; if the callee wants to access them anyway, it has to restore their original value before returning to the caller. For *caller-saved* registers, no such guarantee is made: the callee is free to modify them at will.

During replacement of the sensitive instructions, previous rewriters do not possess any information about which of the caller-saved registers are actually modified by the called emulation routine. Additionally, they do not “know”, which of these registers actually need to retain their contents because they are read later on by the guest OS. Currently, the rewriter conservatively approximates the missing information according to worst-case assumptions:

- all registers are modified by the called emulation routine
- all registers contain values that are accessed again afterwards

As a consequence, the rewriter replaces each sensitive instruction with a code sequence that first stores *all* caller-saved registers (in memory), then calls the desired emulation routine, and finally restores the previously saved registers. We depict this approach in Figure 1.2.

Our analysis (Section 5.3) shows that, in about 75 % of all cases, not all callee-saved registers need to be saved, because only a subset of them is actually used later on. For each

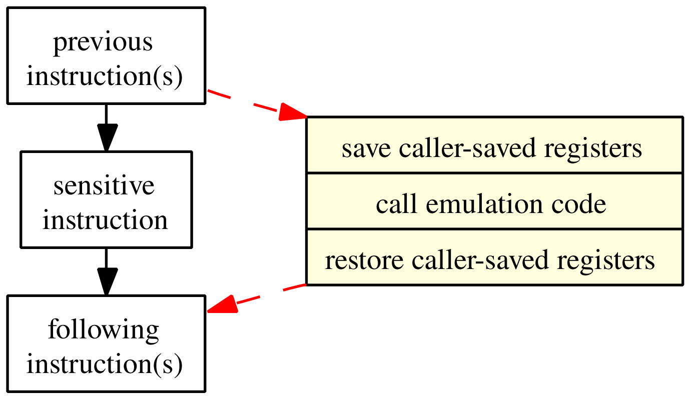

Figure 1.2: Virtualized sensitive instruction. The rewriter substitutes sensitive instructions with virtualization code; the latter often includes a call to an appropriate emulation routine. Calling conventions allow the called function to modify caller-saved registers. The virtualization code must hide these modifications from the guest OS unless the registers are not read afterwards.
Solid edges represent the original control-flow, dashed edges represent the control-flow after the sensitive instruction has been replaced with the virtualization code (right box).

unused register the current approach wastes two instructions[1] and two memory accesses. Estimating a mean number of 50 executed instructions per emulation, each unnecessarily preserved register adds 4 % to the overall virtualization overhead, which we want to avoid.

## 1.2 Approach

To improve the rewritten code, we deliver information about the contexts of each individual sensitive instruction to the rewriter. Based on this information, an enhanced rewriter then generates custom-tailored code that only saves the relevant registers. Additional context-information can be used to statically select more specific emulation code instead of having a dynamic dispatcher select it at runtime, and thus reduce the number of costly control-flow operations.

We develop a system that automatically computes all required information solely based on the pre-virtualized guest OS binary. Together with an enhanced rewriter, this system facilitates efficient rewriting at load time. We verify the effectiveness of our proposals with an implementation for the common IA-32 systems. Measurements on a Pentium 4 reveal that the overall system performance is hardly affected by our optimizations,

[1]Two instructions are needed to either `push` and `pop` the register on and off a stack or to `store` it to a dedicated memory location and `load` it from there afterwards. For architectures without `push` and `pop` instructions, even more code might be needed to update the stack pointer if no dedicated memory can be used—for example in multi-processor systems.

although more than 40 % of the previously preserved caller-saved registers are discarded. Most of the benchmarks we conducted on a pre-virtualized Linux show insignificant improvements of round about 0.5 % due to our optimizations.

## 1.3 Structure of This Thesis

The remainder of this thesis is structured as follows:

Chapter 2 relates this work to previous research in the fields of virtualization techniques and binary rewriting. We point out differences between previous work and our system and show how this work is influenced by formerly published results. In Chapter 3 we present our ideas on improving the load-time rewriting process with context-information. All developed techniques will finally be integrated into an enhanced rewriting system, which automatically extracts the required context-information from the guest OS and makes use of it in a context-aware rewriter. Chapter 4 provides details on our implementation of both the enhanced rewriter and the analysis of IA-32 guest OS binaries, which obtains the context-information to support the rewriter. In Chapter 5 we prove the effectiveness of the proposed techniques with microbenchmarks and their applicability in a pre-virtualized Linux kernel with figures we obtained during rewriting. In this chapter we also present benchmark results that compare the original and our optimized approach to rewriting. Chapter 6 summarizes and concludes this thesis and hints at possible future work based on our results.

# Chapter 2

# Background And Related Work

In this chapter we provide background information on three current approaches to virtualize IA-32 systems. We will then present related work from the research field of binary rewriting.

## 2.1 Virtualization

The basic idea of virtualization is to enable the concurrent execution of multiple operating systems on a single physical hardware platform while providing strong isolation of these guest operating systems. This is similar in spirit to multi-tasking operating systems, which allow for the concurrent execution of multiple user-level applications on a single instance of an operating system. Virtualization is, however, conceptually located one layer below such systems. Where multi-tasking operating systems can define and expose a software API that can be tailored to address the problems of multi-programming, virtualization is bound by the underlying hardware architecture, which is externally defined and rather fixed.

Further problems arise from the nature of the guest operating systems: For one, these expect to execute in the most privileged processor mode. Secondly, they are designed to exert direct control over the hardware but not to share it with concurrently executing operating systems. In a virtual machine, the guests are deprivileged and restricted to accessing the provided virtual hardware in order to reliably control them and to guarantee strong isolation of the concurrently running guests. As a consequence, instructions that rely on the above assumptions must be intercepted and emulated; however, the major part of the instructions of the guests will be executed natively on the underlying physical machine.

### 2.1.1 Terminology

In this thesis we will use the following terms with respect to virtualization: The virtualization environment provides a set of *virtual machines* (VMs), which are software controlled abstractions of the underlying physical machine. We call the software that implements the virtual machines a *virtual machine monitor* (VMM). A *hypervisor*[1] multiplexes the hardware among the virtual machines—that is, the hypervisor provides management of resources such as memory or processor time for the virtual machines. The general structure of such a virtualization environment is shown in Figure 1.1, page 2.

In order to successfully contain the *guest operating system* in its virtual machine, we must prevent *(virtualization) sensitive instructions* from affecting the shared hardware. Instead of executing these instructions natively, we must therefore replace them with VMM-specific emulation code that applies the effects to the proper virtual machine. Sensitive instructions comprise instructions that change the processor's mode of execution—for example change the current privilege level, enter system management mode, halt or reset the processor—or access the now possibly shared devices via port I/O. In addition to sensitive instructions, we also need to intercept *sensitive memory operations* [14], which read or modify hardware-accessed data structures, such as page-tables or interrupt descriptor tables, or trigger memory mapped I/O.

During execution of a guest OS in a virtual machine, the effects of all the sensitive instructions and sensitive memory operations must be redirected to the executing virtual machine rather than be applied to the physical machine. For this purpose, the VMM provides *emulation code*, which is executed instead of the original sensitive instruction. Because the emulation code is usually executed on the same processor that also runs the guest OS, all side-effects of the emulation on the machine state, especially its registers, must be hidden from the guest OS. We shall refer to *any* code that is substituted for a sensitive instruction as *virtualization code*; typically, virtualization code prepares the execution of emulation code and cleans up afterwards in order to hide the emulation from the guest OS.

The remainder of this section gives an overview of different approaches to virtualization together with a summary of their particular strengths and weaknesses.

### 2.1.2 Simulation and Interpretation

Whole-system simulators, such as QEMU [2], bochs [12], or Virtual PC [17], emulate not only virtualization sensitive instructions but the complete instruction set. This enables the execution of guest operating systems that are designed and implemented for hardware architectures different from the one that executes the simulator. Such a flexibility is paid

[1] For the purpose of this thesis, it does not matter whether the virtual machines are hosted on a complete operating system or on a hypervisor. We therefore restrict our discussion to the second approach.

for with an enormous emulation overhead at runtime; we consider the latter to be too high for efficient virtualization.

A similar albeit much more efficient and less general approach to abstracting from the physical hardware has gained popularity with Sun's Java [15] and Microsoft's more recent .NET technology [5]. Both systems provide a runtime environment for the execution of just-in-time compiled byte-code. The runtime is commonly referred to as a virtual machine, which is conceptually an interpreter for the respective byte-code.

Neither technique is suitable for virtualization of IA-32 systems, because these virtual machines do not reflect the IA-32 hardware interface but implement a new "architecture". We know of no efforts to port a current operating system from IA-32 to either Java or .NET. However, the byte-code approach could be adopted to convey detailed information about the contexts of sensitive instructions to the load-time rewriter in our system.

### 2.1.3 Pure Virtualization

Virtual machines as provided by VMware [25] execute on top of a common desktop operating system such as GNU/Linux or Windows. Non-sensitive instructions are passed through to and executed unmodified by the underlying hardware, whereas sensitive instructions are detected and emulated using *binary translation* [21]: In this process the guest OS is automatically ported to a new architecture, which mostly resembles the original IA-32, but lacks the sensitive instructions. All sensitive instructions in the guest OS are implicitly translated to emulation code for the provided virtual machine. As the emulation code is automatically injected into the guest OS, nearly all operating systems can run unmodified in the provided virtual machines. Only the virtual machine monitor (VMM) must occasionally be improved to correctly handle new instructions and hardware features.

On the downside, this approach implies considerable overhead: For one, the larger part of the emulation is performed not within the guest's protection domain, but in a kernel module in the host OS. This necessitates frequent and costly privilege level changes for the emulation of sensitive instructions. Secondly, memory sensitive operations must not only be emulated, but also be detected at runtime. For this purpose, the translator inserts *guard instructions* before every memory access in order to discover references to sensitive data structures such as page-tables or interrupt descriptor tables. As an alternative, the hardware memory management mechanisms segmentation and paging can be utilized to intercept such accesses via segmentation violation or page fault exceptions. Both approaches introduce significant overhead.

### 2.1.4 Para-Virtualization

The para-virtualization technique [26], which is implemented for example in the Xen hypervisor [1] or L4Linux [7], abandons the need to detect and insert emulation code for sensitive instructions at runtime. Instead, this approach provides a high-level, function-call based interface to its emulation routines, which enables virtualization with a very low overhead for two reasons: First, the virtual machine monitor (VMM) is *collocated* with the guest OS—that is, VMM and the guest OS reside in the same address space. Therefore, execution of emulation code often does not require privilege level or even address space changes but only a simple function call. The high-level interface allows to reduce the number of privilege level changes even further by batching several calls to the hypervisor and having all of them handled upon the next unavoidable entry into the hypervisor. Second, the virtualization code is generated by the optimizing compiler that is used to build the guest OS; thus its integration into the guest is automatically optimized as well (as opposed to pre-virtualization, for which this thesis strives to provide optimized virtualization code).

To make use of the new interface, a potential guest operating system must first be ported to this new "architecture" before it can be loaded into the virtual machine. During porting, all sensitive instructions must be removed from the source code of the guest operating system and be replaced with calls to their respective emulation routines in the VMM. Besides source code access, porting an operating system requires intimate knowledge of the ported OS and leads to high engineering cost both for the initial porting and the later maintenance of the guest OS.

Despite its good performance, we believe that the long-term engineering effort, which is caused by porting and maintaining specialized guest OS versions, is too high to make para-virtualization practical on the long run. Our approach will enable the use of optimizing compiler techniques for the generation of virtualization code and thus help to close the performance gap between para- and pre-virtualization—the latter of which we will introduce now.

### 2.1.5 Pre-Virtualization

Pre-virtualization was developed both to overcome the high engineering cost of para-virtualization while keeping up its efficiency and to enable the binary to execute on a variety of hypervisors—or even on raw hardware.

In [14], LeVasseur and colleagues state that, although still minor changes to the guest OS are necessary, the engineering effort is an order of magnitude lower than for plain para-virtualization. The authors report that they can automatically prepare the guest OS for virtualization by using assembler macros to detect, annotate, and pad sensitive instructions for later rewriting. This preparation is integrated into the build process of the guest OS and therefore requires source code access. The resulting binary still obeys the hardware interface; consequently it can be executed on bare hardware and on pure virtualization systems. Additionally, pre-virtualization–aware hypervisors can efficiently instantiate a virtualized version that is targeted for themselves using load-time rewriting: While the guest OS is loaded into the virtual machine, the rewriter uses the provided annotations to easily find and replace the sensitive instructions with inlined emulation code or calls to emulation routines in the collocated VMM. The rewriting process solely relies on information from the binary and does no longer require access to the source code of the guest OS. Consequently, OS vendors need to maintain only a single pre-virtualized binary distribution of their operating systems, which can than efficiently be executed on the targeted hardware as well as in a variety of virtualization environments. The source code of the OS need not be published.

Compared with pure virtualization, the injection of virtualization or emulation code is considerably simplified, because the preparation step provides enough scratch space to hold the injected code. Consequently, no instructions need to be relocated as no code is added to and inserted into the guest OS, so that global rewriting or translation to adjust the references is not required.

In the current design, the mapping from sensitive instruction to virtualization code is fairly static: All occurrences of a sensitive instruction are replaced with identical copies of virtualization code—except for adjustments to the operands to match the original sensitive instruction. This results in inefficient code being emitted for sensitive instructions that are virtualized by calling emulation routines in the VMM, because the rewriter must make sure to preserve all caller-saved registers: Each register *might* need to be preserved at some call-site, although at most call-sites only a subset of the caller-saved registers actually needs to be preserved.

An approach similar to pre-virtualization was presented by Eiraku and colleagues in [6] to run BSD systems as Linux applications. They propose to use an assembler preprocessor to automatically replace the sensitive instructions with their respective virtualization code. As the binary is statically rewritten, this approach tightly couples guest operating system and the targeted "hypervisor", quite similar to para-virtualization. No details about the assembler preprocessor are published, so we assume that it is a mere macro expander. As a consequence, this approach could as well benefit from the results of this thesis by using are more powerful, context-aware preprocessor.

## Analysis

The presented virtualization techniques all exhibit the classic cost vs. efficiency trade-off: While simulation and pure virtualization are cheap in terms of guest OS preparation cost, they imply a considerable runtime overhead. Para-virtualization reduces the runtime overhead of virtualization at the cost of a tremendous engineering effort for preparing the guest OS. The pre-virtualization approach sacrifices some of the performance of para-virtualization its simple rewriter, but greatly reduces engineering cost again. We believe that we can close the performance gap between pre- and para-virtualization with an improved rewriter without reintroducing any additional engineering effort. For this purpose, we will provide context-information for the sensitive instructions to the rewriter and generate more efficient virtualization code, which respects the specific context of the currently rewritten instruction.

## 2.2 Binary Rewriting

The use of context-information, such as live registers, for efficient code generation is common practice in modern compilers [18]. Furthermore, computing this information via data-flow analysis on binaries has also been proposed before and is implemented in a variety of tools. However, we are not aware of any system that implements the analysis and the rewriter in separate tools, as we will do. Most available approaches also rely on additional meta-data, such as debugging symbols or relocation records, to help them in locating code and data in the binaries. Our system will use such information if available, but also work just fine without it.

Binary instrumentation, implemented in tools like ATOM [22] or FIT [4], is similar in spirit to the pre-virtualization load-time rewriting: Both approaches inject foreign code into a readily compiled program without disturbing its execution besides timing behaviour. ATOM and FIT already use live register information to efficiently hide the side-effects of the injected code from the instrumented program. The current load-time rewriter lacks this information and consequently emits inefficient code.

More general binary rewriting tools, such as the Executable Editing Library EEL [11], grant even more access to the code of completely linked binaries. For this purpose, these tools extract a control-flow graph (CFG) representation of the binary and allow to insert or remove instructions on this graph. The modified program can afterwards be emitted as a runnable executable. EEL even provides static program analysis, including live registers, and would therefore be a good starting point for the task of this thesis. Unfortunately, EEL is only available for SPARC-like RISC architectures. As virtualization is primarily interesting for the common CISC IA-32, which introduces many problems like variable instruction lengths that are not addressed by EEL, neither the library itself nor the implemented algorithms can easily be adopted for our purposes.

However, similar tools exist for IA-32: DIABLO [24] is a link-time rewriter, which is available for IA-32 and several other architectures. Being a link-time rewriter, DIABLO does not operate on the completely linked binary but takes the relocatable object files as input. This precludes its use on pre-virtualized guest OS binaries without source-code access. Furthermore, the applicability of this system to operating system kernels is unknown, as it was developed to rewrite user-level applications.

### Analysis

Several tools for efficient rewriting of IA-32 binaries exist, but all of these are intended to *statically* rewrite the binaries. To facilitate the insertion of foreign code, the tools must precisely identify code and data, so that they can correctly adjust references to the following, relocated code. As code and data are inherently indistinguishable in IA-32 binaries, all present tools require additional information about the structure of the binary and therefore operate on relocatable object files. These provide the required information in terms of relocation records.

In this thesis, however, we want to rewrite pre-virtualized guest OS binaries without relying on additional information. This is possible, because we do not rewrite the code globally, but apply only local modifications to replace sensitive instructions. Consequently, we do not need to globally update address references, which relieves us from the burden of having to precisely identify code and data. We only need to find all code fragments, so that the contexts of the sensitive instructions can correctly be determined. For the actual computation of the context-information, we can use a number of standard data-flow analysis [19], which are also implemented in the previously presented tools.

# Chapter 3

# Proposed Solution

In this chapter we will present an enhanced rewriting system for pre-virtualized operating systems on IA-32. First we propose techniques that overcome the inefficiencies of previous rewriters by considering additional information about the contexts of sensitive instructions during their rewriting. Afterwards, we present approaches to amortize some of the virtualization overhead across successive rewritten instructions. The proposed optimizations reduce the number of instructions that are substituted for the sensitive instructions. Our final contribution shall therefore enable us to efficiently skip the unused space.

As the presented optimizations require detailed information about the contexts of the rewritten instructions, we shall conclude this chapter with a presentation of our analysis tool. This tool extracts the required information from the guest OS binary to support the enhanced rewriter, which implements the above techniques. The rewriter and the analysis tool together make up our enhanced rewriting system.

