

Specification-Driven Dynamic Binary Translation

Jens Tröger

December 2004

A dissertation submitted in partial fulfillment
of the requirements for the degree

DOCTOR OF PHILOSOPHY

in Computer Science



Programming Languages and Systems Research Group
Faculty of Information Technology
Queensland University of Technology
Brisbane, Australia

Copyright © Jens Tröger, MMIV. All rights reserved.

jens.troeger@light-speed.de

<http://savage.light-speed.de/>

The author hereby grants permission to the Queensland University of Technology to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

“But I’ve come to see that as nothing can be made that isn’t flawed, the challenge is twofold: first, not to berate oneself for what is, after all, inevitable; and second, to see in our failed perfection a different thing; a truer thing, perhaps, because it contains both our ambition and the spoiling of that ambition; the exhaustion of order, and the discovery – in the midst of despair – that the beast dogging the heels of beauty has a beauty all of its own.”

Clive Barker
(Galilee)

Keywords

Machine emulation; formal specification of machine instructions; dynamic optimization; dynamic binary translation; specification-driven dynamic binary translation.

Abstract

Machine emulation allows for the simulation of a real or virtual machine, the source machine, on various host computers. A machine emulator interprets programs that are compiled for the emulated machine, but normally at a much reduced speed. Therefore, in order to increase the execution speed of such interpreted programs, a machine emulator may apply different dynamic optimization techniques.

In our research we focus on emulators for real machines, i.e. existing computer architectures, and in particular on dynamic binary translation as the optimization technique. With dynamic binary translation, the machine instructions of the interpreted source program are translated into machine instructions for the host machine during the interpretation of the program. Both, the machine emulator and its dynamic binary translator are source and host machine specific, respectively, and are therefore traditionally hand-written.

In this thesis we introduce the Walkabout/Yirr-Ma framework. Walkabout, initially developed by Sun Microsystems, allows among other things for the generation of instrumented machine emulators from a certain type of machine specification files. We extended Walkabout with our generic dynamic optimization framework “Yirr-Ma” which defines an interface for the implementation of various dynamic optimizers: by instrumenting a Walkabout emulator’s instruction interpretation functions, Yirr-Ma observes and intercepts the interpretation of a source machine program, and applies dynamic optimizations to selected traces of interpreted instructions on demand. One instance of Yirr-Ma’s interface for dynamic optimizers implements our *specification-driven dynamic binary translator*, the major contribution of this thesis.

At first we establish two things: a formal framework that describes the process of machine emulation by abstracting from real machines, and different classes of applicable dynamic optimizations. We define dynamic optimizations by a set of functions over the abstracted machine, and dynamic binary translation as one particular optimization function. Using this formalism, we then derive the upper bound for quality of dynamically translated machine instructions. Yirr-Ma’s dynamic binary translator implements the optimization functions of our formal framework by modules which are either generated from, or parameterized by, machine specification files. They thus allow for the adaptation of the dynamic binary translator to different source and host machines without hand-writing machine dependent code.

Our initial experimental results illustrate our techniques for different source and host machines, but provide as yet sparse actual performance measurements. However, we successfully demonstrate the portability of the Yirr-Ma framework and the feasibility of our proposed approach to specification-driven dynamic binary translation.

Contents

List of Figures	xiii
List of Tables	xv
List of Listings	xvii
Abbreviations	xix
Introduction	1
1 Machine Emulation	7
1.1 Introduction	7
1.2 Virtual Machines vs. Emulation of Real Machines	8
1.3 Related Work	9
1.4 Implementation Issues with Machine Emulators	14
1.5 Formal Principles of Machine Emulation	15
1.6 Specification Languages for Machine Emulators	18
1.6.1 Specifying the Syntax of Machine Instructions	18
1.6.2 State Definition and Operational Semantic Specification	19
1.6.3 An Introduction to the SSL Language	20
1.6.4 Using Machine Specification Files	22
1.7 The Walkabout Framework	22
1.7.1 Source Machine Dependent Implementation	23
1.7.2 Machine Independent Core Functionality	25
1.7.3 Instrumentation of Walkabout Emulators	26
1.8 Summary	28
2 Dynamic Optimizations for Machine Emulators	29
2.1 Introduction	29
2.2 Formalization of Dynamic Optimization Techniques	29
2.3 Related Work	31
2.4 Dynamic Binary Translation	36
2.5 Existing Dynamic Binary Translation Frameworks	38

2.6	Summary and Outlook	44
3	The Yirr-Ma Framework	45
3.1	Introduction	45
3.2	An Extension to Walkabout: Yirr-Ma	45
3.3	The Collector and Dynamic Semantic Information	47
3.4	The Hot-Spot Detector	48
3.5	The Emitter Interface and Dynamic Functions	50
3.6	The Code Cache	52
3.7	The Dispatcher	53
3.8	Implementation	54
3.9	Summary and Contribution	59
4	Dynamic Code Generation using Partial Inlining	63
4.1	Introduction	63
4.2	The Semantic Information	64
4.3	Instrumenting a Walkabout Emulator for Yirr-Ma's Inliner	66
4.4	Generation of Dynamic Functions	68
4.5	Implementation	70
4.6	Results	72
4.7	Summary, Contribution and Outlook	75
5	Specification-Driven Dynamic Binary Translation	77
5.1	Introduction	77
5.2	The Upper Bound for the Quality of Translated Machine Instructions	78
5.2.1	An Experiment	79
5.2.2	Examination of the Generated Code	82
5.2.3	On the Quality of Dynamically Translated Machine Instructions	84
5.2.4	Improving the Quality	86
5.3	Yirr-Ma's Dynamic Binary Translator	86
5.4	The Semantic Information	88
5.5	Instrumenting a Walkabout Emulator for Yirr-Ma's DBT	88
5.6	Incorporation of Specification Files	90
5.6.1	The Source Machine Dictionary	90
5.6.2	The Host Machine Dictionary	93
5.7	The Intermediate Representation	94
5.8	State Mapping	98
5.9	Dynamic Optimization of the Mapped IR	102
5.9.1	Pseudo Constants	102

5.9.2	Constant Propagation	103
5.9.3	Constant Folding	103
5.9.4	Dead Effect Elimination	105
5.9.5	Idiom Replacement	105
5.9.6	More Dynamic Optimizations	106
5.10	Code Generation	107
5.10.1	Instruction Selection Using Tiling	109
5.10.2	An Example	111
5.10.3	Additional Remarks	112
5.10.4	Translating System Calls	115
5.10.5	Instruction Emission	116
5.11	Generation of the Instruction Selector from a SSL Specification	117
5.11.1	Extended Tiles	119
5.11.2	Rewriting of Extended Tiles	120
5.11.3	Construction of the Search Tree	122
5.11.4	Generation of the Instruction Selector	123
5.12	Implementation	125
5.12.1	The Dynamic Machine Dictionaries	125
5.12.2	Intermediate Representation and Dynamic Optimizers	128
5.13	Experimental Results	131
5.14	Summary and Contribution	133
6	Experiences and Future Work	135
6.1	Experiences With Walkabout and Yirr-Ma	135
6.2	Future Work	137
7	Conclusions	143
A	Additional Listings	145
A.1	Yirr-Ma's FBD Code Cache on Pentium and PowerPC Host Machines	145
A.2	Walkabout/Yirr-Ma and the Pentium Frontend	146
A.3	Walkabout/Yirr-Ma and the ARM Frontend	150
A.4	Walkabout/Yirr-Ma and the PowerPC Backend	156
	Bibliography	159

List of Figures

I-1	Comparison of JIT compilation and dynamic binary translation	2
I-2	Architecture of the entire Walkabout/Yirr-Ma-DBT framework	4
1.1	Relationship between machine emulator and the host machine	9
1.2	The Dixie toolset	13
1.3	Equivalence of machine emulation and an algebraic Homomorphism	17
1.4	Architecture of the Walkabout framework	23
1.5	Internal components of a Walkabout machine emulator	24
2.1	General architecture of a dynamic optimization framework	31
2.2	The Dynamo dynamic optimizer	32
2.3	UQDBT resembles the static binary translator UQBT	39
2.4	The Crusoe VLIW processor	40
2.5	The Dynamite dynamic binary translator	41
3.1	Architecture of the Walkabout/Yirr-Ma framework	46
3.2	Internal structure of Yirr-Ma	47
3.3	Class diagram of the Hot-Spot Detector interface	49
3.4	Class diagram of Yirr-Ma's Emitter interface	51
3.5	Class diagram of Yirr-Ma's dynamic code caches	52
3.6	Sequence diagram of a successful generation of a dynamic function	54
3.7	Class diagram of Yirr-Ma's Collector	55
4.1	Class diagram of Yirr-Ma's dynamic partial Inliner	68
5.1	Architecture of Yirr-Ma and its statically compiled code cache	79
5.2	Class diagram of Yirr-Ma's DBT	86
5.3	Architecture of the specification-driven dynamic binary translator	87
5.4	Bottom-up constant folding of a SSL expression tree	104
5.5	Tiles representing PowerPC instructions, and an IR effect	109
5.6	Rewrite tree of extended tiles deriving actual tiles	121
5.7	Sequence diagram of a successful trace translation	125
5.8	Class diagram of Yirr-Ma's dynamic machine dictionaries	126
5.9	Class diagram of Yirr-Ma's dynamic binary translator	128

5.10 Class diagram of the state mapper interface	129
5.11 Class diagram of Yirr-Ma's different IR optimizers	130

List of Tables

4.1	Performance results for Yirr-Ma's Inliner on different host machines . . .	73
4.2	Performance results for Yirr-Ma with different code caches	73
4.3	Actual code cache sizes and trace lengths for Yirr-Ma's Inliner	74
5.1	Sizes and performance of Yirr-Ma's statically compiled code cache . . .	81
5.2	Performance results for some translated toy benchmark programs	132
5.3	Sizes and instruction counts of generated dynamic functions	132

List of Listings

1.1	Excerpt of SPARC's operational semantic specification using SSL	21
1.2	Walkabout's memory access macros	25
1.3	Instrumentation of interpreted SPARC trap instructions	27
3.1	Two hot example traces of an emulated SPARC program	50
3.2	Dynamic function epilogue for PowerPC host machines, not linked	58
3.3	Invoking the Yirr-Ma Dispatcher from a Walkabout emulator	60
3.4	Invocation of a dynamic function in Yirr-Ma	60
4.1	Instrumentation of interpretation functions for Yirr-Ma's Inliner	65
4.2	Extended instrumentation for SPARC's CTI instructions	67
4.3	Inlined SPARC instruction implementations for a Pentium host	70
4.4	Implementation of the <code>Inliner::emit()</code> method	71
5.1	SPARC host instructions for a statically compiled SPARC trace	83
5.2	Instrumentation of interpretation functions for Yirr-Ma's DBT	89
5.3	Static operational semantics of SPARC's <code>ADDCCimm</code> instruction	91
5.4	Dynamic operational semantics of SPARC's <code>ADDCCimm</code> instruction	92
5.5	State specification of the PowerPC	94
5.6	SPARC example trace for a translation to PowerPC	95
5.7	Static operational semantics of the example trace	96
5.8	Dynamic operational semantics of the example trace	97
5.9	IR state mapped to the PowerPC host machine	99
5.10	IR state mapped to the Pentium host machine	100
5.11	Dynamic optimization of the mapped IR	106
5.12	Further improvements of the optimized IR	107
5.13	Implementation of the SSL operator <code>?:</code> on PowerPC host machines	114
5.14	PowerPC instructions that implement a SPARC system call	116
5.15	Generated PowerPC instructions for the example trace	118
5.16	Generated PowerPC instruction selector	124

Abbreviations

- CISC *Complex Instruction Set Computer*: A processor architecture where individual instructions may perform complex operations such as effective address computations, memory access and arithmetic calculations.
- CLR *Common Language Runtime*: Microsoft’s virtual machine implementation for the .NET platform.
- CTI *Control Transfer Instruction*: A machine instruction that modifies the value of the program counter register, thus changing the control flow of a program.
- DBT *Dynamic Binary Translation*, also *Dynamic Binary Translator*: Process of, or framework for, translating the machine instructions of a program for one machine into semantically equivalent machine instructions for another machine at program interpretation time.
- ILP *Instruction Level Parallelism*: Degree of data independence between consecutive instructions.
- IR *Intermediate Representation*: An instantiation of data structures that represent information carried between two or more computation steps.
- ISA *Instruction Set Architecture*: Set of instructions interpreted by a machine over the implemented machine state.
- JIT *Just-In-Time*: Or synonymous “on the fly”; performing a task on, or to, a program during its execution.
- JVM *Java Virtual Machine*: Sun Microsystems’ virtual machine implementation to execute Java byte-code programs.
- RISC *Reduced Instruction Set Computer*: In contrast to CISC, a processor architecture where individual instructions perform only simple operations.

- STL *Standard Template Library*: A collection of C++ template classes and generic algorithm implementations.
- UQBT *University of Queensland Binary Translator*: An experimental static binary translation framework developed at The University of Queensland, Australia. It is now being maintained as the [Boomerang Decompiler](#) .
- VLIW *Very Long Instruction Word*: a class of ISAs where several in parallel executable operations are encoded into, and executed as, one single “very long” instruction.

Statement of Original Authorship

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Jens Tröger
December 2004

Acknowledgments

I consider my thesis a professional as well as a personal work. Moving overseas posed a significant number of challenges that I am thankful to have experienced and grown because of. I thus take the opportunity here to express my gratitude.

I thank my supervisor Prof. John Gough for his guidance, humor and invaluable advice in many different situations throughout the past years. He allowed me the freedom to explore my own ideas and helped me understand many frustrating details one faces when battling at runtime. My thanks also go to Mike Van Emmerik from the University of Queensland for his help and patience in explaining the UQBT code base; to Dr. Cristina Cifuentes for inviting me to undertake an internship at Sun Research in the autumn of 2002; and to Dr. Youfeng Wu for offering me an internship at Intel Research Labs during the spring and summer of 2004. Furthermore, my best wishes and thanks go to all the postgraduate students and staff at the PLAS research group I had the pleasure to work with, in particular the “PLAS Plebs” and my cubicle comrades: Aaron, Jiro and Simon, and to the other side of our Wall: Dominic, Richard, Greg and Krusty. I am also very grateful to have experienced the fun, excitement and challenge of teaching, and I would like to thank Dr. John Hynd and Mike Roggenkamp for letting me loose on a generation of poor and unsuspecting students.

Personally, I am thankful to have met many lovely people here in the land of Oz, only some of whom I have the space to recognize: Simon Kent and his beautiful wife Elizabeth who have grown into dear friends of mine; Stephan Breutel with whom I had much fun running, climbing and camping; Dan Lane for talks and smokes over wine; our indoor soccer team the “Mexican Staring Frogs”; Paul Gleeson for long and intriguing conversations in his second-hand bookstore around the corner from my place, and for reading my thesis; Quentin Cregan for constantly nitpicking on my English; and last but not least Jens Langner for just being my best friend. I am specially indebted to Laura Gardner, my gorgeous friend and challenger from Surfers: thank you for showing me my patterns.

At long last I thank my Mum, my sister Jana and my family back home in Auerbach for their love, their support and encouraging belief in me. This one is for you.

Introduction

The sheer diversity of modern processor architectures has given rise to several problems. The migration of legacy software applications to a new or different processor architecture can be difficult, in particular if the source code files are missing, incomplete, or obsolete, or if there is as yet no compiler that supports this architecture. The compilation of programs across different processor architectures complicates their distribution and increases the risk for the host architecture if the program contains malicious code, particularly given the dangers posed by the Internet. In addition, the diversity of operating systems that run on one or different processor architectures poses further challenges for the distribution of programs in a non-homogenous computing environment. In the following, we refer to a particular processor architecture/operating system pair simply as *machine*.

The above problems can be overcome by utilizing *machine emulators*. A machine emulator executes on a *host machine* and simulates a particular processor architecture to a certain extent, optionally including an operating system interface. We call this emulated machine the *source machine*. A machine emulator interprets source machine instructions in the simulated environment of the source machine. The principles of machine emulation are also used to implement *virtual machines* (or “abstract machines”, as opposed to real machines). Virtual machines commonly implement high-level programming languages such as Modula, Oberon, Perl or Python, and different programming language paradigms and thus programming languages such as Prolog, Haskell or Smalltalk. More recently, Sun Microsystems’ Java and Microsoft’s CLR implementations of virtual machines became popular for managing and securing the execution of untrusted programs in heterogeneous network environments. Furthermore, machine emulation is an inexpensive way to experiment with new processor architectures, to test compiler backends whose target architectures are not yet available, and to illustrate the functioning of existing computer architectures for educational purposes.

The interpretation of a source machine program by a machine emulator, compared to the program’s execution speed in its native environment, is slower. That is because the emulator itself is a program which interprets instructions over a simulated machine, thus introducing complex execution overhead. Virtual machines are, because of their growing importance, the focus of current research aiming at improving the performance of machine emulators through dynamic, i.e. runtime, optimizations. Such optimizations

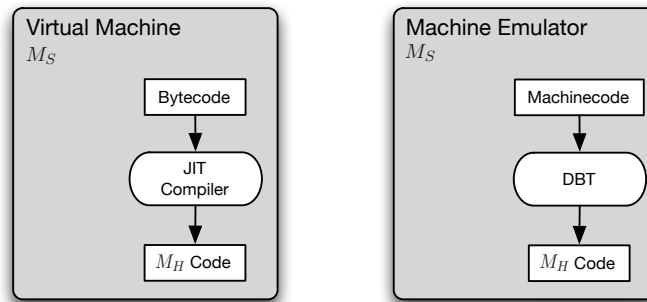


Figure I-1:

Just-In-Time compilation of byte-code for a virtual machine is similar to dynamic binary translation of machine code for a machine emulator. The differences lie within the input languages and applicable optimization techniques, whereas the actual applied compilation techniques resemble one another.

are applied on demand to selected and frequently interpreted sequences of source machine instructions, called *traces*, during the interpretation of the source program. These dynamic optimizations can be applied to either or a combination of

- the interpreted source machine instructions
- the emulated source machine
- the implementation of both on the host machine.

The execution of optimized traces in preference to their original interpretation usually results in an improved performance of the machine emulator, and therewith increases the execution speed of the interpreted source program.