## 3.1 Terminology

Throughout the remainder of this work we will use the following terms to refer to well-defined objects: To keep the virtual machines isolated from each other, we must redirect the effects of sensitive instructions in a guest operating system (guest OS) to its assigned virtual machine. The *emulation code*, which implements this functionality, comes in two guises: For most instructions, the virtual machine monitor (VMM) provides appropriate emulation code in terms of compiler-generated subroutines (external emulation). For simple but frequently executed instructions, the rewriter inlines the emulation code into the guest OS (inlined emulation).

Especially for external emulation, we often require additional code to (a) compute arguments for the emulation, (b) hide side-effects of the emulation from the guest OS, (c) implement the call of the emulation routine, and (d) clean up the argument stack afterwards. We use the term *virtualization code* to refer to these code fragments. Again, virtualization code can either be completely inlined or (partially) be deferred to helper routines. We call the latter *stubs*, as they abstract from the details of the actual call similar to stubs in remote procedure calls [3].

To simplify the injection of inlined emulation or virtualization code into the guest OS, we prepare the latter during assembly: A set of assembler macros pads each sensitive instruction with an instruction-dependent number of semantically neutral `nops` to provide *scratch space*. The rewriter can use this space in addition to the space previously occupied by the original sensitive instruction to store the generated code. We subsume the sensitive instruction and its dedicated scratch space in a *patch*. More global optimizations not only consider the current patch, but also the surrounding *basic block*, which is a straight-line sequence of instructions without jumps or jump targets in between. Only the last instruction in a basic block may be a branching instruction.

## 3.2 Optimizations for Individual Patches

Previous rewriters with static virtualization code must save and restore all registers that might be modified by the emulation code at every rewritten instruction, because their contents might still be needed after some of these instructions. Our observation is, however, that in most cases only a subset of the registers is in fact relevant for the subsequent execution of the guest OS. We therefore propose to dynamically generate the virtualization code according to the context of the currently rewritten instruction. During rewriting, the irrelevant registers can then safely be discarded, if the rewriter can distinguish relevant from irrelevant registers.

### 3.2.1 Discarding Irrelevant Registers

We shall now present a formal approach to computing the minimal set of registers, whose contents must be preserved across the emulation of a sensitive instruction $i$. To avoid confusion, we will subsequently use *instruction* to refer to a particular instance of an instruction—that is, a single occurrence of its encoding in the examined binary—and not to all instructions with the same opcode or type.

#### Relevant Registers

We call a register *relevant* at the instruction $i$ if its content from before the execution of $i$ is possibly read afterwards. Intuitively, the set of all relevant registers is the minimal set of registers that must retain their original value after the execution of the virtualization code for $i$.

To formalize this definition, let $Assigns(i)$ denote the set of registers that are potentially modified during the non-virtualized execution of $i$. Let us further refer to the set of live registers[1] immediately *after* the execution of $i$ as $\mathit{LiveAfter}(i)$. We then define

$$\mathit{Relevant}(i) := \mathit{LiveAfter}(i) \setminus \mathit{Assigns}(i). \tag{3.1}$$

The set $\mathit{Relevant}(i)$ is in fact the earlier mentioned set of *relevant registers at* $i$*:*

"$\subseteq$": Let $r \in \mathit{Relevant}(i)$ be a register. From Equation 3.1 follows that $r \in \mathit{LiveAfter}(i)$ and $r \notin \mathit{Assigns}(i)$. From the definition of live registers we can conclude that $r$ is accessed after $i$ has been executed; furthermore, the content of $r$ is not changed by the execution of $i$, as $r \notin \mathit{Assigns}(i)$. Consequently, $r$'s content from before the execution of $i$ is accessed afterwards—$r$ is a *relevant register* at $i$.

"$\supseteq$": To make $r$ a *relevant register*, its value prior to the execution of $i$ must be accessed after $i$'s execution. In other words: $r$ is live after $i$ has been executed and $r$ is not modified during the execution of $i$. This implies $r \in \mathit{LiveAfter}(i)$ and $r \notin \mathit{Assigns}(i)$, thus $r \in \mathit{LiveAfter}(i) \setminus \mathit{Assigns}(i)$ and finally $r \in \mathit{Relevant}(i)$. □

## Saved Registers

Not all relevant registers need to be preserved across the emulation (let us refer to the code that is used to emulate $i$ as $\mathit{Emulate}(i)$): We can safely ignore registers that are known to remain untouched by the emulation code. As we provide this code, we can also provide the set of registers that are potentially modified during its execution; let $\mathit{Assigns}(\mathit{Emulate}(i))$ denote this set.[2] As a consequence, we need to save and restore all registers $r$ with

$$r \in \mathit{Relevant}(i) \cap \mathit{Assigns}(\mathit{Emulate}(i)).$$

All other registers can safely be discarded.

If the emulation code is provided as a compiler-generated subroutine in the VMM—which is the common case—, the task of preserving register contents is partially deferred to the emulation code due to compiler-enforced calling conventions: For subroutine calls the register set of the processor is logically split into caller-saved and callee-saved registers. *Caller-saved* registers may be left modified by the called subroutine; the caller has to save and restore them if their contents is still needed. The usually larger part of the register set is *callee-saved*; the called subroutine must guarantee that the contents of callee-saved registers appears to be unchanged by its execution.

[1] A register is commonly considered to be *live* if its current value is *potentially* read by one of the following instructions.

[2] For the sake of readability, we implicitly extend our functions from the domain of single instructions to the domain of sets of instructions in the natural way: For arbitrary code $c$—represented as a set of instructions—we define $\mathit{Assigns}(c) := \bigcup_{i \in c} \mathit{Assigns}(i)$.

Consequently, the minimal set $\mathit{Saved}(i)$ of registers that need to be saved and restored by the generated virtualization code is given as

$$\mathit{Saved}(i) := \mathit{Relevant}(i) \cap \mathit{Assigns}(\mathit{Emulate}(i)) \cap \mathit{Callersaved}.$$

### 3.2.2 Reducing the Overhead of Callee-Saved Registers

In the previous section we have defined the set $\mathit{Saved}(i)$ to help in generating efficient code for caller-saved registers. However, the callee-saved registers are still saved unnecessarily if they are not relevant at the call-site: The subroutine must work correctly in all contexts, so that the compiler approximates the combined $\mathit{Relevant}$ set of all call-sites with the set of all registers. With $s$ denoting the set of instructions that make up the subroutine, this results in the generation of code to save and restore

$$\{\text{all registers}\} \cap \mathit{Assigns}(s) \cap \mathit{Calleesaved} = \mathit{Assigns}(s) \cap \mathit{Calleesaved},$$

although for each call, only the subset that is relevant at the current call-site $c$ effectively needs to be preserved:

$$\mathit{Relevant}(c) \cap \mathit{Assigns}(s) \cap \mathit{Calleesaved}$$

We will now present three approaches to counter the negative effects of unnecessarily preserved callee-saved registers on overall virtualization performance.

#### Storing Caller-Saved Registers in Callee-Saved Registers

Our first approach is to preserve relevant caller-saved registers not in memory but in irrelevant callee-saved registers. This effectively makes some irrelevant callee-saved registers relevant, because their values are used after returning from the emulation routine to restore the originally relevant caller-saved registers. Furthermore, less registers are saved in memory, which reduces the number of memory accesses in favour of register-to-register copies. Additionally, the presented approach reduces the stack footprint of the emulation, which might improve cache effectiveness and therefore indirectly increase overall performance. We depict this technique in Figure 3.1.

The main advantage of this approach is its general applicability: Whenever callee-saved registers are considered irrelevant while at least one relevant caller-saved register exists, we can save two memory accesses—storing and restoring the caller-saved register—by preserving the relevant caller-saved in the irrelevant callee-saved register.

On the downside, this approach does not reduce the number of instructions that are necessary for saving the register state. Its direct effect on the virtualization runtime is therefore rather negligible.

| (a) original virtualization code | (b) improved virtualization code |
|---|---|
| `pushl %eax` | `movl %eax,%esi` |
| `pushl %ecx` | `movl %ecx,%edi` |
| `pushl %edx` | `pushl %edx` |
| `call emulation` | `call emulation` |
| `popl %edx` | `popl %edx` |
| `popl %ecx` | `movl %edi,%ecx` |
| `popl %eax` | `movl %esi,%eax` |

Figure 3.1: Caller-saved registers preserved in callee-saved registers. Relevant caller-saved registers can be protected from modification by the callee by moving them into irrelevant callee-saved registers before the call and restoring them from there afterwards. In this example we assume %EAX, %ECX and %EDX to be relevant caller-saved registers, whereas %ESI and %EDI are assumed to be irrelevant callee-saved registers.

### Locally Adjusting the Calling Conventions

Our second approach to reducing adverse effects of callee-saved registers is to completely banish them from emulation code and make all registers caller-saved instead. As discussed earlier in this section, we can handle caller-saved registers efficiently, so that this approach effectively reduces the overhead that was previously incurred by unnecessarily preserving irrelevant callee-saved registers.

We propose to modify the calling conventions that apply to our emulation routines, so that *for these functions*, all registers are considered caller-saved and none callee-saved. We are free in the choice of calling conventions for our emulation routines, because they are never directly called from within compiler-generated code that relies on the original conventions. Emulation routines are exclusively called from a rewritten guest OS, so we only need to adapt the injected virtualization code to pure caller-saved calling conventions. No modifications to the guest OS are necessary.

We must, however, not ignore the implications on the size of the virtualization code: Previously, the code that is needed to save and restore the originally callee-saved registers was present only once in the prologue and the epilogue of the emulation function. In the worst case, we now have to replicate this code at each call-site, thus increasing the total code size. An increased code size generally reduces the effectiveness of the (trace-)caches and consequently degrades the overall system performance.

The pre-virtualization approach limits this effect by providing only a small scratch space for the rewriter, thus tightly bounding the size of the virtualization code. But this raises a new problem: What shall we do if the provided scratch space is not sufficient to save all relevant registers? Enlarging the scratch space is not possible if we have no access to the source code of the guest OS. Moreover, a larger scratch space would negatively affect the performance of the guest even if it is executed natively due to increased cache pollution.

The only obvious solution to this problem is to call the emulation routines indirectly via stubs and to delegate the task of preserving the relevant registers there. With $N$ registers to consider, we either need to statically provide $2^N$ stubs per emulation routine to efficiently handle all combinations of relevant and irrelevant registers, or we need to generate the required stubs dynamically during rewriting. In the worst case, even the dynamic approach requires the complete set of stubs.

In addition to the implied cost of indirectly calling the emulation routine, this approach—when applied for many emulation routines—dramatically enlarges the code size of the VMM and therefore considerably increases the risk for TLB misses during emulation. For IA-32, we need to handle $N = 7$ general purpose registers (%ESP must always be restored by the callee), which results in 128 stubs per emulation routine. Assuming a realistic mean stub size of 16 bytes and 4 kByte pages, only 2 sets of stubs fit into one page, so that we require round about 13 pages only for the stubs of the 25 emulation routines present in the current VMM. Furthermore, most of the 128 stubs will reside in a different cache line than the emulation routine, whereas in the single stub case, stub and emulation can easily be partially collocated in one line. Consequently, the new approach often requires one additional cache line for each called emulation routine as compared with the original approach, which also increases the risk for memory stalls due to cache misses.

Concluding, we state that modified calling conventions can in fact be used to our benefit, but only if the according emulation routines have few call-sites and only if these provide enough rewriting space, so that we do not need the stubs. We apply these conventions only to instructions that explicitly require preserving all registers as part of their emulation; for IA-32, these instructions are `cpuid` and `iret`. Here the pure caller-saved conventions imply no additional cost, as all registers are preserved by the caller anyway, but prevent the callee from preserving these registers again.

### Reallocating Registers

Our third approach to making callee-saved registers relevant at sensitive instructions uses *register reallocation*.

Instead of copying relevant caller-saved registers to irrelevant callee-saved registers at runtime as proposed in the first approach, we now try to reallocate a relevant caller-saved register $r$ to a callee-saved register $s$ throughout the whole lifetime of $r$ (see Figure 3.2). If such an $s$ is available, we can replace all references to $r$ with $s$, beginning at the previous assignment to $r$ up to and including the last use of the assigned value.

The result of this technique is that the reallocated caller-saved register $r$ is no longer relevant at the sensitive instruction, so that the virtualization code needs to save less registers. On the other hand, the now relevant callee-saved register $s$ implies no additional

| original code | reallocated registers |
|---|---|
| `movl (%ecx),%edx` | `movl (%ecx),%edi` |
| `pushl %edx` | |
| `call emulation` | `call emulation` |
| `popl %edx` | |
| `movl (%eax,%edx),%edx` | `movl (%eax,%edi),%edx` |

Figure 3.2: Effects of register reallocation. By reallocating live values from caller-saved to callee-saved registers, we can reduce the number of preserved registers. In this example we assume that %EDX is caller-saved and not used in the emulation. Furthermore, we assume that %EDI is an irrelevant callee-saved register.

cost, because the callee preserves its content anyway.

Register reallocation is superior to the first approach with respect to efficiency, because it reduces not only the number of memory accesses, but also the number of executed instructions: Reallocation completely removes both the save and the restore instructions for now irrelevant caller-saved registers.

This advantage is paid for with a reduced applicability: Caller-saved registers often cannot be reallocated, because they are used as *implicit* or *fixed* operands: On IA-32, `mul %eax` implicitly assigns to %EDX, whereas the `in` and `out` instructions for port I/O require their operands in parts of %EAX and %EDX. In both cases, the registers cannot be replaced with different ones. Another problem arises from registers, whose lifetimes extend into multiple basic blocks with disjoint sets of available callee-saved registers. For these cases, register reallocation is hard if not impossible without introducing additional register-to-register copy instructions. The latter would reduce or even negate the intended reduction of runtime and must therefore be avoided. The first technique, however, can easily be applied even in these cases.

### 3.2.3 Statically Evaluating Dispatch Tables

Some sensitive instructions, especially those related to device I/O, require different emulation code depending on the current value of one of their operands. For example, the emulation of IA-32's `out` instruction first retrieves the addressed port from %EDX to determine the accessed device. Then it selects and calls the appropriate emulation routine of the according virtual device. We term the first called emulation routine a *dispatcher*, as it only dispatches emulation requests to more specific handlers.

Occasionally, the value of the deciding operand is a previously assigned literal or can otherwise be determined statically. In these cases, we can replace the call of the dispatcher with a direct call of the more specific emulation routine. This allows us to remove one unnecessary control-flow indirection (a `call`/`return` pair) at runtime, which is depicted

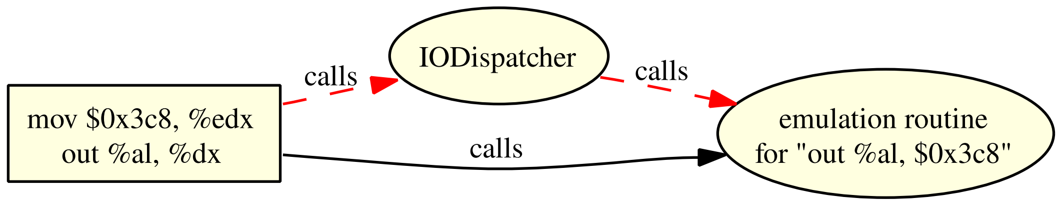

Figure 3.3: Static dispatch. Some sensitive instructions require different emulation code based on operand values. If their values are statically known, we can rewrite the sensitive instruction to directly call the specific emulation routine and bypass the otherwise needed dispatcher.

in Figure 3.3.

A similar approach is also applicable to memory mapped I/O, although the device addresses are not fixed, but rather assigned dynamically at system boot-up with page-size granularity (*plug'n'play*). The device registers are always located at constant offsets in the assigned memory region, and these offsets can be inferred from static program analysis. Together with the accessed device, which is available from the annotation for the sensitive memory operation, we have all the information that is required to statically select the appropriate emulation routine.

## 3.3 Optimizations for Successive Patches

The previously discussed techniques are limited to locally affect code generation for a single sensitive instruction, only register reallocation also modifies the surrounding code. These approaches deny possible synergistic effects that can result from generating code for more than one sensitive instruction. In the following sections we shall provide approaches that improve code quality at a more global scope.

### 3.3.1 Deferring Register Restoration

If two sensitive instructions are separated only by instructions that do not access relevant caller-saved registers, it is likely that the relevant registers at the first instruction are also relevant at the second one[3]. In this case we need not restore the saved registers after the first sensitive instruction, but can defer their restoration until after the second one. Furthermore, we need not save the registers again before the second sensitive instruction,

[3]The only exception occurs when the second sensitive instruction itself accesses the previously relevant registers. In all other cases, we know that the contents of the relevant registers prior to the first sensitive instruction are used after its execution; otherwise the registers would not be considered relevant. As neither the code in between the two sensitive instructions nor the second sensitive instruction accesses these registers, we conclude that they are still relevant at (and unchanged after) the second sensitive instruction.

| original virtualization | deferred register restoration |
|---|---|
| `pushl %eax` | `pushl %eax` |
| `pushl %edx` | `pushl %edx` |
| `call emulation1` | `call emulation1` |
| `popl %edx` | `popl %edx` |
| `popl %eax` | |
| `...` | `...` |
| `pushl %eax` | |
| `pushl %edx` | `pushl %edx` |
| `call emulation2` | `call emulation2` |
| `popl %edx` | `popl %edx` |
| `popl %eax` | `popl %eax` |

Figure 3.4: Deferred register restoration. By considering the virtualization code of two consecutive sensitive instructions, we can remove unnecessary restore/save pairs in between. For this example, %EDX is assumed to be accessed between the two patches, whereas %EAX is not used until after the second patch.

because they are still saved. Thus we amortize the preservation overhead across two successive emulations; by applying the same technique iteratively, we can occasionally defer the restoration even further. The effects of a single application are shown in Figure 3.4; %EAX is effectively preserved across two sensitive instructions at the cost of preserving it across only one.

We must take special care if the stack pointer is used by one of or in between the two sensitive instructions: As we usually preserve registers by pushing them onto the stack, the stack pointer has a different value than expected by the guest OS code. If the stack pointer is used as part of an address expression, we are in some cases able to adjust a specified offset to skip the additional values on the stack. In cases where such an adjustment is not possible, we cannot use this technique.