Because the application of one optimization technique may give rise to other ones, the different optimization techniques are often combined. One particular combination of dynamic optimizations is *dynamic binary translation*. Similar to the Just-In-Time (JIT) compilation of the byte-code for virtual machines, dynamic binary translation optimizes the interpretation of a trace of source machine instructions by translating source machine instructions in the context of an optimized mapping of the source machine state to the host machine (cf. figure I-1).

Machine emulators are tailored to simulate one particular source machine, and the applied dynamic optimization techniques are specific for both the source and host machines. Because of that, optimizing machine emulator frameworks are usually hand-written, and building such a framework for a new source machine and/or porting it to another host machine is a tedious and error prone task. In contrast, machine specifications describe relevant properties of a particular machine and abstract from implementation details.

Machine specifications are often used to generate or parameterize machine specific components of application programs like a static compiler or machine emulator. However, little has been done to build optimizing machine emulators that utilize dynamic binary translation from specification files.

In this thesis we propose a dynamic binary translation framework for a certain class of machine emulators which is completely built from machine specification files. In particular, the contributions of this thesis are as follows:

- definition of a formal framework that describes the process of machine emulation and, using this framework, derivation of a set of functions that define different dynamic optimizations, in particular dynamic binary translation, for machine emulation
- design and implementation of Yirr-Ma¹, a generic and machine independent dynamic optimization framework that uses generated Walkabout machine emulators as a vehicle
- a case study of a dynamic binary translator for Yirr-Ma that utilizes dynamic partial inlining of instruction interpretation functions in order to generate an optimized version of selected source machine traces
- investigation and definition of the upper bound of the quality for dynamically translated machine instructions
- analysis of the properties of machine specification files and, as our major contribution, the design and implementation of a dynamic binary translator for Yirr-Ma that is completely built from machine specification files.

We introduce the Yirr-Ma framework, an extension to generated Walkabout emulators for user-level programs. Walkabout was developed by Sun Microsystems, and it allows for the generation of simple machine emulators from machine specification files, their functional extension through instrumentation and therewith it allows for the implementation of machine code debuggers or dynamic optimizers. Figure I-2 illustrates the schematic architecture of Walkabout together with our generic dynamic optimization framework Yirr-Ma. A Walkabout emulator is generated from SLED and SSL machine specification files, and its instruction interpretation functions can optionally be instrumented at the users will. Using such an instrumented Walkabout emulator as a vehicle, Yirr-Ma observes the interpretation of a program at instruction level, collects information about hot traces on demand, optimizes hot traces and invokes the generated host machine code if possible. Through well defined interfaces, Yirr-Ma also allows us to

¹ Yirr-Ma (Wagiman): “to gather”, “to collect”. Wagiman is an Australian Aboriginal language spoken in Australia’s state of the Northern Territory.

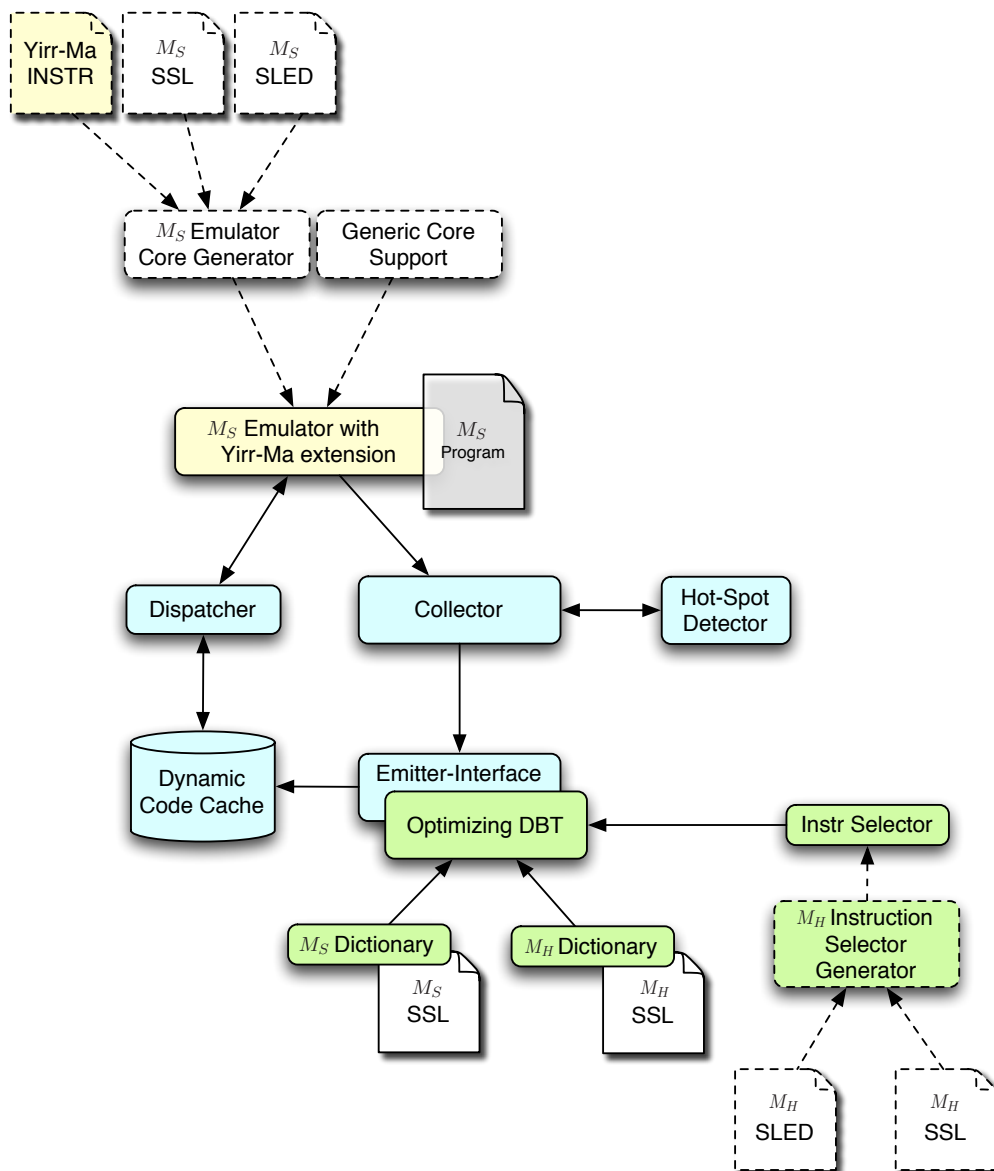


Figure I-2:

Overall schematic architecture of the Walkabout/Yirr-Ma framework utilizing our optimizing dynamic binary translator. The dashed components are used statically to generate machine dependent implementations of some of Yirr-Ma's components, and solid components represent our dynamic optimizer framework Yirr-Ma. White components represent parts of the Walkabout and UQBT framework, yellow components contain code that is shared between Walkabout and our Yirr-Ma dynamic optimizer. The light green components comprise our generic dynamic optimizer, and the dark green components comprise our specification-driven dynamic binary translator. We elaborate on the individual components, their interface specification, interaction and implementation at due places throughout this thesis.

experiment with different hot-spot detection techniques, code cache implementations and dynamic optimization techniques for frequently interpreted source program traces. Because the major focus of this thesis is on specification-driven dynamic binary translation, we analyze the properties of machine specification files and introduce algorithms and methods to build and employ a dynamic binary translator. Yirr-Ma's experimental dynamic binary translator, shown in dark green in the figure, is built from annotated and extended Walkabout specification files, a fact which illustrates the feasibility of our techniques. The quality of the instruction translation has an upper bound that is *not* defined by the applied translation and optimization techniques. We show how that upper bound is determined, and how it can be improved. Finally, where relevant, we give performance results and analyze the outcomes of our methods. For instance, one implementation for Yirr-Ma's dynamic optimizer interface is a partial Inliner which improves the performance of interpreted benchmark programs. In contrast, our specification-driven dynamic binary translator as yet fails larger applications, but successfully translates toy benchmark programs, however at high runtime cost.

We have implementations of Walkabout/Yirr-Ma running on SPARC, Pentium and PowerPC host machines. In addition to the stable SPARC frontend, we also generated experimental and incomplete ARM and Pentium frontends, a SPARC, Pentium and PowerPC backend for our dynamic partial Inliner, and a PowerPC and GNU Lightning backend for our specification-driven dynamic binary translator.

After having emphasized the contributions of this thesis, we stress what this thesis does not address. Walkabout is an experimental framework, and so is Yirr-Ma: neither of the two frameworks claims industrial strength. The focus of this thesis is not the investigation of sophisticated hot-spot detection methods and their management, nor the code caching techniques of generated host machine instructions. In our work we do not focus on the fine-tuning of generated machine code for one particular host machine, but on the development of generally applicable dynamic compilation and optimization techniques.

This thesis is structured as follows. In chapter 1 we review and formalize the concepts of machine emulation and talk about different implementation techniques for machine emulators. We also introduce the Walkabout emulator framework. Some dynamic optimization techniques are put into, and examined within, their formal context in chapter 2, and we motivate dynamic binary translation as a combination of two particular dynamic optimization techniques. In chapter 3 we introduce our generic dynamic optimization framework Yirr-Ma, and we show how Yirr-Ma extends a Walkabout emulator using a custom instrumentation for Walkabout's interpretation functions. We introduce the concept and implementation of our dynamic partial Inliner in chapter 4, and we present performance results. Chapter 5 is dedicated to the major contributions of this thesis.