Despite the complex preconditions, this technique is often applicable especially in conjunction with device I/O: A first sensitive instruction issues a command, such as *read disk block,* whereas a second sensitive instruction provides additional data, reads the result or checks the status of the device. Another common scenario involves devices that only expose an address register and a data register. For such devices, each access first selects the desired internal device register by writing its index to the address register. The desired data is afterwards read from or written to the data register using a second sensitive instruction. In both cases, no or only few instructions are present in between the two device accesses.

| original code | rescheduled code |
|---|---|
| `movl $0x3c4,%edx` | `pushf` |
| `movl $0x80,%al` | `movl $0x3c4,%edx` |
| `pushf` | `movl $0x80,%al` |
| `...` | `...` |
| `out %al,%dx` | `out %al,%dx` |

Figure 3.5: Rescheduling. By reordering the instructions, we can reduce the number of live registers at sensitive instructions. In the original code, both %EAX and %EDX are live across the execution of the sensitive instruction `pushf`, whereas in the rescheduled code, both are irrelevant.

### 3.3.2 Rescheduling the Instruction Stream

During scheduling of the instructions, the compiler that is used to build the guest OS is not aware of sensitive instructions; especially, it does not perceive that the respective emulation code requires additional registers. Consequently, the compiler does not try to reduce the number of caller-saved registers that are live *across* such a sensitive instruction.

As we do know about sensitive instructions and the possible need for further available registers, we can reallocate registers in order to free up caller-saved registers as discussed in Section 3.2.2, and we can reorder the instructions, so that less caller-saved registers are live across sensitive instructions. We give an example for the latter approach in Figure 3.5: In the original code, %EDX and %AL are live during execution of the sensitive `pushf` instruction. The rescheduled code frees up both registers for use by the appropriate emulation code.

A second goal of rescheduling is to cluster sensitive instructions, so that the previously discussed deferred register restoration technique (Section 3.3.1) can be applied to reduce the virtualization overhead. For this purpose we need to either move two sensitive instructions as close to each other as possible or, equivalently, move non-sensitive instructions that are embraced by two sensitive instructions before the first or after the second sensitive instruction.

### 3.3.3 Compacting Basic Blocks

Most of the previously discussed optimizations, both for single and successive patches, reduce the number of instructions that make up the virtualization code. As the rewriter cannot change the size of the scratch space that is provided for the current sensitive instruction, the space that was previously occupied by unnecessary instructions must now be filled with innocuous instructions. Executing the latter costs additional cycles at runtime and thus reduces the effects of the presented optimizations.

We therefore propose to compact the basic blocks that contain sensitive instructions,

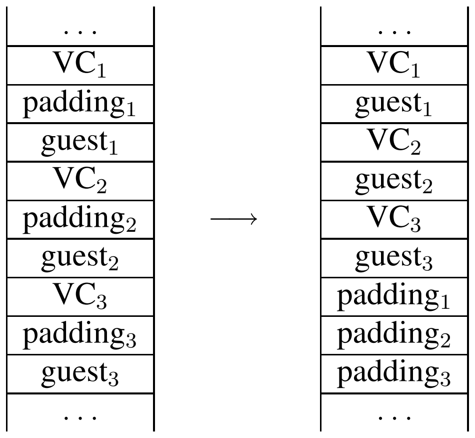

Figure 3.6: Basic block compaction. By moving unused scratch space to the end of the containing basic block we can more efficiently skip over it. $\mathrm{VC}_i$ denotes the virtualization code, $\mathrm{padding}_i$ represents the unused space in the same patch and $\mathrm{guest}_i$ symbolizes the interleaving code from the guest OS. The basic block ends after $\mathrm{guest}_3$.

after the virtualization code has been generated. In this process we propagate all unused space to the end of the basic block, where we can then implement efficient ways to skip it either by emitting a `jump` over the unused space, or by filling it with multi-byte `nops` as proposed in [8]. By collecting all unused scratch space, we only need to skip one large block instead of multiple small ones, which significantly increases the effectiveness of the multi-byte `nop` approach.

The end of the basic block is an ideal place for the unused scratch space: By keeping the space in its original block we avoid to move labels, which would require a global adjustment of all references. In the presence of indirect jumps, such an adjustment is not even possible. Furthermore, many basic blocks end in branching instructions. For unconditional branches, the presented approach completely removes the overhead of skipping over the unused space, as it has been moved to after the branch and out of the critical path. Similarly, taken conditional branches skip the unused code for free. Additional instructions to skip the unused space are therefore only required on the fall-through path of conditional branches and for basic blocks that do not end with a branching instruction. Figure 3.6 provides a visualization of basic block compaction.

## 3.4 Context-Aware Rewriting System

In the previous two sections, we proposed techniques that help to generate efficient virtualization code for sensitive instructions. Most of the discussed techniques require information such as live registers about the contexts of the sensitive instructions. In this section we shall present the design of a system that incorporates both an analysis component, which computes the necessary context-information, and a context-aware rewriter to implement the presented techniques.

### 3.4.1 Goals of Our Design

Our system is intended to enhance the pre-virtualization approach, so we do not want to trade in the primary goals and achievements of the latter for the improvements that are offered by our system. We therefore state the following goals for the context-aware rewriting system:

#### Reduced Runtime Overhead

The basic goal of our system is to effectively reduce the runtime overhead that is caused by replacing sensitive instructions in a guest operating system with *inefficient* calls to emulation routines.

#### Fully Automated Improvement

As stated in [14], pre-virtualization reduces the engineering effort of para-virtualization by “orders of magnitude” while maintaining nearly the same performance. Our system shall close the performance gap between pre- and para-virtualization without reintroducing significant engineering effort.

#### Hypervisor-Independent Enhancements

Pre-virtualized guest OS binaries adhere to the targeted platform API; they can be executed on bare hardware as well as in pure virtualization environments and even on all hypervisors with support for pre-virtualization. Our system shall not hinder this flexibility, but offer its improvements to all pre-virtualization–aware hypervisors and avoid to degrade the performance of the natively executed guest OS.

#### Efficient Rewriter

The rewriter must replace sensitive instructions with appropriate, efficient virtualization code. As this task has to be carried out whenever a guest OS is loaded into a virtual machine, we want to keep the rewriter fast.

#### General Applicability

We want to be independent of access to the source code of the guest OS to the largest possible extent. We can then efficiently virtualize even commercial guest operating systems

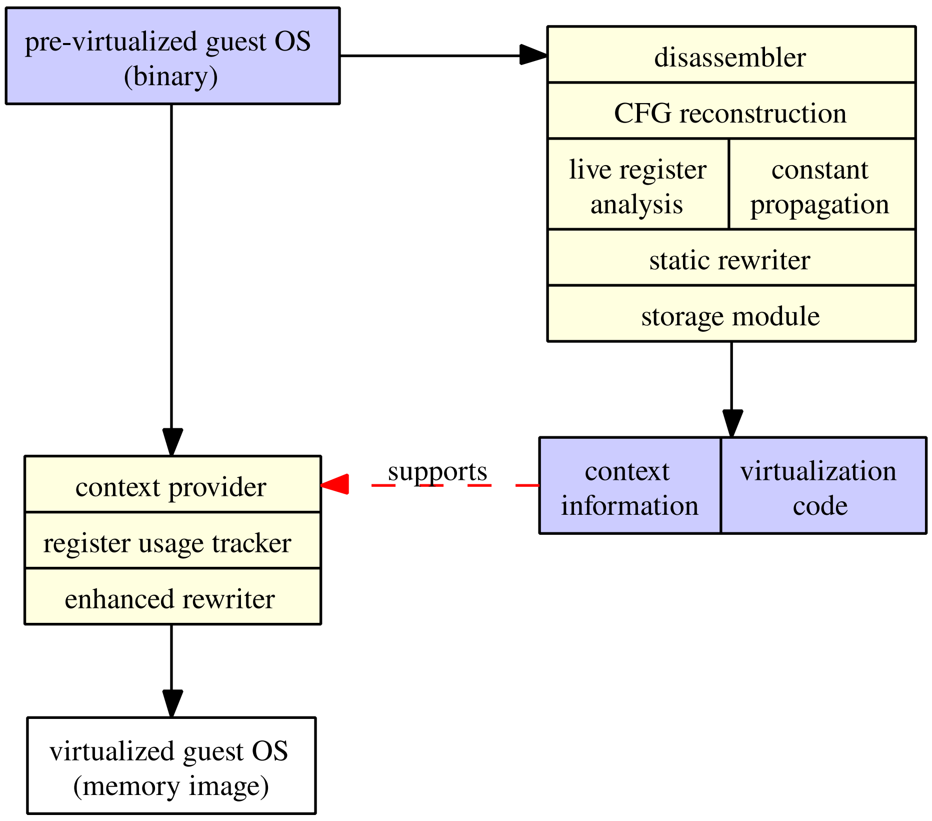

Figure 3.7: Structure of the proposed system. In an analysis phase (top-right box), we augment the pre-virtualized guest OS binary with context-information on all sensitive instructions. At load-time, the optimizing rewriter replaces the sensitive instructions in the memory image of the guest OS with dynamically generated, efficient virtualization code based on the results of the analysis.

that are delivered as pre-virtualized binaries but without source code.

### 3.4.2 System Structure

In order to keep the rewriter fast and simple, we propose to split the rewriting system into two components (Figure 3.7): An *analysis component* extracts the desired context-information from the guest OS binary and appends the analysis results. This is useful, because the information is solely dependent on the binary and does not change over time. Computing it once suffices and saves time during rewriting, as the information will instantly be accessible.

A *rewriter component* implements the previously discussed techniques to replace sensitive instructions with efficient virtualization code while a guest OS is loaded into a virtual machine. The rewriter dynamically generates the virtualization code based on the type of the rewritten instruction and on the context-information that is provided by the analysis component.

### 3.4.3 Sources of Context-Information

We have evaluated several sources for the required context-information, but found most of them to be insufficient or impractical.

**Compiler-Internal Data** While building the guest OS binary, the compiler already computes some of the information we need for our enhanced rewriting techniques: Both live registers and propagated constants, which we require to determine the relevant registers and to support the static dispatcher respectively, are also used during code generation. Unfortunately, the information is only available for compiler-generated code. As the machine-specific sensitive instructions have no counterpart in modern high-level languages such as C or C++, they are inserted using inline assembly or even plain assembler source files. In both cases, the compiler does not analyze the code and therefore does not directly deliver the desired information.

**Assembly Code** Our second option is to analyze the generated and hand-written assembly code, where all instructions are clearly identified as such and are easily parseable. Furthermore, symbolic labels and references explicitly convey the control-flow relations between instructions; we can even identify jump tables in the code. Still we opted against analyzing the assembly code for two reasons: First, access to the assembly code is equivalent to access to the guest OS source code, which we want to avoid in order to also support closed-source guest OSs. Second, the individual assembly source files only allow file-local analysis. In order to acquire precise information about live registers at return instructions, we need to perform whole-program analysis, which considers register usage information from all call-sites. As an alternative to whole-program analysis, we could augment the compiler to emit information about the registers that are used to return values to the callers for each function, but that would still require access to the source code.

**Object Files** The assembled object files no longer provide easily parseable instructions or explicit, symbolic control-flow relations within the files. As they require source code access similar to the assembly code but only provide less information, we also opted against using the object files as our source of context-information.

**Binary** The pre-virtualized guest OS binary is the only source of information, that is always available. To our benefit it contains all of the guest OS code, except for loadable kernel modules, so that we can use whole-program analysis to obtain precise information.

On the downside, the binary contains the instructions in their machine encoding; we first need to disassemble them to extract useful information. Furthermore, the labels that mark control-flow targets are no longer accessible but must be inferred from other sources, such as disassembled instructions or debugging symbols. Lastly, instructions and data cannot be distinguished in the binary, which complicates the analysis.

**Decision** Despite the indicated problems, we opted to obtain the desired context-information solely from analysis of the guest OS binary, as this is the only source of information that is always available and does not require access to the source code of the guest OS.

### 3.4.4 Analysis Component

The analysis component provides context-information on all sensitive instructions to prevent the rewriter from having to recompute this information over and over whenever the guest OS is loaded. We have identified several types of information that can be used during rewriting and can be precomputed in the analysis component: The rewriter requires information on live registers to identify and discard irrelevant registers and uses literally known register contents (alias propagated constants) in static dispatching. Furthermore, we inform the rewriter about the registers, whose restoration can be deferred until after the next patch because they are not used in between. In order to support an efficient implementation of basic block compaction, we deliver the addresses and sizes of basic blocks to the rewriter. We also provide the locations of addresses, that are evaluated at runtime relative to the current instruction pointer: As block compaction includes code motion, we need to adjust these addresses whenever the associated instruction has been moved.

We use standard data-flow analysis to compute live registers and statically known register contents (via constant propagation). The remaining information can easily be obtained once the necessary data structures are present: Data-flow analysis require a control-flow graph (CFG) representation of the analyzed code, along with additional data-flow annotations. We construct such a CFG from the results of a disassembler. During disassembly we extract the control- and data-flow relations for the instructions and store this information in abstract instructions. We then combine the latter according to their control-flow relations to make up the CFG, on which we perform all further analysis.

We also move the more complex rewriting techniques, namely register reallocation and rescheduling, into the analysis component. The results of this static rewriter are appended to the binary together with updated context-information for the rewritten blocks. To use the virtualization-friendly code, the load-time rewriter must patch both the guest OS and the context-information; alternatively it may completely ignore the information from the static rewriter. This approach is compatible with all hypervisors and does not influence native execution of the binary, so that we do not lose generality by implementing static rewriting early. We only reduce the complexity of the rewriter and speed up the rewriting process.

### 3.4.5 Rewriter Component

We split the load-time rewriter into three modules: The first one, the *context provider,* encapsulates the acquisition and retrieval of context-information. Although we propose to precompute this information and store it in the binary, this module can as well be implemented to compute the information on demand. The *register usage tracker* provides an abstract interface to the code generation routines that are affected by our optimizations. Most importantly, this module provides abstract register save/restore operations that use the context-information to implement the desired operation efficiently: irrelevant registers are silently discarded, whereas relevant registers are preserved in callee-saved registers if possible or on the stack. The third module provides the basic code generation and rewriting mechanisms. This module also integrates the rewriter component into the surrounding virtualization environment.

# Chapter 4

# Implementation

To demonstrate our approach, we implemented a context-aware rewriter and an analysis tool for IA-32 systems. The implementation of the rewriter is based on the Afterburner project [13], which is developed at the University of Karlsruhe. We reused the provided infrastructure and rewriting mechanisms to implement our enhancements. For the analysis tool, no such foundation was available, so that we implemented it from scratch.

In the following sections, we will first present how we implemented the proposed optimizations in the load-time rewriter. Afterwards, we will sketch the implementation of our analysis tool.

## 4.1 Rewriter Component

The Afterburner project provides a complete pre-virtualization framework: It includes assembler macros to automatically annotate and pad the sensitive instructions while the guest OS is built, the required emulation routines, and a rudimentary load-time rewriter. We reused the complete system, including the rewriting mechanisms, and integrated our optimizations.

We implemented the relevant techniques for efficient handling of caller-saved registers in abstract `save_registers` and `restore_registers` routines. Together with data-structures for the necessary bookkeeping, these routines make up a separate module, the register usage tracker. We integrated static dispatching into the code generation for both port I/O and memory mapped I/O. Furthermore, we also implemented deferred register restoration and compaction of basic blocks. As the latter optimizations require information about the structure of the generated code, we apply both of them in an additional pass over the patches after all code has been generated.

We begin the detailed discussion of the rewriter with a description of our approach to generating virtualization code and present the implementation of the optimizations afterwards.

### 4.1.1 General Approach

When a guest OS is loaded into the virtual machine, we use the pre-virtualization annotations to iterate over all sensitive instructions and replace them with appropriate emulation or virtualization code. The annotations provide neither the type nor the operands of the instructions, so we first disassemble the instructions to obtain this information. For inline emulated instructions such as `cli`, we then emit the hand-crafted emulation code (usually a single instruction) and continue with the next sensitive instruction.

For externally emulated instructions, more work needs to be done: If the current instruction references an operand in memory, we must first compute its effective address, so that we can pass it on to the emulation routine. We store this address temporarily in %EAX, but only after we have preserved its original value on the stack. We encapsulated the code generation for calls to emulation routines in a dedicated function, which generates the required code to preserve caller-saved registers, to pass arguments to the emulation routine, and to call it.

This function takes a descriptor for the requested emulation routine and an array of `(type, value)` pairs for each argument that is to be passed to the emulation routine. `type` can be either `register` or `immediate`, and `value` represents the according register number or literal value of the operand. Based on this information, we generate virtualization code as shown in Figure 4.1: First, we save the caller-saved registers %EAX, %ECX and %EDX, unless the descriptor for the emulation routine indicates that they are expected as *implicit* arguments. In the latter case, the registers are not saved, but pushed onto the stack as additional arguments to the emulation routine. We distinguish between *save* and *push* operations, because the former may be discarded for irrelevant registers or implemented as a register-to-register copy, see Section 4.1.2. A push operation always pushes its argument onto the stack.

Having handled the caller-saved registers, we then push the explicit arguments in reverse order onto the stack according to the usual C calling conventions and call the emulation routine. The remaining code cleans up the stack and restores all previously saved or pushed registers.

We will now show, how we implemented our optimizations for efficiently handling caller-saved registers in the save and restore operations.

### 4.1.2 Efficient Handling of Caller-Saved Registers

While generating virtualization code, we need to emit code to save and restore registers. The original rewriter directly generated appropriate `push` and `pop` instructions. We instead delegate the code generation to more abstract save and restore routines. These use the available context-information to transparently discard irrelevant registers by simply not emitting any code, to preserve relevant ones in unused callee-saved registers as proposed in Sections 3.2f, or to `push` them onto the stack. For convenience, the

| default virtualization code |
|---|
| save caller-saved registers |
| push accessed registers |
| `push arg`$_N$ |
| `...` |
| `push arg`$_1$ |
| `call emulation` |
| `addl $4*`$N$`,%esp` |
| pop accessed registers |
| restore caller-saved registers |

Figure 4.1: Generated virtualization code. The default virtualization code implements the standard C calling conventions between rewritten guest OS and emulation code.

| | | |
|---|---|---|
| %ECX | stack | ⟵ top of locations |
| %EDX | stack | |
| %EAX | %ESI | |

| | |
|---|---|
| | |
| 1 | ⟵ top of records |
| 2 | |

Figure 4.2: Internals of the register usage tracker. On one stack (left), the tracker records the storage location of all previously saved registers. The number of records that were created during each operation is stored on a second stack (right), so that the restore routine can undo exactly the effects of the preceding save operation.

`save_registers` routine accepts a bitmask to indicate the registers that should be preserved.

To be able to correctly restore the previously saved registers, we need to record, which register has been saved in which location—on the stack or in a callee-saved register. Additionally, we must keep track of which callee-saved registers have already been used to preserve caller-saved registers. For these reasons, we implemented the abstract save/restore routines in a dedicated module, the register usage tracker. Besides the two functions, this module also provides a stack, onto which we push a (register, location) pair whenever a register is actually saved; discarded registers are not recorded. Instead, we store the number of pairs that have been created by each call to the save routine on a second stack. In the restore routine, we first take this value off the stack and then restore the according number of registers from their respective locations.

In Figure 4.2 we depict a situation, where a first call of the save routine caused two registers to be saved: At first, we emitted code to preserve %EAX in %ESI, followed by code to push %EDX onto the stack. The 2 on the right stack indicates that we actually saved two registers. A second call of the save routine additionally pushed %ECX onto the

| (a) virtualization code | (b) complete stub | (c) stub discarding %ECX |
|---|---|---|
| `push arg`$_N$ | `stub:` | `stub_5:` |
| `...` | ` pushl %eax` | ` pushl %eax` |
| `push arg`$_1$ | ` pushl %ecx` | ` pushl %edx` |
| `call stub` | ` pushl %edx` | ` lea 8(%esp),%eax` |
| `addl $4*`$N$`,%esp` | ` lea 12(%esp),%eax` | ` call emulation` |
| | ` call emulation` | ` popl %edx` |
| | ` popl %edx` | ` popl %eax` |
| | ` popl %ecx` | ` ret` |
| | ` popl %eax` | |
| | ` ret` | |

Figure 4.3: Indirectly calling emulation routines. (a) The virtualization code only pushes the operands of the sensitive instruction onto the stack and calls a stub. (b) The stub preserves the caller-saved registers and passes a pointer to the "real" arguments to the emulation routine. (c) To discard irrelevant registers, a set of stubs must be provided, each of them saving a different subset of the caller-saved registers. The example discards %ECX.

stack, creating another pair on the left stack and pushing the 1 onto the right stack. If this second call again indicated that %EAX should be saved, we would detect that it is already preserved and not save it again.

The next call to the restore routine will first pop the 1 off the right stack and then emit `pop %ecx` to restore the first register according to the top pair on the left stack. The second call of this routine will then take the 2 off the right stack and emit `pop %edx` followed by `movl %esi,%eax` to restore the remaining registers.

Unless we use indirect calling conventions (see Figure 4.3), these two routines completely implement the techniques we proposed to efficiently handle caller-saved registers.

### Variant for Indirect Calling Conventions

We also implemented a variant of the original calling conventions, which inlines only the necessary code into the guest OS and delegates the preservation of the caller-saved registers to *stubs*. We show the generated virtualization code and an appropriate stub in Figure 4.3(a) and (b). As the stubs are not generated dynamically but are provided statically as assembly functions, the above approach to discarding irrelevant registers fails: No registers are preserved during code generation.

To still benefit from the available context-information, we supply 8 stubs per emulation routine, each of which preserves a different subset of the 3 caller-saved registers[1].

[1] We do not support using callee-saved registers as storage locations with indirect calls; this would require numerous additional stubs to also cover all subsets of available callee-saved registers.

The code generation function selects and emits code to call one of these, depending on which registers need to be saved. For this purpose, we represent %EAX with 1, %ECX with 2, and %EDX with 4 and use the sum over the relevant registers to obtain a key. With this key we then index an array in the descriptor of the emulation routine to obtain the entry point of the stub. Each of the provided stubs preserves exactly the registers that are indicated by its respective index. Figure 4.3(c) shows stub number $5 = 1 + 4$, which preserves registers %EAX and %EDX, but discards %ECX.

Although our default calling conventions are more efficient due to reduced indirections in control-flow (one call/return pair less) and in argument access, we still have to use the indirect approach under certain circumstances: If the provided scratch space does not suffice for the efficient conventions, we must fall back to the compact ones.

We use automatically generated glue code to reconcile the two different stack layouts—the stubs push the caller-saved registers after the arguments have been pushed, whereas our default code preserves them before the arguments are pushed. This allows us to call the same emulation routines using either convention.

### 4.1.3 Static Dispatch

Static dispatch circumvents calls to dispatcher routines by statically evaluating their dispatch tables based on literally known register contents.

Implementing static dispatch for port I/O was straight forward, as all required routines were already present. The relevant `in` and `out` instructions are both available in two types: the first one encodes the accessed port as an 8 bit immediate operand, the second one expects the port number in %DX. The first type was already handled with a static dispatcher, which emits code to circumvent the dynamic dispatcher for the only virtualized device with port I/O, which is the programmable interrupt controller (XT-PIC). We only added lookup of the statically known content of %DX for the second form, and passed the determined value on to the very same static dispatcher. If we have no information about the content of %DX, or if the static dispatcher does not handle the specified port, we fall back to the original implementation and emit code to call the dynamic dispatcher.

For memory mapped I/O, we augmented the rewriting routines to take a device-specific static dispatcher as an additional argument. For each sensitive memory operation, we invoke this function with the type of the operation, its operands, and the displacement of the memory operand, from which the static dispatcher can infer the accessed device register. If the combination of operands is not handled by the static dispatcher, we proceed as in the original implementation. At present, no static dispatcher for memory mapped I/O devices is available, so that the validity of this approach still has to be evaluated.

```
typedef struct
{
  uint32_t vaddr;       /**< vaddr of the patch */
  uint32_t prev;        /**< vaddr of the previous patch */
  uint32_t size:6;      /**< length of the patch */
  uint32_t nops:6;      /**< number of NOPs at the end */
  uint32_t eax_push:5;  /**< offset to pushl %eax */
  uint32_t ecx_push:5;  /**< offset to pushl %ecx */
  uint32_t edx_push:5;  /**< offset to pushl %edx */
  uint32_t eax_pop:5;   /**< offset to popl %eax */
  uint32_t ecx_pop:5;   /**< offset to popl %ecx */
  uint32_t edx_pop:5;   /**< offset to popl %edx */
  uint32_t pcrel:5;     /**< offset to call target */
  uint32_t common:3;    /**< bitmask of possibly deferred
                             registers from previous patch */
  uint32_t unused:14;
} __attribute__((packed)) df_deferred_t;
```

Figure 4.4: Structure used to support deferred register restoration.

### 4.1.4 Deferred Register Restoration

By deferring the restoration of a preserved register until after the next patch, we can remove unnecessary restore/save pairs for the register in between.

We implemented this optimization in two steps: During rewriting, we use a dedicated structure for each patch (Figure 4.4) to record the locations of `push` and `pop` instructions that preserve registers, but do not pass arguments to the emulation routine. From this structure, the fields `vaddr`, `prev` and `common` are precomputed by the analysis tool to convey the immediately preceding patch and the set of deferrable registers: Registers that are used between two sensitive instructions cannot be deferred to after the second one.

After all patches have been processed, we iterate over these structures and try to defer each `common` register as follows: From the previous patch, we determine the register that is restored last (largest offset of all `xxx_pop` fields). If this register matches the one that is first saved in the current patch, we disable both the `push` and the `pop` instruction by replacing them with `nops` and update the patch description. Our calling conventions guarantee that the passing of arguments to the emulation routine, which also uses the stack, and the presented optimization do not interfere with each other: The `push` and `pop` instructions for preserved registers embrace the stack operations for the arguments, and the arguments do not make assumptions about the number or location of preserved registers on the stack.

After having disabled the superfluous instructions, we compact the patches by moving the instructions that follow the disabled ones to fill the gaps. This way, we collect all unused space at the end of the patch and simplify the upcoming basic block compaction. Moving instructions is uncritical in most cases, only the `call` of the emulation routine encodes its target relative to its own address. During code generation, we therefore also record the location of this address in the `pcrel` field of the per patch structure and adjust the target address whenever we move it around.

Let us now introduce a shorthand notation for the patches, in which we represent operations on registers with single letters: `push` instructions in uppercase and the matching `pop` instructions in lowercase. We shall further use a colon ":" to separate individual patches. In this notation the above approach successfully transforms $Aa : Aa$ into $A : a$. However, the approach fails to optimize $ABba : Bb$ to $BAa : b$, because the last restored register $a$ does initially not match the first saved register $B$ of the following patch. Our refined implementation also applies in these cases. We first change the order in which the registers are saved in both patches to establish the required match, so that $ABba : Bb$ becomes $BAab : Bb$, which is then improved to the desired $BAa : b$. If two registers are both saved and restored in one patch, their order on the stack is irrelevant except for this optimization.

Our approach is greedy in so far as it always applies all local optimizations that are possible per patch. We therefore might end up in a local optimum, which need not coincide with the global optimum[2]. Finding such a global optimum and exporting appropriate guides to the rewriter is certainly possible in the analysis tool, but is beyond the scope of our implementation. Similarly, our implementation is limited to deferring registers that are preserved on the stack. We deem this to be sufficient to demonstrate our approach; expanding it to also defer the restoration of registers that are preserved in callee-saved registers has been left for future work.

### 4.1.5 Compaction of Basic Blocks

With basic block compaction, we collect all unused scratch space at the end of the basic blocks to more efficiently skip over it.

This optimization includes moving code, similar to the previous one. We therefore reuse the per-patch structure to locate all patches and determine the size of the unused scratch space (field `nops`). We use additional information from the analysis tool about the boundaries of basic blocks, so that we can move the complete code that follows the current patch in the same basic block to fill the unused scratch space. Furthermore, we use relocation information from our analysis tool to find and update relative addresses in the moved code.

[2] Consider $Aa : ABCcba : BCcb$, which is transformed to $A : BCcba : BCcb$. The inner patch cannot be reordered. A better solution for this example would be $Aa : BCAa : cb$ (by first combining the 2nd and 3rd patches), or possibly even $A : BC : bca$.

We fill the combined unused space with multi-byte `nops` as proposed in [8] and add a `jump` instruction at the beginning to skip the code if appropriate. We tested several approaches to deal with unused space and implemented the most efficient variant depending on the number of unused bytes; according benchmark results shall be given in Section 5.2.2.

## 4.2 Glue Code Generators

As we simultaneously support different calling conventions between the rewritten guest OS and the emulation, we need glue code to adapt the emulation code: Our default conventions always pass the accessed registers as the last arguments, whereas the emulation code might specify them earlier; the glue code must reorder the arguments. For the original, indirect conventions, the stubs only pass a pointer to the argument frame to the emulation routine; here, the glue code resolves this indirection and provides the emulation code with its real arguments.

To simplify the transition to different calling conventions for the emulation routines, and to enforce consistent conventions for all of them, we implemented a glue code generator based on the GNU tools flex and Bison. To specify the glue code, we use annotated header files in a restricted C/C++ dialect. Our generator transforms this header file into a set of assembly functions (the stubs) and C routines, which we dub *trampolines*. Additionally, the generator emits a descriptor for each emulation routine, which conveys enough information to the rewriter so that it can generate proper code to call them via the generated glue code.

Related to our glue code is the introduction of pure caller-saved calling conventions into the compiler, which we proposed in Section 3.2.2. We shall therefore conclude this section with a presentation of our modifications of the GCC, which selectively enable the pure caller-saved conventions.

### 4.2.1 Specification Language for Glue Code

To prevent mismatches between the specified emulation routines and their implementation, we designed the specification language for glue code as a transparent extension to C/C++. During compilation of the emulation code, the compiler verifies that the signatures of the specified emulation routines and their implementations are compatible. From the verified specification header file we generate the stubs and trampolines. We added two keywords to support glue code generation:

**`__emulation__`** is used to indicate that the preceding function prototype describes an emulation routine. This avoids the generation of unnecessary stubs and trampolines, which would only increase the size of the VMM.

**__location__** is an attribute to arguments and indicates the location, in which the argument is to be found. On IA-32, we currently support registers (eax, ecx, edx, ebx, esp, ebp, esi and edi), the stack and the return address in the guest OS, GUEST_RA. The latter is needed for the emulation of updates to %CR0 (paging enable). If no location is specified, stack is assumed per default.

We can therefore use

```
int emulate( int &arg0 __location__( eax ),
             int arg1,
             int guest_ra __location__( GUEST_RA ) )
             __emulation__;
```

to declare an emulation routine that receives its first parameter from %EAX, the second one from the stack and the third one from the stack location that holds the return address in the guest OS. We furthermore distinguish IN and INOUT arguments: The reference operator "&" in conjunction with a register location indicates that the respective register is to be read and modified during emulation. In the previous example, the value of %EAX as exposed to the guest OS can be modified by simply assigning to arg0.

To enable correct processing of the specification file by a regular compiler, we disable our annotations via pre-processor macros.

### 4.2.2 Generated Glue Code

We created a glue code specification for all emulation routines in the Afterburner. From this specification, we automatically generate 9 stubs and a set of trampolines (one per calling convention) for each emulation routine. Additionally, we create a descriptor to convey all the information required for calling these routines to the rewriter. We provide sample code in Appendix A and show the structure of the descriptor in Figure 4.5.

#### Stubs

Stubs are used by some of the supported calling conventions in order to remove the register preservation code from the rewritten guest OS, thus reducing the necessary size of the scratch space.

For the original approach, we emit the according stub for preserving all caller-saved registers as burn_NAME_OLD. For our improved variant, we emit up to 8 stubs, one for each combination of the 3 caller-saved registers being saved or discarded, and name them stub_NAME__SRMn. The n represents the mask of registers that are preserved by the stub.

In the latter case, we reduce the code size by merging identical stubs. These result from emulation routines that take some of their arguments from caller-saved register locations: Instead of both preserving the register and passing it on the stack as an argument, we only pass it as an argument and restore it from this location. Consequently, the stubs that discard or preserve the argument registers are identical—and are merged into only one copy of the code.

### Trampolines

The trampolines are small C functions that are named after the emulation routine, with `_ext_X` appended to make the name unique. `X` indicates the implemented calling convention:

- `stub` denotes the original approach; the trampolines take a pointer to locate the arguments on the stack
- `TRAMPOLINE_T` resembles the original approach but allows to discard irrelevant registers in the stubs; the trampolines again take a pointer to find the arguments
- `STACK` indicates the trampoline of our default conventions, which follow the standard C calling conventions
- `REGPARM` accepts the first $\leq 3$ arguments in registers rather than on the stack

The trampolines are simple wrappers for the emulation routines; their only purpose is to retrieve the arguments for the proper emulation code and thus abstract from the current calling conventions. For efficiency, the trampolines inline the emulation code and thus avoid additional function call overhead.

### Descriptor

Besides the entry points of the stubs and directly called trampolines, the descriptor also conveys the number of arguments that are passed via the stack, the registers that are implicitly used as arguments to the function, and additional flags that indicate, whether a possible return value is returned on the stack or in `%EAX`, as usual.

## 4.2.3 Locally Modified Calling-Conventions

We implemented locally modified calling-conventions (see Section 3.2.2) as an additional function attribute `callersaves` in GCC 3.3.6. We check for the presence of this attribute when the compiler generates prologue and epilogue code for a function and suppress both saving and restoring of callee-saved registers if the attribute is attached to the function. Our modifications comprise only round about 20 lines of code for the complete functionality of the `callersaves` attribute.

```
typedef struct {
  uint32_t n_argsonstack;  /* # of arguments */
  uint32_t accessed_regs;  /* bitmask of implicit args */
  uint32_t inout_regs;     /* bitmask of INOUT args */
  uint32_t pure_in_regs;   /* bitmask of IN args */
  void *trampoline;        /* addresses of the routines */
  void *regparm;           /*   that might get called by */
  void *stackparm;         /*   generated virtualization */
  void *original;          /*   code */
  void *entry_point[8];    /*   cont'd */
  /* flags that influence code generation: */
  struct {
    uint32_t retval_on_stack:1;
    uint32_t retval_in_reg:1;
    uint32_t trampoline_needs_argument:1;
    uint32_t accesses_guest_ra:1;
  } opts;
} burn_stub_desc_t;
```

Figure 4.5: Descriptor for generated glue code.

## 4.3 Analysis Component

We also implemented an analysis tool, which extracts context-information on sensitive instructions from the guest OS binary and exports the results to the rewriter. In the following sections, we shall first recapitulate the structure of the analysis component and then provide details on the comprised modules in the order of their execution during an analysis.

### 4.3.1 Overview

In order to extract context-information on sensitive instructions from the guest OS binary, we first obtain information on each individual instruction using a disassembler. Disassembling IA-32 code is only reliable for sequential blocks of code between a label and the first unconditional jump or return. The following bytes might not contain valid instructions but confuse the disassembler (see Figure 4.7). We implemented a code discovery module, which locates the beginnings or *entry points* of all such code blocks. We then disassemble each located code block, store the data-flow relations in terms of read and written operands in an abstract instruction, and connect the latter according to the control-flow relations of the disassembled instructions to form a control-flow graph (CFG).