It elaborates on the quality of dynamically translated machine instructions, introduces our specification-driven approach to dynamic binary translation, and gives initial experimental results. Finally, we talk about our experiences and future work in chapter 6, and summarize this thesis in chapter 7.

Analysis of Virtual Method Invocation for Static Binary Translation

During the first fifteen months of our research we focused on *static binary translation* using the retargetable UQBT framework [CE99]. We then narrowed our focus of research to consider only dynamic binary translation, the results of which are presented in this doctoral thesis. Our work on static binary translation was a collaboration project with Dr. Cristina Cifuentes at Sun Microsystems. A technical report [Trö01] and a conference paper [TC02] about our research and its results were published, summarizing our work in this area. However, we do not revisit this work in the course of this thesis, and we refer our reader to the published literature.

1 Machine Emulation

Why can we throw a question farther than we can pull in an answer? Why such a vast net if there's so little fish to catch?

(Yann Martel)

1.1 Introduction

Over the course of the past few decades, a number of different processor architectures have been created. This, in conjunction with the large number of operating systems, lead to a diversity of software applications designed for these different operating systems and processor architectures. Although operating systems and processor architectures implement different techniques and approaches to resource management and computation, today's computers share common fundamental principles. The theoretical foundations of formal computation and computability, and of the representation and encoding of algorithms were almost simultaneously proposed in the 1930s by Alonzo Church with his functional λ -Calculus [Bar84], and by Alan Turing et al. with his Universal Turing Machine [Tur36]. Almost ten years later, Turing's more "hardware oriented" proposition inspired actual computer architecture implementations as described among others by John von Neuman et.al. in [BGvN46], whereas Church's λ -Calculus finds application in virtual machine implementations for functional programming languages. Today, decades later, computers and programming paradigms reflect and implement these fundamental principles, some of which motivate the formal foundation of our work.

An "electronic computing instrument", as von Neuman referred to computers, consists of a processor, memory and an input/output device of some sort. Instructions and data for the machine¹ are encoded into a binary representation and stored in the computer's memory, thus defining the computer's current state. The processor fetches an instruction from memory, decodes it, reads the required operand data and then interprets the instruction, thus advancing the state of the machine.

As formalized with the Universal Turing Machine, the state (i.e. instructions and data) of a machine itself can be encoded and stored in the memory of another machine. This other machine then interprets the original instructions over the encoded state and

¹ We use the terms "computer" and "machine" interchangeably to express the close relation to our formal framework later in this chapter.

thus equally advances the state of the original machine. It is said that one machine *emulates* another. In the following, we refer to the emulated machine as the *source* machine, denoted with M_S , and to the emulating machine as the *host* machine, denoted with M_H .

In this chapter we introduce and formalize the concepts of machine emulation, and we discuss machine specification languages that are relevant for our work. Furthermore, we review related work, and we introduce Sun Microsystems' Walkabout framework which serves as research and implementation vehicle for our work.

1.2 Virtual Machines vs. Emulation of Real Machines

Virtual machines *abstract* from features and properties of actual processor architectures, and provide a homogeneous and usually strongly typed execution environment for programs across different computer platforms. They often provide and implement high-level programming language features such as abstract data types, higher-order functions or symbolic evaluation, and they support the execution of programs with advanced memory management (e.g. garbage collection) or managed code facilities.

There exists a substantial number of virtual machines that were developed in pursuit of differing outcomes. Amongst the most recent and popular of them are Sun Microsystems' Java Virtual Machine (JVM) [LD97] and Microsoft's Common Language Runtime (CLR) [MR03]. Both virtual machines secure and manage the execution of platform independent programs and are available for numerous different host machines. Other virtual machines that support such high-level programming languages, to name only a few examples, include Warren's Abstract Machine implementing the Prolog language [AK91]; the Lisp [McC85] and Haskell [Dav92] interpreters implementing functional languages; the SELF [US87] and Oberon interpreters implementing object-oriented languages.

In contrast to virtual machines, an emulator for real machines implements the features and properties of an actual processor architecture. It interprets the processor's instruction set architecture (ISA) over an implementation of the source machine state. Depending on its purpose and precision in modeling the emulated machine, an emulator implements architectural features like register files, memory access, precise hardware exceptions, caches and pipelines. Figure 1.1 illustrates the relationship between a machine emulator and its host computer. However, in order to execute user-level programs on an emulated machine, only an approximation of a real machine is required to be implemented by an emulator. We restrict the focus of our work to this class of machine emulators only.

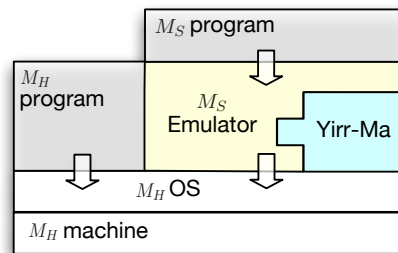


Figure 1.1:

Relationship between a machine emulator for source machine programs and its host machine. Instructions of the source program are interpreted over the simulated source machine state by the source machine emulator, and system service calls are mapped to the host operating system. The source machine emulator executes on top of, and utilizes, the host machine operating system. Arrows illustrate the invocation of system services.

1.3 Related Work

Sun's JVM and Microsoft's CLR

The most popular virtual machines used for research, in commerce and teaching are Sun Microsystems' Java Virtual Machine and Microsoft's Common Language Runtime. Gough gives an introductory comparison of the two virtual machines in [Gou01].

The Java Virtual Machine. The Java programming language [Mel99, GJSB00] is an object oriented language. Once written, Java programs are compiled into a bytecode representation that is then interpreted by the JVM [LY97]. The instruction set of the JVM is tailored to support only Java programs and its object model which complicates the implementation of other programming languages, particularly not object oriented languages. The JVM itself implements an evaluation stack and a set of bytecode instructions that operate on that stack. The stack contains a fixed number of slots for each compiled method, and every slot holds either a 32-bit scalar value or an object reference. 64-bit wide scalars like long integers or double floats occupy two slots. During program execution, a Java program is represented by a set of dynamically loaded class files, each of which defines the behavior of the class and specifies externally accessible member fields and methods. All references to fields and methods are symbolic and are stored in an indexed constant pool.

The symbolic information used by the JVM allows for type-checking and bytecode verification when a class is loaded and, to a smaller degree, at runtime. These features are essential for the implementation of an execution environment in which untrusted code is securely executed without posing a threat to the host machine. Another advantage of interpreted bytecode is the introduction of some form of memory management and

garbage collection, thus eliminating the need for explicit deallocation of memory.

All JVM instructions are strongly typed and directly support the Java object model. The operand type for a particular instruction is encoded into the instruction's mnemonic which saves the interpreter the type inference overhead at runtime. Classes in Java inherit from only one superclass and may implement several interfaces, i.e. purely abstract classes. The JVM also defines instructions that allocate and initialize objects of abstract data types, and instructions that account for the four different types of method invocation in Java: static methods, virtual methods, interface methods and virtual methods invoked from a static context.

The Common Language Runtime. Akin to Sun Microsystems' JVM, Microsoft's CLR [MR03] is a virtual machine that consists of an evaluation stack and a set of bytecode instructions that operate on that stack. In contrast to the JVM, however, the ISA of the CLR was not designed to support only one particular language, and thus provides broader semantics for instructions, especially for the support of object-oriented languages.

The CLR supports scalar data of primitive data types, value classes (i.e. structures and arrays) and reference classes; it provides single inheritance and multiple interfaces for reference classes. Methods are divided into static methods and instance methods that are either virtual or non virtual, and the CLR provides invocation mechanisms accordingly. Parameters are passed to methods by pushing them onto the evaluation stack, supporting both call by reference and call by value.

Unlike the JVM, the CLR does not have an interpretative option. The unit of deployment is an "assembly": a set of classes and/or namespaces, or parts thereof. At load time of an assembly, the contained code is verified, compiled into host machine instructions, and then invoked by the CLR.

Shade

Shade [CK94], a collaborative work between Sun Microsystems and the University of Washington, is a "tracing tool" to collect different types of runtime information about the execution of a program. The program can be instrumented on the fly, allowing the user to inject custom code into the original program without modifying it statically. Shade provides an instruction set simulator for both SPARC and MIPS source machines, and is targeted to run on a SPARC host machine. The simulation of the source machine is precise: signals and exceptions are implemented as well as dynamic linking and other system level services.

The Shade simulator consists of a translation cache and a translation look-aside buffer (TLB), implemented by a hash table. For every source machine instruction that is about to be interpreted, Shade uses the TLB to find an implementation for that instruction

in the cache. If the lookup succeeds, the implementation is invoked, thus advancing the emulated machine state. If the lookup fails, Shade immediately translates a “chunk” of instructions for the host machine, starting with the current machine instruction. A chunk, similar to a basic block, is a sequence of instructions: it starts with the current instruction and ends either with a CTI or a “tricky” instruction. In this context, the authors regard trap instructions or memory synchronizations as tricky instructions. Translation of a chunk of instructions means to wrap the source machine instructions with a prologue and an epilogue that map the emulated registers for the interpreted instruction, to add instrumentation and to adjust the operands of the instructions to the emulated environment.

Results on Shade show a general slowdown of the source programs by a factor of 3 to 6 for not instrumented programs, and up to a factor of 80 for some instrumentations. In addition to the introduction of Shade, the paper also gives an extensive comparison of machine emulators and their features.