Once the CFG for the whole binary is available, we initiate an intra-procedural data-

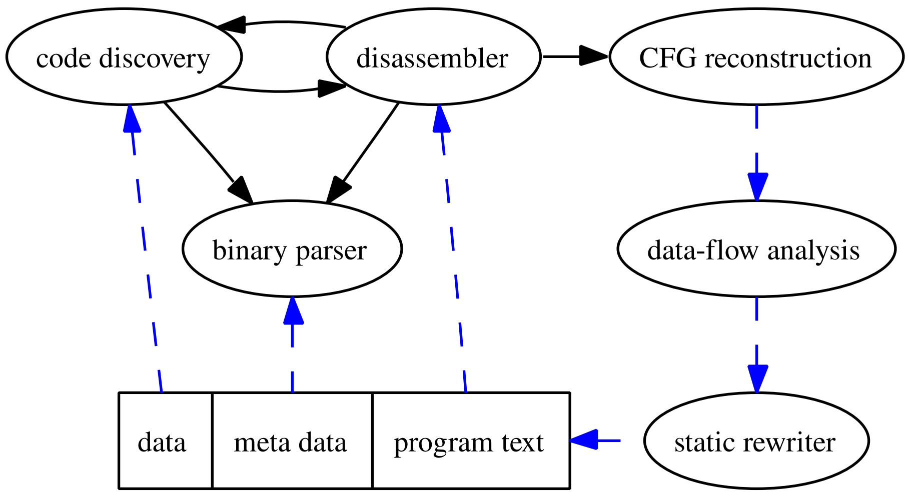

Figure 4.6: Structure of our analysis component. Solid lines represent control-flow (calls), dashed lines represent prevailing data-flow.

flow analysis on each connected subgraph, which correspond roughly to the original subroutines: Compiler optimizations like *tail-call elimination* can connect the subgraphs of several subroutines, but usually each subroutine is represented as a single connected subgraph of the CFG. To improve the precision of the live register analysis, we support it with results from an inter-procedural analysis, thus refining the approximations at control-flow sinks. Further analysis extract deferrable registers from the CFG. Basic block boundaries and the locations of relative addresses, both of which are required to move unused scratch space to the end of each basic block, are directly accessible from the CFG and require no analysis, but only another pass over the graph.

A complete implementation of the proposed static rewriting techniques within the analysis tool is beyond the scope of this work. We therefore limited our implementation to an analysis of possible applications of these techniques.

Figure 4.6 summarizes the control- and information-flow within our system. The binary parser only implements access to ELF files and abstracts from the details of parsing the headers, so that we will skip its implementation details. We shall discuss the remaining modules in the following sections.

(a) intended code:

```
0x0d:   eb 01      jmp    0x10
0x0f:   00         [padding]
0x10:   5b         pop    %ebx
0x11:   58         pop    %eax
0x12:   c3         ret
```

(b) result of linear disassembly:

```
0x0d:   eb 01      jmp    0x10
0x0f:   00 5b 58   add    %bl,0x58(%ebx)
0x12:   c3         ret
```

(c) result of disassembly in conjunction with code discovery:

Instruction stream 1 (ends in unconditional jump, yields new entry point at 0x10):

```
0x0d:   eb 01      jmp    0x10
```

Instruction stream 2:

```
0x10:   5b         pop    %ebx
0x11:   58         pop    %eax
0x12:   c3         ret
```

Figure 4.7: Incorrectly disassembled instructions. In conjunction with variable length instruction encodings, interspersed data like padding can confuse the disassembler and finally lead to incorrectly reconstructed control- and data-flow graphs.

### 4.3.2 Code Discovery

Code discovery is necessary, because linear disassembly[3] of IA-32 binaries is unreliable: It fails in the presence of interspersed data in conjunction with IA-32's variable length instruction encodings as depicted in Figure 4.7.

Furthermore, the call-tree, which is rooted in the main entry point of the binary, does not cover all reachable functions: Most device driver routines are only called via function pointers, but the targets of such indirect calls cannot be determined statically. Therefore the call-graph is incomplete and we need code discovery to find these routines.

[3]We refer to the common disassembly method, which assumes non-overlapping instructions and starts the next instruction immediately after the last byte of the previously disassembled one, as *linear disassembling*. We use linear disassembling for identified instruction streams after code discovery.

We implemented code discovery in two phases: The first phase extracts an initial set of entry points (including the main entry point, which is indicated in an ELF header field) by parsing the symbol-tables and extracting the addresses of subroutines. As symbol-tables need not be present and are generally incomplete, we look for additional entry points in the data sections. For this purpose, we iterate byte-wise over the loadable sections and look for words within the virtual address range of the text section. Each such word is a potential entry point to the binary: It might be part of a jump table (resulting from a C-like `switch` statement), or it might be an initialized function pointer; it might even be no address at all—we cannot know.

To handle these potential entry points correctly, we weigh the consequences of accepting or discarding them: Additional, spurious control-flows in the CFG do not affect the correctness of a data-flow analysis, they can merely reduce its precision (see Section 4.3.5 for more details on data-flow analysis). Missing control-flows, however, lead to incorrect results, as the effects on the missing flows are not accounted for during analysis. Our primary goal is therefore to reconstruct a complete CFG, so we assume all of the above potential entry points to be in fact real entry points to the binary and accept them as such. The presented first phase of code discovery locates the destinations of all memory-indirect jumps and calls as well as all function pointers that are used in the program.

In the second phase of code discovery, we recursively add further entry points from disassembled instructions. We collect the targets of (direct) jumps and calls and use them later on to start disassembling the indicated block of code. Furthermore, we handle immediate operands in a similar way to words from data sections, because they might subsequently be used as the destination of register-indirect branches. To conclude, the second phase determines all code fragments that are reachable from previously identified code blocks. Together with the first phase, we end up with a nearly complete coverage of the guest OS.

Truly unreachable code, such as unused functions, is potentially not found using the presented approach. This is only a problem in the presence of loadable modules that introduce calls to previously unused functions, or with self-modifying or otherwise “queer” code: If the target addresses of indirect jumps are computed at runtime based on relative offsets, our approach fails to find this entry point. Self-modifying code is a general problem for the pre-virtualization approach: The modified code cannot be rewritten with virtualization code, because we only rewrite at load-time, where the new code is not yet present. Both previous cases have turned out to be no problem in practice and are therefore ignored by our implementation. Loadable modules can even be handled correctly, because they reference the functions in the guest OS kernel symbolically and have these references resolved at load-time. Consequently, the called functions have a matching symbol-table entry, thus they are included in the set of entry points in the first phase of code discovery.

## 4.3.3 Disassembly

For our disassembler, we used the IA-32 instruction set specification from the GNU binutils [23] and augmented it to also convey detailed data-flow information for the instructions: We need to know which operands are read and which are potentially modified. Furthermore, we need to determine the implicit arguments of all instructions. We added this information to the specification of each instruction and evaluate it during decoding of the instructions.

The disassembler transforms all instruction streams, whose beginnings are provided by the code discovery module, into an abstract and machine independent representation. The latter comprises all information that is relevant for the subsequent analysis in an easily accessible form:

- all potential control-flow predecessors and successors, represented by their respective addresses
- input operands, including those only conditionally or partly accessed
- output operands, including those only conditionally or partly modified

We store input and output operands in separate bitvectors, which is a representation well suited for an efficient implementation of the upcoming data-flow analysis. We differentiate three types of operands: registers, memory locations, and immediates. Each addressable[4] processor register is represented as a single bit in the input/output bitvectors. For the general purpose registers, which are accessible as 8 bit, 16 bit, or 32 bit entities, we use three bits to discern the disjoint parts of the registers: for %EAX, we provide separate bits for %AL, %AH and the upper 16 bits, %EAX without %AX. An access to %AX therefore affects two bits in our bitvectors, although only one operand is used.

The second type of operands, main memory, is merged into only one single abstract memory location and represented with one bit. This prevents errors due to aliasing (i.e., the same memory location is accessed by two lexically different address expressions) and simplifies correct handling of multi-processor systems: By assuming the single memory location to be *volatile,* we make the analysis multi-processor safe, because the analysis does no longer rely on the memory contents remaining unchanged between store and subsequent load operations. For the same reason, memory mapped device I/O can be handled correctly only if device memory is regarded as volatile: If the guest OS first writes and then reads from a memory location that maps to a device register, a more precise analysis might conclude that the previously assigned value is retrieved again—although the value that is read might in fact be different, depending on what the device returns. Future work should differentiate certain memory regions—for example a processor- and thread-local stack, globally shared memory or device memory—in order to more precisely

[4]For IA-32, for instance the instruction pointer (IP, or program counter, PC) cannot be addressed directly.

track values that are temporarily stored in non-volatile memory. This could increase the number of literally known registers at sensitive instructions and therefore support static dispatching (see Section 3.2.3).

The third type of operands, immediates, need not be indicated in the bitvectors, because they can only be used as input operands and do not represent an addressable storage location. Still we store them in the abstract instructions for use in *constant propagation,* which is the data-flow analysis that yields literally known register content.

### 4.3.4 CFG Reconstruction

During CFG reconstruction, we add explicit control-flow edges to the abstract instructions from the disassembler. Most instructions transfer control to the next instruction in the linearly disassembled stream (fall through), whereas unconditional jumps mark the end of such a stream and transfer control to a different location. Conditional branches even result in two possible control-flow successors: the next instruction in the stream if the condition is false or the explicit destination if the condition is true.

We need to pay special attention to register- or memory-indirect jumps, because we cannot in all cases add the proper target instruction to the set of control-flow successors:

**Memory-indirect** jumps specify a memory location that contains the absolute address of the destination of the jump. If the address of the memory location is statically known and if this address is contained in a read-only region, which is indicated by meta-data from the binary, we can determine the destination from the given memory location. Otherwise, we mark the jump instruction as "end of control-flow", which signifies that we do not know where the execution continues, and enforce safe approximations during analysis.

**Register-indirect** jumps can be handled similarly to memory-indirect jumps: If the register contents is statically known, we use it as the target address of the jump instruction. Otherwise, we mark the instruction as "end of control-flow" as before.

**Direct** jumps provide the relative or absolute address of their destination, which is therefore easily identifiable.

Besides conditional and unconditional jumps, we also need to discuss `call` and `return` instructions: We consider `call` instructions to transfer control to the next instruction in the stream rather than to the indicated subroutine, but additionally create a call-graph that reflects the caller/callee relations. `return` instructions unconditionally mark the end of a control-flow and are linked to their call-sites within the call-graph for convenient inter-procedural data-flow analysis.

### 4.3.5 Data-Flow Analysis

We use data-flow analysis to compute live registers and statically known register contents (also known as propagated constants). Let us therefore briefly introduce you to the general approach of data-flow analysis, before we present our implementation of constant propagation and live register analysis in more detail. We conclude this section with a description of our approach to computing deferrable registers.

#### General Approach

Data-flow analysis annotate each instruction of the analyzed code fragment with sets of properties, which will be (or may be) valid at this instruction during the execution of the program. To do so, the analysis begins with a set of properties that is assumed to be valid at the beginning of the code and propagates it along the control-flow through the whole fragment. At each instruction, the current set of properties is updated according to the effects of the instruction, before it is passed on to the next instruction.

At control-flow mergers, we have to combine the information from the joining flows. Two combining operations are common: *May analysis* accept the union set of the information from all flows; live register analysis is one example for this kind. *Must analysis* only propagate the subset of the information that is common to all joining flows; constant propagation uses this approach.

Additionally, the analysis can be performed either *forwards* as described above or *backwards*. Forward analysis deliver information about the history of the execution of the program to each instruction—for example, previously assigned literals—whereas backward analysis enable each instruction to peek into the future: A live register analysis supplies information about the set of registers that might be read *later on*.

Due to loops in the control-flow, the effects of an instruction can affect this very same instruction. For this reason, we need to update the property sets until we reach a fixpoint. We refer to [19] for a complete theoretical background and more details on data-flow analysis.

We implemented the generic data-flow algorithm, independent of direction and merging behaviour, as an abstract C++ class and derived subclasses for constant propagation and live register analysis.

#### Constant Propagation

Constant propagation tracks the literal value of registers during program execution. For this purpose we pass (register,value) pairs along the control-flow and update them whenever a register is assigned a new value: If the new value is literally known, because it is an immediate or the result of an arithmetic operation with known input operands, we update the pair with the new value. Otherwise, we remove the pair from the set of propagated

**(a) constant propagation:**

| | |
|---|---|
| {unknown data} | |
| `popa` | 1. restore all registers from memory |
| {} | 2. no values are known (volatile memory) |
| `movl $0x3c4,%edx` | 3. literal assignment |
| {(%EDX,0x3c4)} | |
| `movb $4,%al` | 4. literal assignment |
| {(%AL,4),(%EDX,0x3c4)} | |
| `jnz label1` | |
| {(%AL,4),(%EDX,0x3c4)} | (∗) |
| `shl $1,%al` | 5. literal assignment, because %AL is known |
| {(%AL,8),(%EDX,0x3c4)} | (∗∗) |
| `label1:` {(%EDX,0x3c4)} | 6. intersection of (∗) and (∗∗) |
| `out %al,%dx` | 7. emulate using %DX=0x3c4 |

**(b) live register analysis; starts at the bottom:**

| | |
|---|---|
| {%EAX,%EBX,%ECX,%ESP,Z} | |
| `jnz label2` | 7. reads zero flag (Z) |
| {%EAX,%EBX,%ECX,%ESP} | 6. union set of (∗) and (∗∗) |
| {%EAX,%ESP} | (∗∗) |
| `movl $42,%ecx` | 5b. discards %ECX |
| {%EAX,%ECX,%ESP} | copied from jump target |
| `jmp common` | |
| `label2:` {%EAX,%EBX,%ECX,%ESP} | (∗) |
| `addl %ebx,%eax` | 5a. reads %EAX and %EBX, %EAX stays live |
| {%EAX,%ECX,%ESP} | |
| `common:` {%EAX,%ECX,%ESP} | |
| `pushl %eax` | 4. reads %EAX |
| {%ECX,%ESP} | |
| `pushl %ecx` | 3. reads %ECX |
| {%ESP} | 2. all other registers are overwritten in 1. |
| `popa` | 1. writes all registers, reads %ESP |
| {unknown data} | |

Figure 4.8: Sample data-flow analysis. (a) constant propagation: known register contents is propagated forwards through the CFG; (b) live register analysis: live registers are propagated backwards through the CFG. Horizontal lines separate basic blocks and indicate control-flow mergers (at labels) or diversions.

pairs, indicating that we have no knowledge about the current content of the register. On control-flow mergers, we remove all pairs unless *all* joining flows deliver the same pair. In other words, we only keep the common subset of the information from all joining flows, which guarantees that we only propagate safe information. If in doubt, we prefer ignoring partial knowledge over making wrong assumptions. We provide a commented sample analysis in Figure 4.8(a).

Our implementation only handles assignments of literals to registers, register-to-register copies and clearing registers, for example via `xor %eax,%eax`. All other assignments to registers result in their respective (register,value) pairs being invalidated, even if the assigned value could be inferred statically. This is sufficient for the intended use in static dispatching of I/O routines: The index of the accessed port is usually assigned as a literal to a register (most likely %EDX) and subsequently used in a sensitive instruction without intervening modifications.

### Live Registers

A register is considered to be live at an instruction, if its value is potentially read afterwards before it is overwritten. We compute the set of live registers for each instruction by propagating the (bit-)set of live registers backwards through the CFG.

At control-flow sinks (i.e., return instructions and jumps with unknown or uncertain destinations), we first assume no register to be live. At each instruction, we then update the set of currently live registers: First, we remove all registers that are assigned a new value by the instruction to indicate that their previous value has not yet been read and is therefore considered not to be live. Afterwards, we add all registers that are read by the instruction to the set of live registers prior to the instruction. We use the input and output bitvectors of the abstract instructions to update the identically encoded live register sets.

At control-flow mergers[5], we propagate the union of the live register sets of all joining flows: A register is considered live if it *might* be read later on; a register might be read if it is read in at least one of the possible succeeding control-flows. Figure 4.8(b) provides a commented example of an analysis of live registers.

In the first step, during which we assume no register to be live at control-flow sinks, we compute the effects of the individual subroutines and propagate live register sets to the call-sites. In a second step, we refine and rectify our analysis by using more precise initial values for each control-flow sink. For return instructions, we obtain all possible call-sites from the call graph and use the union of their live register sets as the initial value for the return instruction. For other control-flow sinks, which are mostly jumps with unknown destinations, we now assume all registers to be live—a conservative, but correct approximation. We then iterate this second step (effectively an inter-procedural

[5] Note that the analysis runs backwards through the CFG. Consequently, these "mergers" are in fact control-flow diversions as introduced by conditional jumps.

analysis) until we reach a fixpoint for the live register sets. A proof of the existence of this fixpoint along with more details on the theory of data-flow analysis can be found in [19].

#### Deferrable Registers

As we only allow to defer registers within a basic block, we can keep the according analysis pretty simple. Beginning at each sensitive instruction, we iterate forwards over the surrounding basic block and record all **un**deferrable registers: At first, these are only the output registers of the sensitive instruction itself, then we add both input and output registers of all encountered instructions, up to and including the next sensitive instruction. For this second sensitive instruction, we emit the complement of the computed set, as this yields the registers that can be deferred from the previous sensitive instruction.

The complete structure that we emit for each sensitive instruction has already been presented in Figure 4.4 on page 34. In the analysis, we fill in the fields `vaddr`, `prev` and `common`. The other fields are initially set to 0 and are updated during code generation by the rewriter.

### 4.3.6 Static Rewriter

As motivated in Section 3.4.2, we implement register reallocation and rescheduling in the analysis phase, because here all necessary information for such operations is readily available. We limit our implementation of static rewriting to the analysis of the possible effects of the proposed techniques. In the following two sections, we shall therefore present how we determine potential applications of both register reallocation and rescheduling.

#### Register Reallocation

With register restoration, we substitute an irrelevant callee-saved register for a relevant caller-saved register, so that we need not preserve the latter during emulation.

We identify reallocatable registers and possible replacement registers for each sensitive instruction by scanning backwards and forwards through the containing basic block; both scans start at the currently examined sensitive instruction. We use two bitmasks to keep track of the register state: The first mask indicates which caller-saved registers are (still) live, whereas the second mask tracks the available callee-saved registers. Subsequently, we will refer to these masks as *live-mask* and *avail-mask* respectively.

In a backwards scan, we first find the beginnings of the lifetimes of the relevant caller-saved registers, which is indicated by an instruction that writes to such a register, but does not read it[6]. Alternatively, the scan ends if the live-mask is clear, indicating that no

[6] Strictly speaking, the lifetime starts at the assignment, regardless of whether the register is also read by the same instruction or not. For the two-address instructions of IA-32, however, we cannot reallocate only the output register but implicitly also reallocate the input operand, so that we also need to reallocate

more registers need to or can be reallocated. We also stop at the beginning of the basic block: If a register is still live here, we cannot reallocate it locally but have to also analyze preceding blocks. For this first analysis, we restrict ourselves to block-local reallocation.