Embra

Embra [WR96], developed at MIT and Stanford University, is an emulator for the precise emulation of MIPS R3000 and R4000 processors. It models architectural features such as MMU, exceptions and interrupts, privileged instructions, caches and much more. Embra is part of the SimOS framework which allows for the study of operating systems and computer architectures in a simulated environment [RHWG95].

The architecture of Embra is similar to that of Shade. Rather than interpreting source machine instructions, Embra generates sequences of host machine instructions (called “fragments”) from source program basic blocks using *virtualization*², and stores these fragments in a cache. The dispatcher then invokes the generated fragments, thus advancing the emulated source machine state. If a requested fragment was not found in the cache, the dispatcher initiates its generation. Depending on the host machine, Embra emulates a multi-processor source machine with shared memory facilities in different ways: when executing on a single processor host machine, it implements and switches between every simulated processor manually, or, on a multi-processor machine, it implements every simulated source machine processor by a host machine thread.

Results on different benchmark programs report a general slowdown when compared to the native execution of the programs. Additionally, the paper studies results of differing cache simulations extensively.

FX!32

Digital (now Hewlett Packard) developed the FX!32 [CH97] emulator in order to support the transition from Windows NT programs compiled for x86 to Alpha hosts. In fact,

² Virtualization emulates a source machine on the same host machine. In that case, instructions of the emulated machine are not interpreted, but directly executed by the host machine.

FX!32 combines a x86 machine emulator with a static binary translator. No system level instructions or programs (such as drivers) are supported.

Both the x86 emulator and the static binary translator are controlled by FX!32's server. If a source program is executed for the first time, the x86 emulator captures a profile of this first run. A profile contains information that supports the static binary translator: addresses of interpreted `call` instructions, source/target address pairs of indirect jumps, and addresses of instructions that perform unaligned memory accesses. Upon program termination, the FX!32 server invokes the static binary translator which then uses the collected profile information to translate the x86 instructions into Alpha instructions. The so generated host machine profile is stored in a database, and is then invoked for subsequent executions of the same source program. Using this approach, a set of original x86/Windows NT programs is incrementally translated to the Alpha host machine. The x86 emulator itself interprets only a subset of x86 instructions and implements the x86 registers in a private data structure (called "context") for every simulated thread. Upon the invocation of a translated profile, emulated registers are mapped from the context to actual Alpha registers, and written back to the context when the execution continues in the x86 emulator.

The given results compare programs executing on a 200 MHz Pentium Pro with FX!32 on a 500 MHz Alpha host machine. With pure emulation, interpreted x86 programs execute roughly as fast as in their native environment, whereas the translated programs outperform pure emulation by a factor of 2.

Dixie

The Dixie toolset [FE98, FE99] was designed at the Universitat Politècnica de Catalunya in Barcelona. It allows a user to instrument and monitor user-level programs that execute on the Dixie virtual machine (DVM).

The Dixie toolset consists of several components, as shown in figure 1.2. At first, the Dixie compiler, a static binary translator, translates the source program into instructions for the Dixie virtual machine. Using Jango, the compiled Dixie program can be instrumented to the user's requirements. The instrumentation specification defines the instrumentation of the source program on its instruction level, however, Jango inserts the instrumentation into the compiled Dixie program. Speedy is an experimental optimizer for Dixie programs that performs host machine independent optimizations, and optionally compiles Dixie instructions into actual host machine instructions. Finally, a compiled, instrumented and optimized Dixie program is executed by the Dixie virtual machine.

The DVM implements a register-based and RISC like 64-bit vector machine that loads and initializes the translated Dixie program. It provides full architecture coverage of all source machines, including precise exceptions. Dixie's different compiler frontends

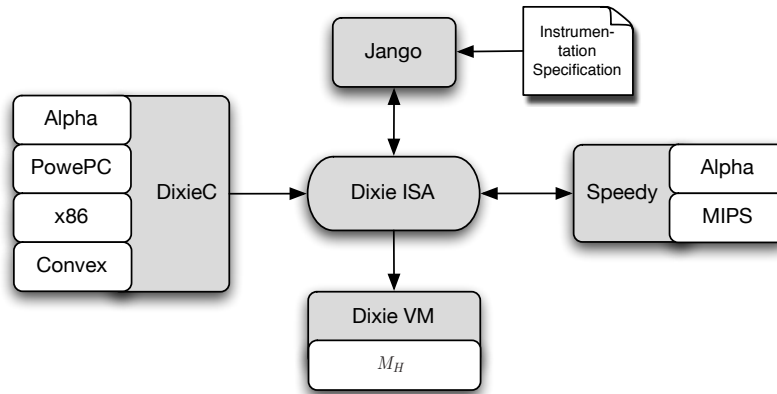


Figure 1.2:

The Dixie toolset consists of several components. At first, the Dixie compiler translates a program into Dixie instructions. The Dixie program can then be instrumented (Jango) and optimized (Speedy), and is then interpreted by the Dixie virtual machine.

and Speedy’s backends implement hand-written interfaces in combination with code generated from custom specification files.

Experimental results show a slowdown of the (unoptimized) interpreted Dixie program by 80% to 160% compared to its native execution. Further analysis of the runtime behavior showed that the overhead added by the DVM depends on the source machine ISA and on the implementation of the DVM on the host machine.

Strata

The Strata framework [SD01, SKV⁺03] emulates arbitrary source machines on the same host machine. Using virtualization, it generates sequences of host machine instructions (again called “fragments”) that implement a basic block of the source program for the emulated machine. Overall, Strata’s architecture resembles that of Shade, Embra and any common dynamic optimization framework.

Through its modularized implementation, Strata can easily be ported to different host architectures and operating systems. Porting Strata requires the user to manually re-implement a set of interface functions that encapsulate actual machine dependent functionality. Those functions implement the decoding of source machine instructions, the generation and emission of fragments, and reading or modifying the emulated source machine state. Furthermore, generated fragments can be instrumented, allowing Strata to observe and intercept the execution of source programs at any time. The authors implemented both a cache simulator and a system call monitor that restricts the execution of untrusted source programs.

The results show a slowdown by a factor of 1 to 3 for a not instrumented source program, depending on the runtime behavior of the program. The papers also demonstrate the instrumentation of arbitrary source programs using Strata.

Bochs

Bochs [Law04] is an open-source emulator that simulates 386, 486, Pentium, Pentium Pro or AMD64 processor architectures, optionally including MMX, SSE, SSE2 and 3DNow instructions. It also simulates all standard peripheral PC devices such as keyboard, networking cards, VGA cards and monitors etc. Bochs, therefore, provides an environment that can run a complete operating system for x86 machines. It can be compiled for, and executed on, Windows, Linux and BSD host operating systems running on x86, PowerPC, Alpha, SPARC and MIPS. Interaction with the host operating system is encapsulated and implemented by modules that forward events of the emulated hardware to the host machine.

The implementation language for Bochs is C/C++ which reduces migration of Bochs to a new host machine to the mere recompilation of the framework. In contrast to other x86 emulators (e.g. the commercial VMWare or Plex86 [NL03]) which employ virtualization to increase their execution speed, Bochs actually interprets every x86 machine instruction. This, in turn, results in a comparatively slow emulation speed, but allows Bochs to run on almost any host machine.

1.4 Implementation Issues with Machine Emulators

The ISA of real machines, compared to that of modern virtual machines, is simple and supports only primitive data types. Modern processor architectures implement their state by a set of registers, called the *register file*, and random access memory. In contrast, the instructions of a virtual machine have complex semantics and operate on a machine state that explicitly supports abstract data types, thus modeling a completely different and more sophisticated computing environment. The instructions and data of a program for either machine type are stored together in the machine's memory, and the instructions of the program are then interpreted over the data. This interpretation of machine instructions is either implemented by a processor architecture in hardware, or, in case of a machine emulator, in software.

To which extent an emulator implements the machine and interprets machine instructions depends on the requirements for the execution of the source program. For example, “well behaved” user-mode programs rarely use privileged instructions that directly affect the hardware. An emulator for such programs thus does not require to implement these instructions nor related hardware features of the emulated machine. On the other hand, an emulator for system software, such as an operating system or a device driver, must implement the architecture of the emulated machine in much more detail. The general approach to the implementation of a machine emulator is to model the emulated machine state with data structures, and to write functions that interpret the machine's

instructions over that modeled machine state.

Therefore, the performance of a machine emulator depends on

- the complexity and precision of the emulated machine state and its implementation on the host machine
- the interpretation functions
- the differences between the emulated machine and the host machine.

For instance, registers of the emulated machine can be stored entirely in the host machine's memory, or they are either partially or completely mapped to host registers. Memory segments of the emulated source machine are mapped into the memory of the host machine. This mapping may require relocation of the segments to a different host address, if the required memory location is already occupied. Relocation of emulated memory segments introduces an additional indirection for every interpreted memory accesses because the original source address must be redirected to the relocated host address. Furthermore, the representation of values in memory (either big or little endian) causes overhead when values of the emulated machine have to be adjusted such that they can be interpreted correctly by the host machine. Equally critical for the emulator's performance is the quality of the instruction interpretation, because interpretation functions depend on the implementation of the emulated machine state and thus add overhead.

If carried out by hand, the implementation of a machine emulator is a tedious task subject to human error. Porting an emulator to a new host machine may prove to be equally difficult. As an alternative to their manual re-implementation, machine emulators can be generated from specification files which describe the source and host machines sufficiently for the emulator's purpose. We examine the formal aspects of machine emulation and relevant machine specification languages in the following.

1.5 Formal Principles of Machine Emulation

This section formalizes the concept of machine emulation on a high abstraction level, thus deriving the formal foundations for our work and defining equations to which we refer in the following chapters.

Let $M = (S, I, \gamma)$ be a machine, where

- S denotes the set of states of the machine
- I denotes a set of machine instructions
- $\gamma : I \times S \rightarrow S$ is the interpretation function for machine instructions over a machine state.

Actual processor architectures that implement the properties of a von Neuman architecture, encode program data and instructions into a bit vector that is stored in (or represented by) the memory of the machine. Therefore, the relation $I \subset S$ holds; however, for simplicity, we consider both distinct from each other³.

Let $s_n \in S$ be the current state of the machine M , then

$$\gamma(i_n, s_n) = s_{n+1}$$

denotes the interpretation for an instruction $i_n \in I$ over the state s_n that computes the successor state $s_{n+1} \in S$ of s_n . Consequently, the iterative application of γ

$$\gamma(i_{n+m}, \dots, \gamma(i_{n+1}, \gamma(i_n, s_n)) \dots) = s_{n+m}$$

denotes the interpretation of a sequence, or *trace*, t of instructions, with

$$t = \langle i_n, i_{n+1}, \dots, i_{n+m} \rangle \in I^*.$$

We abbreviate the interpretation of a trace t with the interpretation function for traces

$$\gamma^* : I^* \times S \rightarrow S,$$

such that

$$\gamma^*(t, s_n) = s_{n+m}$$

holds. Let $s_0, s_f \in S$. If s_0 denotes the initial state for a program, then

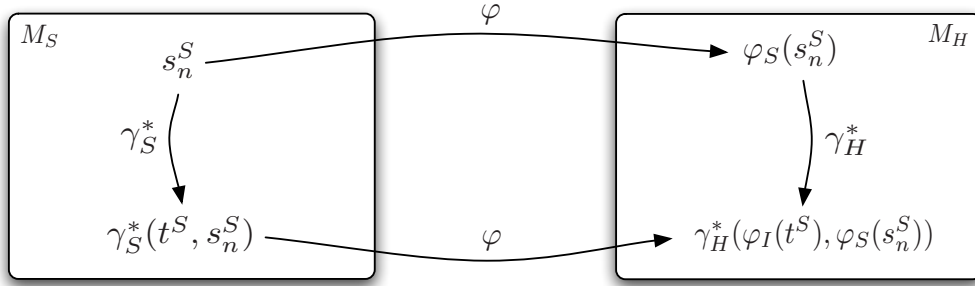
$$\gamma^*(t, s_0) = s_f$$

denotes the run of that program and s_f is the program's final state.

Furthermore, let $M_S = (S_S, I_S, \gamma_S)$ be the emulated or source machine, and let $M_H = (S_H, I_H, \gamma_H)$ be the host machine. Because the data and instructions of a program are encoded in a single bit-vector, this bit-vector can be mapped into the memory of the host machine, thus defining a particular host machine state $s_m^H \in S_H$. Given the mapping of a source machine state $s_n^S \in S_S$ to a host machine state, a host machine can compute the successor state $s_{n+1}^S \in S_S$ by interpreting a source machine instruction over the mapped state s_n^S ⁴. The interpretation of source machine instructions is implemented by traces

³ In fact, given $I \subset S$, instructions can always be inferred dynamically from the state. (Not so statically, where the problem is equivalent to the Halting problem.) We abstract from this implementation detail and regard S and I as distinct sets.

⁴ The theoretical foundations were already described by Alan Turing [Tur36]: the program for an Universal Turing Machine UTM_A (the host machine machine) can be the encoding of another Universal Turing Machine UTM_B (the source machine) that is computed by UTM_A .


Figure 1.3:

The emulation of a source machine M_S by a host machine M_H is similar to an algebraic Homomorphism. The interpretation γ_S^* of a trace t^S of source machine instructions over a source machine state s_n^S is equivalent to the interpretation γ_H^* of the trace of φ_I -mapped source machine instructions over the φ_S -mapped source machine state. Both interpretations compute an equivalent successor state of s_n^S .

of host machine instructions, and the interpretation of the host machine instructions (e.g. by the host processor) then computes a new host machine state which in turn represents the mapped source machine state s_{n+1}^S .

Formally, machine emulation is similar to an algebraic *homomorphism*⁵. Let $\varphi_S : S_S \rightarrow S_H$ be an injective state mapping that maps a given source machine state to an equivalent host machine state, and let $\varphi_I : I_S^* \rightarrow I_H^*$ be an injective trace mapping that maps a trace of source machine instructions to a trace of host machine instructions. Then

$$\varphi_S(\gamma_S^*(t^S, s_n^S)) = \gamma_H^*(\varphi_I(t^S), \varphi_S(s_n^S)) \quad (1.1)$$

with $t^S \in I_S^*$ denotes the emulation of a source machine M_S by a host machine M_H . More precisely, equation 1.1 denotes the interpretation of a trace t^S of source machine instructions over the mapped source machine state s_n^S by the host machine, computing an equivalent successor state. Figure 1.3 illustrates the relationship between the interpretation of a sequence of source machine instructions t^S over a source machine state s^S through M_S , and the interpretation of the φ_I -mapped trace over the φ_S -mapped source machine state through M_H . The figure also illustrates the commutativity of γ and φ ; we will use this property in section 2.2 to illustrate different dynamic optimizations.

Informally, a machine emulator *implements* the source machine state in one way or another (i.e. provides an implementation for φ_S) and *interprets* the source machine instructions over the mapped state by means of host machine instructions (i.e. provides

⁵ The different properties of a relationship between two algebraic structures of the same type are expressed by various morphisms. Let A and B be two arbitrary sets, and let $+$: $A \times A \rightarrow A$ and $*$: $B \times B \rightarrow B$ be two binary operators over A and B , respectively. Now let $(A, +)$ and $(B, *)$ be two algebraic structures. A homomorphism φ between these two structures is defined as the linear mapping $\varphi(a_1 + a_2) = \varphi(a_1) * \varphi(a_2)$ with $a_1, a_2 \in A$. Note that a machine M can be regarded as an algebraic structure $M = (S \cup I, \gamma)$. In fact, the von Neuman architecture implements exactly this union of machine state and instructions.

an implementation for γ_S). The host machine instructions in turn are interpreted by the host machine through the interpretation function γ_H . Based on this formal definition of machine state and instruction interpretation, we introduce the machine specification languages used for our work in the following.

1.6 Specification Languages for Machine Emulators

Machine specification languages are domain specific languages designed for the description of a processor architecture and its related features to a certain extent and with specific detail. Verilog and VHDL, for example, are low level hardware oriented description languages that describe the processor architecture as well as its implementation on a functional or behavioral level. Such specification languages are used to design and simulate hardware implementations of processor architectures, including features like buses, pipelines and caches. However, machine specifications that suit the generation of machine emulators are required to describe instruction decoding, the operational semantics of instructions and, depending on the quality and purpose of the emulator, they also have to describe the machine in certain detail. For example, [ÖG98] describes a framework that allows for the specification and generation of precise machine emulators, including cache behavior and pipelines: the machine specifications describe the architecture of the emulated machine, its ISA and calling conventions sufficiently.

A different approach take declarative specification languages as discussed in [RDF01]. Declarative specification languages focus on the behavioral execution model of a machine, and they abstract from implementation details of the actual underlying processor architecture. They describe the syntax, i.e. the binary encoding or the bit patterns, of machine instructions and their operational semantics. In general, the operational semantics defines the state of the specified machine, and describes how instructions operate (i.e. modify) over that state. Both the instruction decoder and interpreter of a machine emulator can be generated from such declarative machine specifications, thus supporting the portability of machine emulators and overcoming many of the issues of hand written emulators. However, few complete frameworks that allow for the generation of optimizing machine emulators from such specifications exist. In the following sections, we introduce the specification languages that we use throughout this thesis and, where appropriate, point at related work.

1.6.1 Specifying the Syntax of Machine Instructions

The Specification Language for Encoding and Decoding (SLED) [RF94, RF97] is unique and allows for the specification of the syntax of machine instructions. It was developed for the Zephyr compiler project which accounts for the introduction of declarative ma-

chine specifications for different types of tools that deal with assembly and machine code. Unfortunately, neither the Zephyr project nor its related set of specification languages were ever completed.

SLED specifications are used to generate machine instruction encoders and decoders. With SLED, the syntax of instructions is specified by means of fields and tokens that describe opcodes, their operand registers and immediate values. Patterns are values for these tokens, and SLED defines a number of operations over such patterns allowing to define relations between tokens, fields and other patterns. Constructors then define the binary encoding of machine instructions by defining the combination of patterns, fields and tokens into instructions, or parts thereof, thus mapping between the binary representation of machine instructions and their mnemonics in assembly language.

The Walkabout framework uses existing SLED specifications to generate an instruction decoder for the emulator, and we use SLED specifications to generate the instruction encoder for Yirr-Ma's dynamic binary translator.