During this backwards scan, we clear all bits in the avail-mask, whose according registers are read or written by the encountered instructions. This ensures that we only allocate registers that are available throughout the whole lifetime of a caller-saved register. Furthermore, we clear the bits for all registers that are implicit or fixed operands to the instruction in both bitmasks, because such registers cannot be reallocated. If an instruction marks the beginning of the lifetime of a caller-saved register $r$, we store the current avail-mask for later use: $r$ can be reallocated to any of the registers indicated by this bitmask. We then remove $r$ from the live-mask and continue to find the beginnings of the lifetimes of the remaining caller-saved registers.

Once the backwards scan is completed, we perform a similar scan forwards to find the end of the lifetimes of the caller-saved registers: The first instruction after which a register is no longer live signifies this end; we attached register liveness information to the instructions in a previously executed live registers analysis. At the end of the lifetime of a register $r$, we logically AND the avail-mask from the backwards scan and the current avail-mask to obtain another bitmask. The latter represents all callee-saved registers that can in fact be substituted for $r$ throughout its lifetime. Our current implementation emits the possible register reallocations into a logfile for manual inspection.

### Rescheduling

**General Approach** With rescheduling, we want to support two of the proposed optimizations: First, we want to reorder the instructions around a sensitive instruction, so that less caller-saved registers are live across the sensitive instruction. This removes the need to preserve the caller-saved register as described in Section 3.2.1. Our second goal is to cluster sensitive instructions so that we can amortize the preservation overhead across two or more such instructions by not restoring and saving common relevant registers between these instructions (see Section 3.3.1).

In both cases, rescheduling must not modify the semantics of the guest OS. We must ensure that all instructions read the same values from their respective operands in the rescheduled guest as in the original code. We therefore compute a data-flow graph (DFG), whose nodes are instructions and whose edges each represent one of the following data dependencies:

**true dependency** An instruction $i$ that reads a storage location, whose value was previously assigned at the instruction $j$, is *truly dependent* on $j$; we add the edge $(i, j)$ to the DFG.

the register before such an instruction.

**anti dependency** If an instruction $i$ overwrites a storage location that was previously read by an instruction $j$, we say that $i$ is *anti dependent* on $j$ and add the edge $(i, j)$ to the DFG.

**output dependency** Two instructions $i$ and $j$ are said to be *output dependent*, if both assign to the same storage location. If $i$ is executed after $j$ in the original guest OS, we add the edge $(i, j)$ to the DFG.

We call dependencies *register-carried* to indicate that the shared storage location is a register.

The (transitive) dependencies imply a partial order $\prec$ (we define $(i, j) \Rightarrow: i \prec j$; $i$ must execute after $j$) on the instructions, which we must honor during rescheduling. To avoid loops in the dependency chains, we limit the rescheduling approach to basic blocks and redirect dependencies from preceding or succeeding blocks to pseudo instructions at the beginning and the end of each basic block.

For each instruction $i$, the partial order partitions the set of instructions into three groups: Instructions $b$ with $i \prec b$ must be executed before $i$, whereas instructions $a$ with $a \prec i$ must be executed after $i$. For instructions $j$ with $j \not\prec i$ and $i \not\prec j$, we are free to schedule them either before or after $i$. We call the last group of instructions ($i$-)*concurrent*. Only concurrent instructions are interesting for rescheduling, because the ordering of the other instructions does not affect register livenesses at $i$. Figure 4.9 illustrates both dependencies between instructions and concurrent instructions.

We can now compute a lower bound for the set of live registers at $i$: For each register-carried true dependency between instructions $a$ and $b$ with $a \prec i$ and $i \prec b$, the respective register is unavoidably live during the execution of $i$: Its value must be assigned prior to $i$ and is inevitably accessed after $i$. To reach the first stated goal, the scheduling algorithm should therefore disregard the unavoidably live registers and find a schedule for the $i$-concurrent instructions, which minimizes the number of the remaining caller-saved live registers.

Regarding the second goal, the clustering of sensitive instructions, we only need to schedule the $i$-concurrent sensitive instructions together with $i$. Note that two instructions that are both $i$-concurrent need not be concurrent with each other (see Figure 4.9), so that often not all $i$-concurrent sensitive instructions can be scheduled together. This is solely caused by the dependencies between instructions and no limitation of our approach.

**Realization** We create the DFG from the annotated CFG by starting a forward and a backward scan through the basic block at each instruction $i$, so that we can obtain all dependencies. For this purpose we propagate the input and output bitvectors of the instruction $i$ through the block, compare them with the bitvectors of the current instruction, and add appropriate edges to the DFG: All of these are directed *against* the control-flow and represent the above *must execute after* relation $\prec$. We also add the reverse edges to simplify the following analysis.

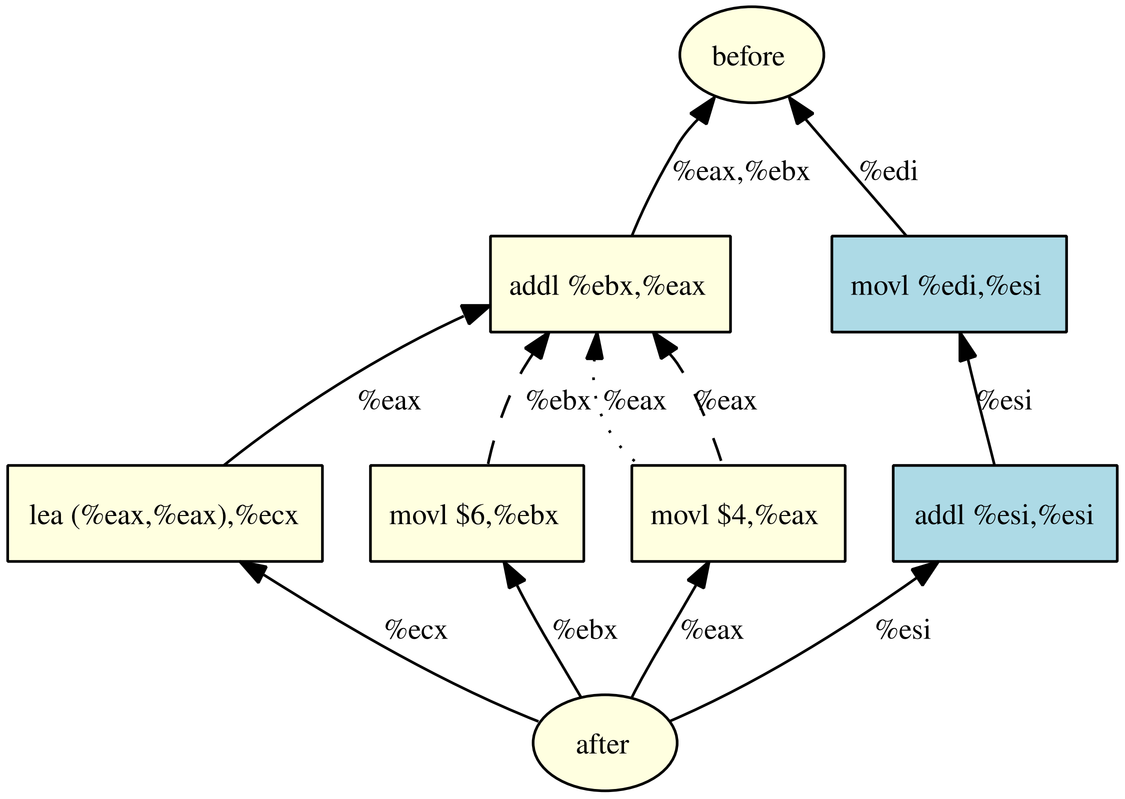

Figure 4.9: Example data-flow graph. Both shaded instructions are concurrent with all non-shaded instructions, but due to the true dependency on %ESI between them, they are not concurrent with each other. True dependencies are indicated with solid edges, anti dependencies with dashed and output dependencies with dotted edges; the edge labels denote the registers that carry the dependency. Additional dependencies to the "before" and "after" nodes and transitive dependencies have been omitted for clarity.

Based on this DFG, we then compute for each sensitive instruction $i$ the *before*-set of instructions that must be executed before $i$. This set is easily obtained as the transitive hull along the edges in the DFG. Similarly we compute the *after*-set as the transitive hull along the reverse edges. We then compute the $i$-concurrent set by subtracting the before- and after-sets from the set of all instructions in the basic block. Finally, we determine the unavoidably live registers by iterating over the after-set and checking the dependencies of each instruction: If a true dependency points to a node in the before-set, the carrying register is definitely live during the execution of $i$.

For each sensitive instruction, we then emit the unavoidably live registers, the live registers in the original schedule, and the locations and number of concurrent sensitive instructions into a logfile, thus documenting possible effects of the rescheduling technique.

# Chapter 5

# Evaluation

To prove the effectiveness of our enhancements, we conducted measurements on three scales: At first, we measured the runtime of virtualization code as generated in different contexts and according to different calling conventions, thus obtaining upper bounds for the possible improvements. We then verified the applicability of the proposed optimizations based on numbers we obtained during rewriting of a pre-virtualized Linux kernel. Finally, we performed a series of minibenchmarks to determine the effects of our enhancements on a larger scale. In the following sections we shall first describe the setup that we used for the measurements and then present the results.

## 5.1 Test Environment

All measurements presented in this chapter were performed on a 1.5 GHz Intel Pentium 4, stepping 10. The machine was equipped with 1 GB of main memory, a 256 kByte 8-way set associative second level cache, and an 8 kByte 4-way set associative first level data-cache, both with 64 bytes per cache line. For instructions an additional 8-way set associative trace-cache for 12k $\mu$-ops was present. During network benchmarks, we connected the described machine with an identical one via switched gigabit ethernet.

We setup the virtual machines on L4 Virtualization Technology [10] as depicted in Figure 5.1. We use the L4Ka::Pistachio microkernel as the hypervisor in our setup. L4 provides an abstract interface to the underlying machine in terms of threads, address spaces, *mapping* mechanisms for virtual memory management and inter-process communication facilities (IPC). Based on these abstractions, Marzipan implements a virtual machine monitor and provides basic virtualization infrastructure including console I/O, thread and memory management, pagefault handling and—most important—virtual machines. Furthermore, Marzipan distributes resources among its client virtual machines and loads guest OSs into the latter. To run the pre-virtualized Linux kernel, we instruct Marzipan to load the Afterburner as its guest OS; the Afterburner then loads and rewrites

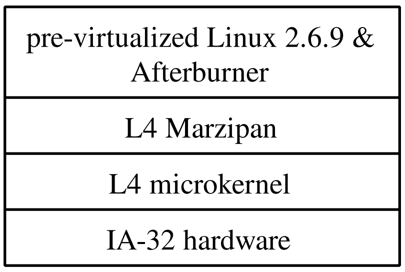

Figure 5.1: Virtualization environment. The pre-virtualized guest OS and the Afterburner execute together in a virtual machine, which is controlled by the Marzipan resource monitor. The virtual machines are hosted on the L4 microkernel, which is used as the hypervisor in this configuration.

the supplied kernel image and initiates its execution.

As we are not interested in measuring the quality of the emulation code, we configured the Afterburner to allow pass-through access to the devices for the Linux kernel. We limited the virtual machines to 128 MB of main memory. All testruns were conducted using the same statically linked kernel image; we did not use any loadable modules. To preclude further influences, we also used the same 20 MB ramdisk image throughout our tests, which was based on the ttylinux distribution [20]. For network benchmarks both machines used the same kernel and ramdisk images and were furthermore rewritten using the same options to the load-time rewriter.

## 5.2 Microbenchmarks

By using microbenchmarks we examined the absolute effects of our optimizations under well-defined conditions. For this purpose we manually implemented virtualization code as it would have been generated by the rewriter, but made it call an empty emulation routine. We compared the calling conventions of the original rewriter, which indirectly call the emulation routine via stubs, with our default conventions. Additionally, we evaluated the effects of discarded registers and compared different approaches to skip the unused scratch space using single- or multi-byte `nops`.

### 5.2.1 General Virtualization Code

We measured the runtime of the plain virtualization code in its different implementations by hand-coding it in assembly functions. These functions contained nothing but the code that is usually generated by the rewriter and a trailing `ret`. In order to determine the best-case improvements, we did not pad the code fragments to fixed “scratch space” lengths as would be done in the guest OS. The called emulation routine contained just two `nops` and a `ret` and was located on the same page that also contained the virtualization code. All functions were aligned to 32 byte boundaries to prevent the functions from crossing cache line boundaries.

We measured the runtime with the processor internal timestamp counter and emitted a serializing `cpuid` instruction before obtaining a timestamp. This ensures that the measured runtimes comprise the complete virtualization code, but also turns them into worst-case times despite the best-case conditions we provided: The functional units of the processor will be under-utilized due to serializing; in the real guest OS, the processor would start to process the following instructions earlier.

In Figure 5.2, we compare the original calling conventions (using stubs), a more efficient variant of them, and our default conventions, that inline the stub code. The figure conveys the number of cycles necessary for the execution of the complete virtualization code including the call to and immediate return from the empty emulation routine. Where applicable, we measured the runtime depending on the number of registers that are preserved across the call. For completeness, we also provide the numbers for 0 to 3 passed arguments, although none of them is actually accessed in the called function. As arguments we first pass %EAX, then the literal 4, and as the third parameter %EDX via the stack.

The measurements support our claim: Our new calling conventions are superior to the original ones, as they require 8–10 cycles less than the original code—-a reduction of 40–50 %. Furthermore, discarding irrelevant registers even reduces the cost of calling an emulation routine by as much as 60 %.

One call/return pair costs about 8 cycles (the baseline cost of our default conventions); both indirect approaches require two such pairs and expose a consistent baseline cost of just over 16 cycles. Accordingly, we conclude that the improvements of our new conventions are dominated by the removed control-flow indirection.

An interesting effect is the stagnation of the runtime of the virtualization code for few saved registers: In these cases, the virtualization code comprises only a handful of instructions: two per preserved register, one per argument plus an additional instruction to clean up the stack, one `call` and one `ret` instructions. As we enclose this fragment in serializing instructions to measure its complete runtime, the stagnation supports our conclusion that the execution time is dominated by the high-latency call/return instructions. The processor-internal out-of-order engine can schedule the additional instructions for argument passing to execute in parallel to these instructions, thus effectively hiding their presence.

We attribute the slight reduction of runtimes for increased numbers of passed arguments, which is apparent in most of the presented cases, to imprecise measurements.

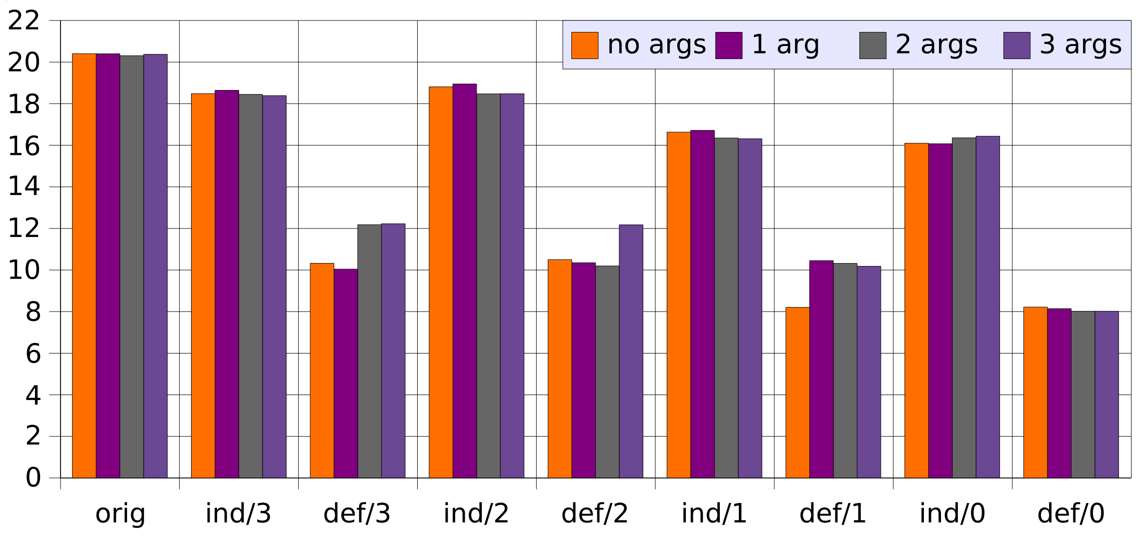

Figure 5.2: Comparison of different calling conventions. The y-axis indicates the clock cycles per call of an empty emulation routine. The x-axis lists the different calling conventions and contexts. Within each group, the individual bars represent 0–3 arguments that are passed to the emulation routine.
orig: original calling conventions (indirect via stubs), ind/$n$: our more efficient variant of the original conventions, def/$n$: the default conventions used by our rewriter (no stubs), $n$: number of registers that are preserved by the virtualization code

## 5.2.2 Unused Scratch Space

Our second microbenchmark evaluates different strategies to deal with unused scratch space. We compared four approaches: At first, we filled the unused space with single-byte `nops` as did the original implementation. Our second approach uses multi-byte `nops`, such as `movl %esi,%esi` for 2 bytes or `lea 0(%edi),%edi` to cover 3 or 6 bytes, depending on the size of the displacement. In both cases we then measured the effects of a leading `jmp` instruction, which replaces the first two bytes and jumps right over the unused space.

To preclude other influences besides the executed code, we used a page-aligned 4 kByte array to represent the scratch space. For each test, we filled this array from its beginning with proper `nops` and a trailing return and called this self-made function via a function pointer. The overhead of the function call is included in the runtimes we present in Figure 5.3. In accordance with [8], multi-byte `nops` enable faster execution of a given number of bytes than single-byte `nops`. 12 bytes can be efficiently skipped using two 6-byte `nops`, whereas 10 and 11 bytes must be combined from three instructions covering 6+3+1 or 6+3+2 bytes.