1.6.2 State Definition and Operational Semantic Specification

The operational semantics of an ISA is specified by a state definition and for every instruction a sequence of functions over the state which define the semantics of the instruction. The functions are often referred to as *register transfers* or *effects*. Note that the specification of a Turing complete machine⁶ requires a specification language of the same expressive power.

The definition of a machine state specifies architectural features like the register file, register types and properties, as well as attributes of the memory (e.g. endianness of data representation) that are relevant for the purpose of the language. The state definition may also describe caches or pipelines, if these details are required by the application domain of the specification language. The operational semantics of machine instructions is defined by a list of effects (or functions) for every instruction, and operates over the defined state. Each of the effects describes an atomic change of the machine state, the sum of which is the effect of the specified instruction itself.

Examples of operational semantic specifications are: the Semantic Specification Language (SSL) [CS98], λ -RTL [RD99] or ADL [ÖG98]. λ -RTL, like SLED, is part of the Zephyr project. It is tied to SLED and, similar to [Col90], uses the λ -calculus to describe the functional effects of an instruction. A λ -RTL specification is difficult to write due to the potential complexity of the semantics of modern machine instructions. SSL, in contrast, follows a simpler approach.

⁶ Existing computers are only Turing complete modulo resource limitations. Recall that the classical Turing Machine has a tape of unlimited length which implies a computer with unlimited memory resources.

1.6.3 An Introduction to the SSL Language

SSL was originally designed for the UQBT static binary translation framework [CEe99], and modified versions are used by the Boomerang decompiler [Ee02] and by Walkabout. With UQBT, SSL was used as an abstraction language to bind decoded machine instructions to their semantic representation for further analysis in the context of the static decompilation process.

A SSL specification file consists of two logical parts: a state definition and the specification of the instructions' operational semantics. Listing 1.1 shows an excerpt of the SSL specification for SPARC. The state definition describes memory attributes (for SPARC big-endian byte order) and it defines the register file. A SPARC processor provides 32 integer registers, each of which is 32-bit wide. Special purpose registers are marked using an aliasing construct. The operational semantics of an instruction is specified by a list of weakly typed effects over the defined state – a “microprogram” that implements the instruction. SSL provides a set of operators that are sufficient to express the operational semantics of most instructions.

In listing 1.1, for example, the LD instruction has three register operands: `rs1`, `rs2` and `rd`. The first two are the source operand registers, and the last one is the target register. The actual type of the operand register must be inferred from the context when the operational is instantiated for a particular machine instruction. LD loads a 32-bit wide value from a memory address (the sum of the values in registers `rs1` and `rs2`) into the target register `rd`. Four effects in total define the operational semantics of the LD instruction. Line 32 shows the *primary effect* that specifies the load of the 32-bit value into the target register. The effects in lines 31, 33 and 34 define *secondary effects* that further modify the machine state.

The distinction between primary and secondary effects is irrelevant to an emulator because it has to compute the entire change of the emulated machine state. However, only the primary effect is required to select an instruction for a given semantic expression. We elaborate on the differences of primary and secondary effects in more detail in section 5.10.1.

SSL has some drawbacks that, especially for the purpose of this thesis, eventually lead to problems for our dynamic binary translator. None of the SSL operators nor their operands are explicitly typed. Although most types of effects and their sub-expressions can be inferred through a three-pass traversal over the effect and with consideration of the state definition, this step proves to be an inconvenient intermediate step at runtime. Furthermore, SSL does not provide loop constructs, i.e. the explicit repetition of effects. For example, the operational semantics of some instructions (e.g. Pentium's or PowerPC's string processing instructions) requires the repetition of effects while a condition holds. In SSL this is expressed using the pseudo registers `%SKIP` and `%RPT`:

```

1  # representation of values
2  ENDIANNESS BIG;
3
4  # integer registers; normal registers have a mapping
5  # to a cardinal, special register maps to -1
6  INTEGER
7
8  # general purpose registers
9  [ %g0, %g1, %g2, %g3, %g4, %g5, %g6, %g7,
10   %o0, %o1, %o2, %o3, %o4, %o5, %o6, %o7,
11   %l0, %l1, %l2, %l3, %l4, %l5, %l6, %l7,
12   %i0, %i1, %i2, %i3, %i4, %i5, %i6, %i7 ][32] -> 0..31,
13
14   %sp[32] -> 14, # Aliases
15   %fp[32] -> 30;
16
17   # control registers
18   [ %pc, %npc, %Y, %CWP, %TBR, %WIM, %PSR, %FSR ][32] -> -1,
19   # standard flags
20   [ %CF, %ZF, %NF, %OF, %AF ][1] -> -1,
21   # floating point flags
22   [ %FZF, %FLF, %FGF ][1] -> -1;
23
24   ...
25
26   NOP
27       *32* %pc := %npc
28       *32* %npc := %npc + 4;
29
30   LDreg rs1, rs2, rd
31       *32* %g0 := 0
32       *32* r[rd] := m[r[rs1] + r[rs2]][32]
33       *32* %pc := %npc
34       *32* %npc := %npc + 4;
35
36   ...

```

Listing 1.1:

Excerpt of SPARC's SSL specification. The specification file consists of a state definition that describes relevant features of the register file and memory properties. The operational semantics of SPARC instructions are then defined over this state definition.

```
REPMOVSVD
*1*  %SKIP := %ecx = 0  # pseudo register %SKIP
*32* m[%edi] := m[%esi]
*32* %esi := %esi + ((%DF = 0) ? 4 : -4)
*32* %edi := %edi + ((%DF = 0) ? 4 : -4)
*32* %ecx := %ecx - 1
*1*  %RPT := 1          # pseudo register %RPT
```

Repetition is inferred from the context: if `%SKIP` is assigned true, all effects defining the instruction are skipped, and if `%RPT` is assigned true the effects are repeated. The writer of such a specification has to ensure that adequate pseudo repetition effects describe the instruction correctly, and the employed tools which process the specification files are required to handle this information correctly. Also, the specification of the conditional execution of a sequence of effects is not supported by SSL. This complicates writing specification files for a predicated ISA such as ARM, and has a negative impact on the performance of the generated emulator components.

1.6.4 Using Machine Specification Files

Declarative specification files can be used to *generate* a specific implementation of an algorithm from, or to *parameterize* an algorithm's generic implementation with. Walkabout emulators, the basis for our work on dynamic binary translation, are generated from machine specifications, thus following the former approach. However, we will use a hybrid approach to build our dynamic binary translator. That is: one, we generate instrumented Walkabout machine emulators and an instruction selector from SLED and annotated SSL specifications; and two: we parameterize a generic state mapper using an extended SSL state specification. The intermediate representation used by our dynamic binary translator is generic and machine independent, and it represents the dynamic operational semantics of instructions derived from a SSL specification. Note that we do not require any additional abstract intermediate representation and hence do not add further abstraction overhead to our dynamic optimizer.

1.7 The Walkabout Framework

The Walkabout framework [CLU02] allows among other things for the generation of behavioral machine emulators from machine specification files⁷, as illustrated in figure 1.4. A generated Walkabout emulator interprets user-level programs without supporting low-level processor features like exceptions, caches or pipelines. The framework was developed by Sun Microsystems, and it grew out of experiences gathered with the UQBT

⁷ In spite of the paper's title, Walkabout, in combination with Pathfinder, does *not* implement a dynamic binary translator as per our definition, but a virtualizer for SPARC. However, because it also implements instrumented machine emulators, we use it as a vehicle for our work on actual dynamic binary translation.

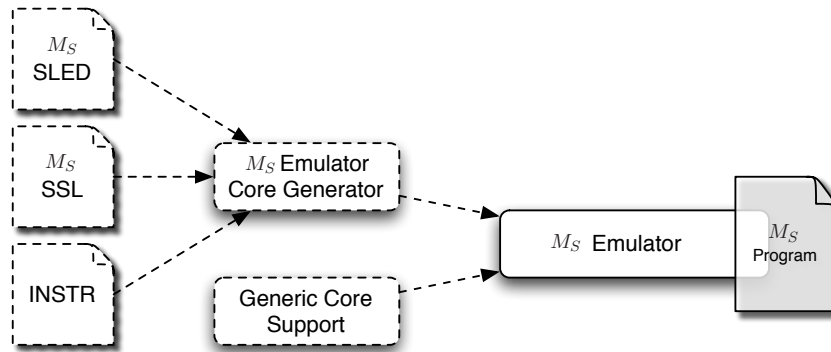


Figure 1.4:

The Walkabout framework allows for the generation of optionally instrumented behavioral machine emulators from machine specifications. The syntax and operational semantics of the source machine instructions are specified using the SLED and SSL specification languages, respectively. The emulator generator generates an instruction decoder and interpreter that is linked with an emulator support library, producing a source machine emulator for user-level programs.

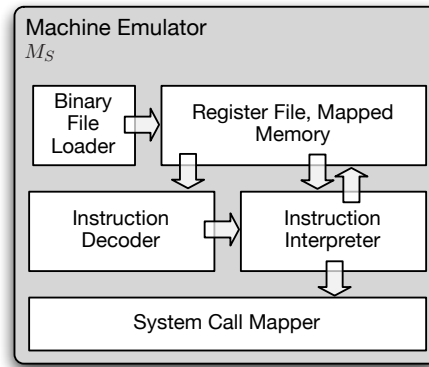
and UQDBT binary translators. However, its current experimental implementation supports only the generation of SPARC emulators. We extended Walkabout such that generated Pentium emulators run on selected host machines.