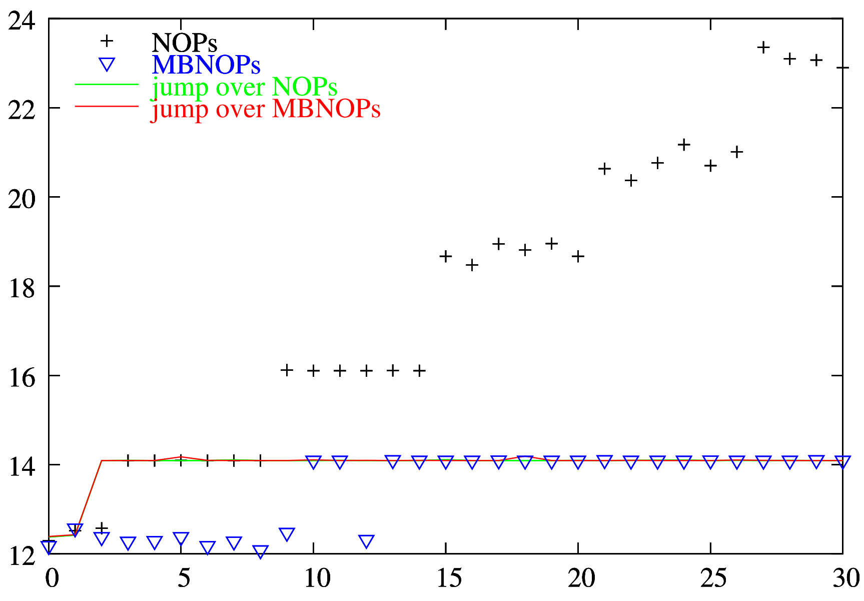

Figure 5.3: Comparison of different approaches to handle unused scratch space. The x-axis gives the number of bytes to skip, the y-axis gives the required time in clock cycles.

We conclude that few large areas can be skipped more efficiently than many small ones by covering the whole area with as few multi-byte `nops` as possible and jumping over regions of more than 15 bytes.

## 5.3 Applicability

To evaluate the applicability of our optimizations, we collected statistics about the generated code during rewriting. According to these statistics, the examined Linux 2.6.9 kernel contains 1,500–4,800 sensitive instructions—depending on the number of included device drivers. The kernel that we used in our benchmarks comprised 4733 sensitive instructions, including 994 `ud2` instructions to debug the kernel and 793 instructions with inlined emulation code. A total of 64 sensitive instructions—instances of `iret`, `int`, long jumps and `cpuid`—could not be ported to our new calling conventions: These either jump to their emulation routine instead of calling it or require a special assembly stub.

The remaining 2882 instructions can effectively benefit from our optimizations, although 708 of them already used standard C calling conventions (without stubs)[1]. Consequently, the latter do not profit from our new conventions but are still improved by discarding irrelevant registers. We will now present details on the applicability of all of

[1] These direct calls result primarily from memory mapped I/O operations.

| discarded registers | 0 | 1 | 2 | 3 | total | **revised** | |
|---|---|---|---|---|---|---|---|
| call sites | 592 | 865 | 1042 | 383 | 2882 | | |
| total of discarded registers | 0 | 865 | 2084 | 1149 | 4098 | 3628 | 43.9 % |
| total of preserved registers | 1552 | 1612 | 1006 | 0 | 4170 | 4640 | 56.1 % |
| preserved register on stack | 1488 | 1400 | 437 | 0 | 3325 | 3795 | 81.8 % |
| preserved registers in registers | 64 | 212 | 569 | 0 | 845 | 845 | 18.2 % |

Table 5.1: Number of discarded caller-saved registers per call-site.

the proposed techniques.

### Discarded Irrelevant Registers

The statistics from the rewriter reveal that more than 40 % of the caller-saved registers can be discarded due to the provided context-information. With 3 caller-saved registers being saved and restored around the emulation this results in an average of $0.4 \cdot 3 \cdot 2 = 2.4$ saved instructions per emulation call. Table 5.1 provides more detailed information, but disregards a total of 470 registers, which are first considered to be irrelevant but are later on pushed onto the stack as arguments nonetheless. These must be subtracted from the discarded registers and counted as preserved registers to get the statistics right; we give the revised figures as well.

From the table we further infer that we can discard at least one register at 2290 out of the 2882 call-sites, which is more than 79 %. At 49 % of the call-site we can discard two or more registers, and at 13 % of all call-sites we can even discard all three caller-saved registers.

### Caller-Saved Registers Stored in Callee-Saved Registers

Table 5.1 also conveys the applicability of saving caller-saved registers in callee-saved ones: 845 out of 4640 preserved registers, about 18 %, need not be written to memory but can be preserved in a register.

### Locally Adjusted Calling Conventions

We apply modified calling conventions only for the emulation routine of the `iret` and `cpuid` instructions: The provided emulation takes all registers as arguments on the stack and has them restored afterwards by the virtualization code, so that the callee does not need to preserve any registers. Our new conventions prevent the 4 callee-saved registers %EBX, %ESI, %EDI and %EBP from being saved and restored in the emulation routine unnecessarily.

Our measurements attest a decreased runtime for the emulation of `iret` in an idle system (we measured only the C emulation routine, the assembler part with the actual `iret` is not included): Without embracing the emulation in serializing instructions, the runtime goes down from 112 to 104 cycles (-7 %); embraced in `cpuid` instructions, we see a decrease from 568 to 560 cycles (-1.4 %). The latter measurement includes the execution of the serializing `cpuid` instruction.

Virtualized `cpuid` instructions are not executed often enough to obtain reliable results; the results would not even be interesting for the same reason.

### Reallocated Registers

We analyzed our test kernel for beneficial applications of the proposed register reallocation technique, but found only 26 possible locations. Most of these are present either in `cpu_init` or e1000 driver routines and are not on a frequently executed critical path.

### Statically Evaluated Dispatch Tables

For port I/O, we can determine the accessed port in 100 of 338 cases ($\approx$30 %) and therefore potentially circumvent a dynamic dispatcher. But as the according devices, mainly the VGA graphics controller, are not yet virtualized in the framework we used, actually none of the 100 possible cases results in a statically dispatched emulation routine. We cannot present figures for memory mapped I/O, because no static dispatcher was available. Concluding, we were unable to determine the effects of static dispatching due to an incomplete test environment.

### Deferred Register Restorations

We previously claimed that this approach mostly applies to routines for device I/O. The rewriting statistics substantiate this statement: In our testkernel we can defer 430 register restorations. The annotated e1000 driver alone contributes 113 deferrable registers; floppy disk (105), VGA (60) and XT-PIC (20) also play a major role. The remaining applications occur mostly in conjunction with processor identification and setup and the CMOS real-time clock.

### Rescheduled Instruction Streams

With rescheduling we envisage two goals: Firstly, we want to reduce the number of caller-saved registers that are live during the execution of a sensitive instruction. Secondly, we want to cluster sensitive instructions to support the deferred register restoration approach.

Our analysis on rescheduling reveals 281 possible applications, at which we can reduce the number of live caller-saved registers by one. On additional 20 locations the proposed technique can even remove the need to preserve two such registers across the

| patches per basic block | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | >9 |
|---|---|---|---|---|---|---|---|---|---|---|
| applications | 2034 | 502 | 164 | 55 | 32 | 15 | 5 | 10 | 5 | 23 |
| average size [bytes] | 8 | 16 | 19 | 38 | 44 | 55 | 64 | 68 | 69 | 150 |

Table 5.2: Results of basic block compaction.

emulation of the sensitive instruction. More detailed analysis of these locations disclosed that only one of the possible applications is contained in a frequently executed function: In `schedule` we can discard two additional registers using rescheduling.

The second goal of rescheduling, clustering of sensitive instructions, can only be achieved once in a frequently executed function, namely in `system_call`. Unfortunately, even without this clustering, we can defer two registers across the two affected sensitive instructions, so clustering cannot improve the situation significantly.

We conclude that the static rewriting techniques can be applied in our testkernel, but are not likely to yield significant improvements.

### Compacted Basic Blocks

Basic block compaction moves unused scratch space to the end of the containing basic block with two intentions: For one, this approach combines the unused scratch space of all patches within a basic block, so that only one large space must be skipped instead of multiple smaller ones. In Table 5.2 we therefore present how often we combined the unused scratch space of any number of patches. Furthermore, we also give the average size of the resulting "large" space of unused bytes. About 70 % of the basic blocks contain only a single patch, so that no combining is possible; in 17 % of all cases we can combine 2 patches.

The second reason for moving the unused space to the end of each block is that many basic blocks end in conditional or even unconditional branches or returns. For backward jumps or returns the overhead of skipping the unused space is therefore completely removed, only the fall-through code and forward jumps are still affected (see Section 5.2.2).

Again, we instrumented our rewriter to determine the type of the last instruction in the compacted blocks. We found out that about 14 % of them end in unconditional branches (98 % of these are directed backwards), and further 9 % end in a `return` instruction. Conditional jumps mark the end of additional 29 % of the compacted blocks, but only 11 % of these are directed backwards. The remaining 48 % of the blocks fall through to the next block.

To sum this up, in about 75 % of the cases we still need to skip over the unused space. We can avoid doing so in only 25 % of the cases due to returns or unconditional and taken conditional jumps backwards.

## 5.4 Minibenchmarks

To evaluate the effects of our enhancements on a real operating system we conducted a series of benchmarks on a pre-virtualized Linux 2.6.9. The presented results were mostly obtained from tests taken from the lmbench3 and netperf test suites. We will conclude the evaluation with an analysis of the execution frequencies of sensitive instructions during some of the benchmarks.

### 5.4.1 Latency Tests

In a first series of benchmarks we measured the effects of our new calling conventions and optimizations on latencies as reported by a subset of the lmbench3 suite [16]. We obtained the results in three rounds, during each of which we executed the whole set of tests five times in a row. We rebooted the machine after each of the three rounds. The given figures are based on the median of all 15 measurements.

Figure 5.4 presents the relative performance of different calling conventions and optimizations as compared to the original indirect calling conventions. *ignore* implements the new conventions, but disregards all available context-information. In the *default* case, the rewriter uses context-information to discard irrelevant registers, but fills unused scratch space with single-byte `nops` and does not perform basic block compaction. The *nop* case implements efficient padding of unused space with multi-byte `nops`, whereas *skip+nop* moreover adds a jump over the multi-byte `nops`. The *compact* case shows the effects of basic block compaction, filling the scratch space with single-byte `nops` again. Finally, *all* implements all proposed optimizations: It performs basic block compaction, fills the space with multi-byte `nops`, defers register restorations, and preserves registers in callee-saved registers.

The remaining two cases, *indirect* and *indirect+*, represent our improved variant of the original calling conventions, which also indirectly call the emulation routine via stubs. *indirect* ignores the provided context-information, whereas *indirect+* queries it and uses one of 8 stubs to discard registers at each call-site.

Disregarding the *indirect* variants for the moment, our measurements show that all benchmarks benefit from our optimizations. Although many tests only improve by about 0.5 %, which is hardly significant, the stat and open/close syscalls as well as TCP/IP connects to localhost (lat_connect) show a reduction of about 1 % in the measured latencies. File locking via fcntl (lat_fcntl), signal catching and local communication via UNIX pipes even gain nearly 2 %.

The *ignore* case shows that, in most cases, more than 50 % of the beneficial effects originate from the new, direct calling conventions; discarding registers does not yield consistent or significant effects. Of all implemented optimizations, the multi-byte `nops` are generally the most influential one, closely followed by basic block compaction (*compact*). The combination of both of them clearly dominates the effects of the *all* case. Register-

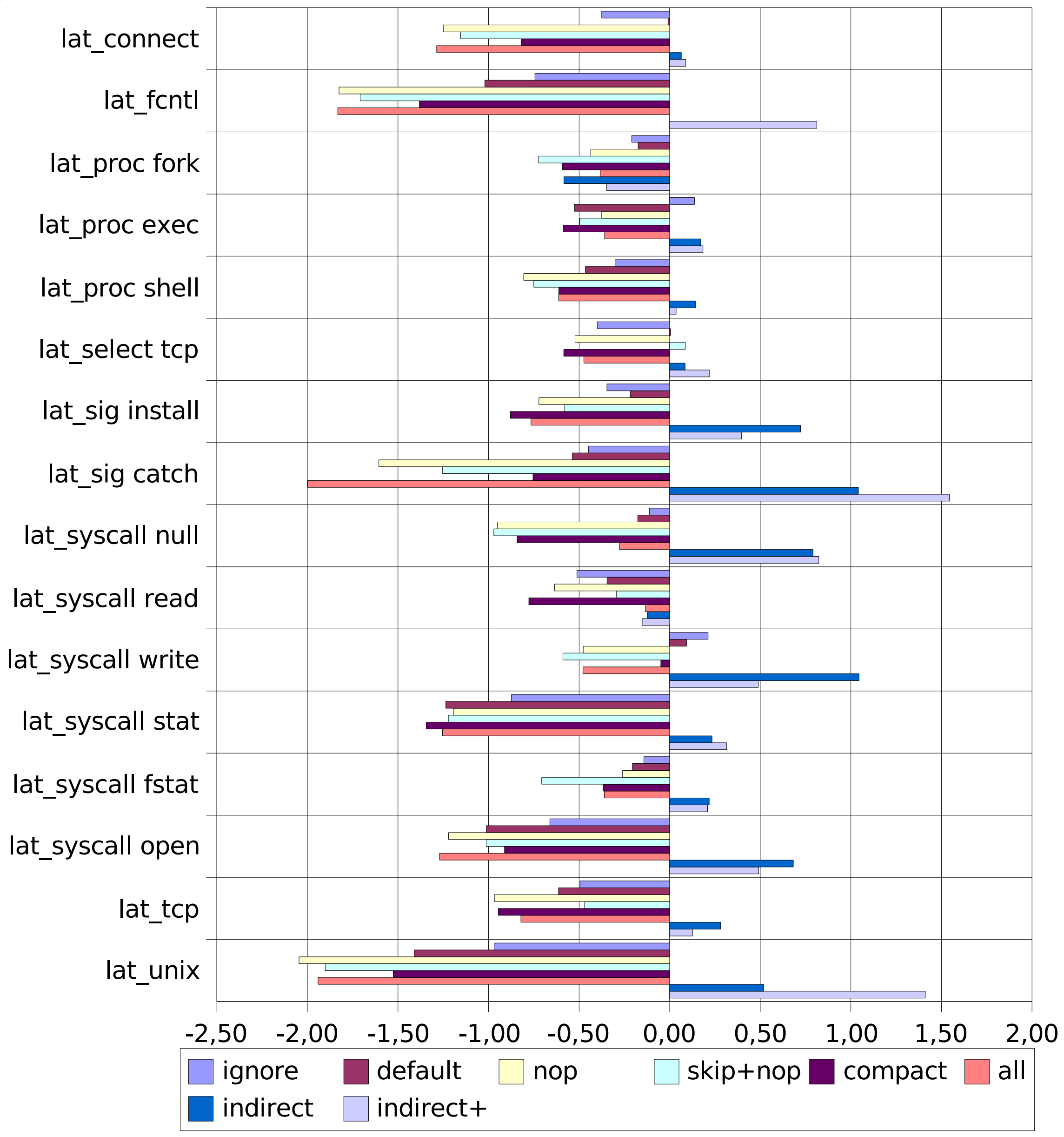

Figure 5.4: Benchmark results on latencies. The figures are differences in per cent, the reference measurements are those obtained with the original, indirect calling conventions. All benchmarks measure latencies, so smaller values are better.

to-register copies or deferring registers do not seem to contribute much, otherwise the effects of *all* should be significantly different from the pure *nop* and *compact* cases.

Lastly, both *indirect* and *indirect*+ generally perform worse than the original indirect approach. We attribute this observation to the larger glue code: For both approaches we supply 8 stubs per emulation routine; combined with 25 emulation routines and 16 bytes per stub this makes up for 3200 byte of stub code—nearly a page of its own. Compared to the original approach, which only requires 400 bytes for the stubs, the execution of many emulation routines will now require one additional TLB entry for the page containing the stub. Consequently, overall system performance degrades firstly due to TLB faults on the stubs and secondly when reestablishing a potentially evicted TLB entry.

*indirect*+ performs significantly worse than the *indirect* approach in three of the tests, namely file locking, signal catching and communication via UNIX pipes. We relate this effect to the increased working set of the *indirect*+ approach: Both the original approach and *indirect* only use one stub per emulation routine—the one that preserves all caller-saved registers. *indirect*+ selects one of the 8 provided stubs depending on the context of the call-site, consequently this approach touches more code fragments and more cache lines and floods the trace-cache with more $\mu$-ops. The larger cache footprints might cause the eviction of "hot" entries from the caches and thus negatively affect the overall performance.

To summarize our evaluation of the latency tests, our optimizations improve 3 of the 16 presented tests by about 2 %, additional 3 by about 1 %, and leave the remaining tests virtually unaffected.

### 5.4.2 Bandwidth Tests

In the second and final set of benchmarks we compared the previously presented configurations regarding (network) bandwidths. In Table 5.3 we give the results of four benchmarks from the netperf benchmark suite [9] and a pingpong (or hot potato) benchmark, which repeatedly transfers a single byte via a pipe between two processes. UDP_STREAM and TCP_STREAM transfer data from the test machine (client) to the server, whereas TCP_MAERTS does the same, but from server to client. TCP_CRR connects to the server, sends a request and waits for the reply. The presented figures are averages over 9 results, which were obtained in three rounds with three test runs each. The machines were rebooted after every round. The netperf tests were setup to run 10 seconds and to achieve a confidence interval of 3 % at a confidence level of 99 %.

UDP_STREAM and pingpong show reliable improvements of 1.3–1.7 %. All TCP-related measurements suffered from high variances, which is expressed in fairly long confidence intervals (2.18–6.49 %). Therefore the results are inconclusive, but indicate no significant improvement.

| | original | ignore | default | nop | all |
|---|---|---|---|---|---|
| UDP_STREAM [MBit/s] | 729 | 733 | 731 | 734 | 741 |
| difference [%] | @ 0.61 | +0.6 | +0.3 | +0.7 | +1.7 |
| TCP_STREAM [MBit/s] | 684 | 669 | 688 | 678 | 686 |
| difference [%] | @ 2.18 | -2.2 | +0.6 | -0.9 | +0.4 |
| TCP_MAERTS [MBit/s] | 704 | 684 | 704 | 696 | 720 |
| difference [%] | @ 2.52 | -2.8 | ±0.0 | -1.2 | +2.3 |
| TCP_CRR [connects/s] | 1788 | 1735 | 1803 | 1791 | 1660 |
| difference [%] | @ 6.49 | -3.0 | +0.8 | +0.2 | -7.2 |
| pingpong [kByte/s] | 22.7 | 22.9 | 22.9 | 23.0 | 23.0 |
| difference [%] | @ 0.55 | +0.9 | +1.0 | +1.3 | +1.3 |

Table 5.3: Benchmark results on bandwidths. We give the absolute throughput as reported by netperf (or pingpong) and the relative performance as compared to the original calling conventions. In the first column, we give the intra-row maximum 95 %-confidence interval in percent of the absolute throughput instead of relative performance. The columns represent different calling conventions and optimizations.

## 5.5 Execution Profiles

We conclude our evaluation with execution profiles of the previously discussed benchmarks. In an extra run of the tests, we injected an additional increment instructions before each sensitive instruction, so that we could record the execution frequencies of all sensitive instructions. Table 5.4 gives the results, expressed as the percentage of each sensitive instruction type during the execution of the listed benchmarks. At the bottom we also provide total numbers of sensitive instructions executed and clock cycles spent during the execution of the tests. The last row conveys the average number of clock cycles between two sensitive instructions.

These figures provide a possible explanation for the mediocre benchmark results: From the commonly executed instructions, only `pushf`, `popf`, `iret` and `cli` actually profit from our optimizations. The others are emulated inline and do not use the new calling conventions, to which attributed about 50 % of the effects on latencies.

From Table 5.4 we infer that for lat_connect and lat_unix, which could both be improved by 1.5–2 %, about 50 % of the executed sensitive instructions are emulated externally. As both tests yield an average of about 2,300 clock cycles between two sensitive instructions, this results in an application of our optimized calling conventions about once every 4,600 clock cycles. Similarly, about 60 % of the sensitive instructions in UDP_STREAM are emulated externally; with one sensitive instruction every 1,258 cycles, an external emulation routine is called about once every 2,100 cycles using our improved conventions.

| | lat_connect localhost | lat_unix | lat_syscall null | lat_syscall write | UDP_ STREAM | TCP_ CRR |
|---|---|---|---|---|---|---|
| cli† | 25.83 | 24.80 | 12.95 | 12.69 | 27.88 | 25.09 |
| pushf | 31.59 | 19.97 | 0.56 | 0.21 | 33.46 | 24.94 |
| popf | 18.91 | 19.97 | 0.56 | 0.20 | 25.88 | 18.14 |
| push %DS† | 2.77 | 3.20 | 12.18 | 12.35 | 0.55 | 2.50 |
| pop %DS† | 2.77 | 3.20 | 12.18 | 12.35 | 0.55 | 2.50 |
| iret | 2.77 | 3.20 | 12.18 | 12.35 | 0.55 | 2.50 |
| move to %DS† | 2.77 | 3.20 | 12.18 | 12.35 | 0.55 | 2.50 |
| pop %ES† | 2.77 | 3.20 | 12.18 | 12.35 | 0.55 | 2.50 |
| move to %ES† | 2.77 | 3.20 | 12.18 | 12.35 | 0.55 | 2.50 |
| push %ES† | 2.77 | 3.20 | 12.16 | 12.35 | 0.55 | 2.50 |
| sti | 4.14 | 3.22 | 0.19 | 0.11 | 1.99 | 5.93 |
| move to %CR3 | 0.00 | 3.17 | 0.00 | 0.00 | 0.00 | 0.00 |
| move from %FS† | 0.00 | 3.17 | 0.00 | 0.00 | 0.08 | 0.67 |
| move from %GS† | 0.00 | 3.17 | 0.00 | 0.00 | 0.08 | 0.67 |
| out | 0.08 | 0.08 | 0.29 | 0.21 | 1.87 | 3.38 |
| e1000 write | 0.00 | 0.00 | 0.00 | 0.00 | 3.65 | 1.33 |
| $10^3$ SIs | 28,731 | 27,783 | 1,128 | 10,202 | 85,101 | 9,664 |
| $10^6$ cycles | 64,228 | 63,952 | 9,728 | 61,986 | 107,078 | 22,535 |
| cycles/SI | 2,235 | 2,302 | 8,624 | 6,076 | 1,258 | 2,332 |

Table 5.4: Benchmark profiles with respect to sensitive instructions. The figures convey the relative execution frequencies of each type of sensitive instructions (SIs) per benchmark. Below we give the total number of executed sensitive instructions and the total number of cycles spent during execution of the tests. Instructions marked† are emulated inline.

For the lat_syscall tests, only about 13 % (or every 7th) of the executed sensitive instructions are emulated externally, raising the number of cycles between two applications of our optimizations to 42,000–60,000. The well-improved tests execute sensitive instructions about ten times more frequently than the others, which might be the cause for their stronger reaction to our optimizations.

Unfortunately, the reverse does not hold: TCP_CRR has about 50 % of its sensitive instructions emulated externally, which are therefore executed about once every 4,600 cycles. This matches the figures for lat_connect and lat_unix, so TCP_CRR should also benefit from our optimizations. The measurements we performed were inconclusive, but indicated rather a degradation (-7 %) than a positive trend.

To conclude our analysis of the execution profiles, we shall give an estimation of the expected effects of our optimizations: As shown earlier, our new calling conventions save 8–10 cycles per application as compared to the original conventions. With a single application of these less than every 4,600 cycles, we should expect improvements on runtime of below 0.2 %. We attribute the improvement of 1–2 % for certain tests to non-uniform distribution of sensitive instructions.

# Chapter 6

# Conclusion

In the following sections we shall briefly summarize this thesis and point out open questions. We shall then provide ideas for future work based on the results of this thesis.

## 6.1 Summary

This thesis was motivated by the observation that the pre-virtualization approach to virtualizing IA-32 hardware used static virtualization code to replace sensitive instructions. We reasoned that this approach incurs unnecessarily high overhead in many cases due to unnecessarily preserved registers and proposed to dynamically create more specific virtualization code on a per-instruction basis.

We have therefore provided techniques that use context-information like *live registers* for sensitive instructions to generate efficient virtualization code. To increase the applicability and effectiveness of the proposed optimizations, we proposed and evaluated additional, static rewriting techniques—namely register reallocation, rescheduling, and modified calling conventions for emulation code. We then described how the context-information can be obtained solely from the binary by means of static program analysis. To overcome the problem of incomplete knowledge about the structure of the analyzed guest OS, which is caused by indirect branches, we provided safe approximations for the context-information where necessary.

The evaluation of our implementation was two-sided: On the one hand, we confirmed that the proposed techniques actually do improve the quality of the generated virtualization code—about 40 % of the previously saved registers have been identified as irrelevant and are discarded. On the other hand, the majority of the conducted benchmarks did not show statistically significant differences between optimized and unoptimized guest OSs.

## 6.2 Open Issues

We conclude that the presented optimizations are incompatible with the microarchitecture of the test machines: Our optimizations require a monotonic increasing mapping of executed instructions to clock cycles spent, because most of them are designed to reduce the number of instructions in the generated code. Our microbenchmarks showed that this prerequisite is not always fulfilled by the NetBurst Microarchitecture, which is implemented in the Pentium 4 that served as our test machine, due to out-of-order execution.

A second limitation to the possible effects of our optimizations results from the small register set of IA-32 machines: With 8 registers and only 3 caller-saved ones, the possible effects of of discarding caller-saved registers rather than preserving them are tightly bounded.

Finally, we reasoned that the low execution frequency of non-inlined virtualization code—about once every 4,000 clock cycles—presumably limits the effects on runtime of our optimizations to less than 0.2 %.

## 6.3 Outlook

Future work is required to evaluate our system on different implementations of IA-32, such as the Pentium III or the Opteron, to verify or falsify our conclusion about the Pentium 4 microarchitecture. Also porting our system to architectures like IA-64 or even RISC architectures, such as PowerPC or MIPS, with a larger register set seems to be worth exploring.

Furthermore, we believe that the analysis component can be reused in a more complex rewriting tool, that does not rewrite individual sensitive instructions but rather complete subroutines. Such a system could even abolish the preparation phase of pre-virtualization by finding and rewriting sensitive instructions fully-automatically, thus moving towards binary translation. Only annotations for sensitive memory operations would still be required.

Apart from virtualization, dynamic binary instrumentation tools might use the information from our analysis to efficiently integrate the instrumentation code into their target.

# Appendix A

# Examples of IA-32 Glue Code

This appendix lists three different calling conventions between the rewritten guest OS and the emulation routines. For each convention we give the rewritten guest OS code, which replaced the sensitive instruction being emulated, and the glue code involved in the call. We also provide a snapshot of the stack right after the trampoline has been called. This shall demonstrate why indirect approaches require additional indirection when accessing their argument, whereas inlined stub code can efficiently use standard C conventions for parameter passing.

The examples below show the general case of $N$ arguments being passed to the emulation routine, typical emulation code requires $0 \leq N \leq 3$ parameters. We additionally demonstrate the special cases that one of the arguments is a caller-saved register or the return address (RA) to the guest OS.

## A.1 Original Calling Conventions

Rewritten guest OS:

```
  pushl  arg{N-1}
  ...
  pushl  arg0
  call   stub
RA_guest:
  addl   $4N, %esp
```

Stub code:

```
stub:
  /* save caller-saves */
  pushl  %eax
  pushl  %ecx
  pushl  %edx
  /* argument to trampoline */
  pushl  %esp
  subl   $8, (%esp)
  call   trampoline
RA_stub:
  /* remove argument */
  addl   $4, %esp
  /* restore caller-saves */
  popl   %edx
  popl   %ecx
  popl   %eax
  ret
```

Stack layout:

```
  arg{N-1}
  ...
  arg0
  RA_guest
  %eax
  %ecx
  %edx
  pointer to "RA_stub"
  RA_stub  <-----+
```

Trampoline used to access the arguments:

```
typedef struct {
  uint32_t  stub_ra;
  uint32_t  frame_pointer;
  uint32_t  edx;
  uint32_t  ecx;
  uint32_t  eax;
  uint32_t  guest_ra;
  uint32_t  param[0];
} burn_clobber_frame_t;

void
trampoline( burn_clobber_frame_t *frame )
{
  emulation( frame->param[0], frame->param[1], ...,
             frame->param[N-1], &frame->edx,
     frame->guest_ra );
}
```

This approach requires a fixed structure of the burn_clobber_frame, which hampers discarding of irrelevant caller-saved registers: The stack layout must match the burn_clobber_frame and therefore include all caller-saved registers' slots.

## A.2 Efficient Calling Conventions Using Stubs

We propose the following variant of the above scheme to allow discarding irrelevant caller-saved registers.
Rewritten guest OS (unmodified):

```
  pushl  arg{N-1}
  ...
  pushl  arg0
  call   stub
RA_guest:
  addl   $4N, %esp
```

We provide a set of stubs, each saving a different subset of the caller-saved registers. Lines like `?pushl %eax?` indicate that the enclosed code is only present in some of these stubs. The number of saved caller-saved registers is known for each stub and symbolized with an $X$ in the following stub code:

```
stub:
  /* register arguments */
  pushl  %edx
  /* save relevant caller-saves */
  ?pushl %eax?
  ?pushl %ecx?
  /* argument to trampoline */
  lea    4*X(%esp), %eax
  call   trampoline
RA_stub:
  /* restore caller-saves */
  ?popl  %ecx?
  ?popl  %eax?
  /* restore register arguments */
  popl   %edx
  ret
```

Stack layout:

```
  arg{N-1}
  ...
  arg0
  RA_guest
  %edx    <----- %eax
  ?%eax?
  ?%ecx?
  RA_stub
```

Trampoline used to access the arguments:

```
void __attribute__(( regparm(1) ))
trampoline( uint32_t *args )
{
  emulation( args[2], args[3], ...,
             args[N+1], &args[0], args[1] );
}
```

Using this approach, we use the argument to the trampoline to skip over the variable part of the stack. The pointed to region on the stack only contains items that are always necessary and in a fixed order: First, the $N$ arguments are pushed onto the stack, then the stub gets called, which implicitly pushes the return address to the guest OS on the stack. In the stub, we first push all caller-saved registers that have been marked as arguments to the emulation (regardless of whether they are relevant or not). Afterwards, we are free to preserve the relevant caller-saved registers—or refrain from doing so if they may be discarded. We then pass the address of the fixed frame to the trampoline—in %EAX to avoid an additional stack access.

Besides allowing to discard irrelevant registers, this approach also saves two instructions in passing the argument to the trampoline.

## A.3 Inlined Stubs

Both previous approaches implied both an additional control-flow indirection via a stub and an additional indirection for accessing the arguments within the trampoline. Inlining the complete stub code into the guest OS removes both indirections and further reduces the number of instructions needed to call the emulation:
Rewritten guest OS:

```
  /* save relevant caller-saves */
  ?pushl %eax?
  ?pushl %ecx?
  /* register arguments */
  pushl  %edx
  /* arguments */
  pushl  arg{N-1}
  ...
  pushl  arg0
  call   trampoline
RA_guest:
  addl   $4N, %esp
  /* restore register arguments */
  popl   %edx
  /* restore caller-saves */
  ?popl  %ecx?
  ?popl  %eax?
```

Stack layout:

```
  ?%eax?
  ?%ecx?
  %edx
  arg{N-1}
  ...
  arg0
  RA_guest
```

Trampoline used to access the arguments:

```
void
trampoline( uint32_t arg0, uint32_t arg1, ...,
            uint32_t arg{N-1}, uint32_t edx )
{
  emulation( arg0, arg1, ..., arg{N-1}, edx,
             __builtin_return_address(0) );
}
```

We reordered the stack frame layout, so that we can call the trampoline along the usual C calling conventions—no further indirections are required. We therefore first preserve all relevant caller-saved registers (unless they are explicitly used as arguments to the emulation), then pass the explicit register arguments, followed by the $N$ “variable” arguments: The latter depend on the operands of the replaced sensitive instruction.

# List of Figures

1.1 Structure of a virtual machine environment. . . . . . . . . . . . . . . . . . 2
1.2 Virtualized sensitive instruction. . . . . . . . . . . . . . . . . . . . . . . . 3

3.1 Caller-saved registers preserved in callee-saved registers. . . . . . . . . . 17
3.2 Effects of register reallocation. . . . . . . . . . . . . . . . . . . . . . . . 19
3.3 Static dispatch. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.4 Deferred register restoration. . . . . . . . . . . . . . . . . . . . . . . . . 21
3.5 Rescheduling. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.6 Basic block compaction. . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.7 Structure of the proposed system. . . . . . . . . . . . . . . . . . . . . . . 25

4.1 Generated virtualization code. . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2 Internals of the register usage tracker. . . . . . . . . . . . . . . . . . . . 31
4.3 Indirectly calling emulation routines. . . . . . . . . . . . . . . . . . . . . 32
4.4 Structure used to support deferred register restoration. . . . . . . . . . . 34
4.5 Descriptor for generated glue code. . . . . . . . . . . . . . . . . . . . . . 39
4.6 Structure of our analysis component. . . . . . . . . . . . . . . . . . . . . 40
4.7 Incorrectly disassembled instructions. . . . . . . . . . . . . . . . . . . . . 41
4.8 Sample data-flow analysis. . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.9 Example data-flow graph. . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

5.1 Virtualization environment. . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.2 Comparison of different calling conventions. . . . . . . . . . . . . . . . . 56
5.3 Comparison of different approaches to handle unused scratch space. . . . 57
5.4 Benchmark results on latencies. . . . . . . . . . . . . . . . . . . . . . . . 62

# List of Tables

5.1 Number of discarded caller-saved registers per call-site. . . . . . . . . . . . 58
5.2 Results of basic block compaction. . . . . . . . . . . . . . . . . . . . . . . . 60
5.3 Benchmark results on bandwidths. . . . . . . . . . . . . . . . . . . . . . . . 64
5.4 Benchmark profiles with respect to sensitive instructions. . . . . . . . . . . 65

# Bibliography

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP)*, pages 164–177. ACM Press, October 2003.

[2] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX Association, April 2005.

[3] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] Bruno De Bus, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. The design and implementation of FIT: a flexible instrumentation toolkit. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 29–34. ACM Press, June 2004.

[5] ECMA. *ECMA-335: Common Language Infrastructure (CLI).* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002.

[6] Hideki Eiraku and Yasushi Shinjo. Running BSD kernels as user processes by partial emulation and rewriting of machine instructions. In *Proceedings of BSDCon 2003*, pages 91–102. USENIX Association, September 2003.

[7] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of $\mu$-kernel based systems. In *Proceedings of the 16th Symposium on Operating System Principles (SOSP)*, pages 66–77. ACM Press, October 1997.

[8] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*, June 2005.

[9] Rick Jones. Netperf benchmark. Available from http://www.netperf.org/netperf/NetperfPage.html.

[10] The L4Ka Team. Marzipan: The L4Ka virtual machine environment. Available from http://l4ka.org/projects/virtualization/resourcemon/.

[11] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300. ACM Press, June 1995.

[12] Kevin Lawton. The bochs IA-32 emulator project. Available from http://bochs.sourceforge.net/.

[13] Joshua LeVasseur. Pre-virtualization with compiler afterburning. Available from http://l4ka.org/projects/virtualization/afterburn/.

[14] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[15] Tim Lindholm and Frank Yellin. *The Java$^{TM}$ Virtual Machine Specification*. Addison-Wesley, 1996.

[16] Larry McVoy and Carl Staelin. LMbench - tools for performance analysis. Available from http://www.bitmover.com/lmbench/.

[17] Microsoft. Virtual PC 2004, 2004. Information available from http://www.microsoft.com/windows/virtualpc/.

[18] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[19] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[20] Pascal Schmidt. ttylinux. Information and distribution available from http://www.minimalinux.org/ttylinux/showpage.php?pid=1.

[21] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[22] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205. ACM Press, June 1994.

[23] GNU Team. GNU binutils. Available from http://www.gnu.org/software/binutils/.

[24] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 7–12. IEEE, December 2005.

[25] VMware. VMware ESX server. Information available from http://www.vmware.com/products/server/esx_features.html.

[26] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX Association, June 2002.

[27] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, pages 195–210. USENIX Association, December 2002.