SLED and SSL specification files describe the syntax, machine state and operational semantics of the interpreted ISA. Using these specifications, Walkabout’s emulator generator generates the C/C++ implementation of the instruction decoder and interpreter for the machine emulator. The generated components are compiled and then linked with the generic part of the emulator, producing the actual source machine emulator. We generated emulators for Solaris/SPARC running on different host machines: Solaris or Linux on SPARC, PowerPC and Pentium. We also experimented with Pentium and ARM emulators on Pentium hosts machines.

The work we present in this thesis extends Walkabout emulators. Such a generated emulator serves as the vehicle for our generic dynamic optimizer Yirr-Ma. Note that the properties and limitations of Walkabout and its generated machine emulators directly influence the structure and methods of our work and dynamic binary translation. We discuss these issues further in section 5.2.

1.7.1 Source Machine Dependent Implementation

The source machine dependent components generated by Walkabout comprise the instruction decoder, an implementation of the source machine state, and functions that interpret source machine instructions over the state implementation. The instruction decoder is generated from a SLED specification using the New Jersey Machine Code

**Figure 1.5:**

A Walkabout machine emulator consists of machine dependent components like the instruction decoder, interpreter and the machine state implementation, and the machine independent binary file loader and system call mapper. All machine dependent components are generated from machine specification files.

Toolkit (NJMC Toolkit) [RDF01]. The state implementation and the instruction interpretation functions are generated from a SSL specification. For every specified source machine instruction, Walkabout generates a single function that implements the interpretation of this instruction, for example:

```

void executeLDREG(sint8_t op_rs1, sint8_t op_rs2, sint8_t op_rd)
{
    regs.rd[0] = 0;
    regs.rd[op_rd] =
        getMemsint32_t(regs.rd[op_rs1] + regs.rd[op_rs2]);
    regs.r_pc = regs.r_npc;
    regs.r_npc = (regs.r_npc + 4);
}

```

The resemblance of the generated function `executeLDREG` that interprets SPARC's LD instruction and the instruction's SSL specification in listing 1.1 is obvious. Because LD has three operands, the interpretation function takes three parameters that represent these operands. The actual parameter values passed to the interpretation function are offsets into the emulated SPARC register file `regs`, a global variable which implements the emulated SPARC register file by means of a compound data structure. This data structure is stored in the host machine's memory, and individual emulated registers are accessed by memory load and store instructions of the host machine.

The Walkabout emulator maps and relocates the emulated memory segment directly into the address space of its own process on the host machine. The global variable `mem` holds the low address of the emulated memory segment, and a set of macros implementing the memory access (cf. line 4 in above example). These memory access macros are generic and, if source and host machines differ in their data representation, they also

```

1  #include <byteswap.h>
2  ...
3  extern char *mem;
4  ...
5  #ifndef _SWAPBYTES
6      #define getMemsint32_t(a)    (*((sint32_t *)&mem[a]))
7  #else
8      #define getMemsint32_t(a)    bswap_32 (*((sint32_t *)&mem[a]))
9  #endif

```

Listing 1.2:

Generic Walkabout memory access macros that load, under consideration of the endianness of the host machine, a signed 32-bit integer value from the emulated memory segment `mem`.

implement an endianness conversion.

Listing 1.2 shows the macros that implement the load of a 32-bit signed integer value from the emulated memory segment. The macro in line 6 relocates the memory access and is used if the emulator runs on a host machine with the same endianness as the source machine. In contrast, the macro in line 8 extends the former one by swapping the bytes of the loaded value for the emulated source machine, if source and host machine implement different endianness (e.g. SPARC on a Pentium host).

The generated machine dependent components of a Walkabout emulator are compiled and then linked with the generic core support functions.

1.7.2 Machine Independent Core Functionality

The generic part of a Walkabout emulator is comprised of the binary file loader and a system call mapper.

The binary file loader loads and relocates all program sections (code, data and such) of the source program into the memory segment dedicated for emulation, and it initializes the runtime environment for the emulated process using information from the binary file. The only currently supported binary file format is ELF which proved to be sufficient to interpret Solaris and Linux programs. The binary file loader reads the `.aux` auxiliary vector of the binary file and, together with the environment settings of the emulator process, initializes a valid process environment for the loaded source program. The `.interp` section of the source binary file contains the path to the “interpreter” for the loaded program which is usually the source machine’s dynamic linker `ld`, or it is empty for statically linked programs. Next, the Walkabout emulator loads and relocates the dynamic linker binary and then starts interpretation at the linker’s entry point. This, in turn, loads the required shared libraries into the emulated memory segment, dynamically links them with the original source program and eventually transfers control to the `main()` function of the source program.

Programs compiled for SysV types of operating systems (e.g. Linux or Solaris) invoke operating system services by executing a dedicated interrupt instruction. The emulator implements this instruction with a generic `doTrap()` function that implements a system call mapper. A new implementation of the system call mapper is required for every new source machine. For example, if the emulated program was compiled for Solaris and if the host machine runs either Solaris or Linux, system calls are almost identical and can easily be mapped. The system call mapper copies the parameters for the invoked system service from the emulated source machine registers to the designated host machine registers, and then invokes the equivalent service of the host operating system. The results of the system call are then copied back to the emulated register file or parameter structure, preserving a consistent emulated state across mapped system calls. Note that copying parameters, in this context, also requires to adjust structures passed to, and returned from, the system call. Adjustment across machines then means to change the data representation and alignment of all items using a deep traversal over the structure!

Both machine dependent and generic components of a Walkabout emulator are linked together, thus producing a machine emulator that executes on arbitrary host machines. Due to slight differences of a process' address space segmentation on Linux running on different processor architectures, we had to tweak the generic core implementation of Walkabout emulators for different host machines. [CLU02] gives some performance results that show a slowdown of programs interpreted by a Walkabout emulator by a factor of 200 when compared to their native execution.

Throughout this thesis we primarily focus on a generated SPARC emulator, but we also point at results and experiences with other source and host machines, where appropriate.

1.7.3 Instrumentation of Walkabout Emulators

A generated Walkabout emulator can be optionally instrumented at the instruction level. Using the instrumentation language INSTR [CLU02], arbitrary C/C++ code is added before and/or after the actual interpretation of a source machine instruction. As indicated in figure 1.4, the instrumentation is defined in a separate file. The emulator generator reads and processes an instrumentation specification and, while it generates the interpretation functions, inserts the instrumentation code into the interpretation functions.

Listing 1.3 shows an instrumentation specification for the SPARC `trap` instructions. The generated SPARC emulator intercepts calls to the system service `open()` (cf. also the Strata instrumentation example in [SKV+03]). As can be seen in the listing, an instrumentation specification consists of three logical parts: a definition of instructions

```
1  DEFINITION
2
3  traps [ "TA", "TN", "TNE", "TE", "TG", "TLE", "TGE", "TL",
4          "TGU", "TLEU", "TCC", "TCS", "TPOS", "TNEG", "TVC",
5          "TVS"
6          ]
7
8  traps eaddr
9  {
10     if (PARAM(eaddr) == 8 && SSL(%g1) == 5)
11     {
12         char* filename = SSL(%o0);
13         // intercept open() call
14     }
15     // actual instruction semantics follows
16     SSL_INST_SEMANTICS
17 }
18
19 FETCHEXECUTE
20 {
21     SSL_INST_SEMANTICS
22 }
23
24 INSTRUMENTATION_ROUTINES
25
26 // no additional user code
```

Listing 1.3:

Instrumentation of SPARC's trap interpretation functions. The inserted code checks whether the invoked trap is a system call (`PARAM(eaddr) == 8`) and the system call number is 5 (a call to `open()`). The system call can then be intercepted at the user's will.

and their respective instrumentation, a definition of the emulator’s fetch-and-execute cycle, and additional user code. The instructions that are to be instrumented are listed in a table, here labeled `traps`, and the instrumentation code is inserted as specified. The emulator generator replaces the `INSTR` keyword `SSL_INST_SEMANTICS` with the actual instruction semantics, and it expands the `PARAM` and `SSL` keywords with their respective implementation when it generates the interpretation functions. The following listing shows the generated interpretation function for SPARC’s TA (unconditional trap: “trap always”) instruction:

```
void inline executeTA( sint32_t op_eaddr )
{
    if (op_eaddr == 8 && regs.r_g1 == 5)
    {
        char* filename = regs.r_o0;
        // intercept open() call
    }
    // actual instruction semantics follows
    doFlagTRAP((1 == 1), (op_eaddr + 128)) ;
}
```

All other interpretation functions of the listed trap instructions are instrumented in a similar fashion.

In order to link a generated Walkabout emulator with our generic dynamic optimizer Yirr-Ma, we use a custom instrumentation for *all* interpreted SPARC instruction. This instrumentation allows Yirr-Ma to observe, intercept and dispatch the interpretation of a source program, such that Yirr-Ma’s optimizers are invoked for selected interpreted traces. We introduce this custom instrumentation at due places.

1.8 Summary

We introduced the concepts of machine emulation in this chapter. We abstracted and formalized relevant aspects of machine emulation, thus showing the equivalence of machine emulation with an algebraic homomorphism. The properties of a homomorphism, like the commutativity of the mapping function φ and the interpretation function γ , help us formalizing dynamic optimizations and, in particular, dynamic binary translation in the next chapter. We also introduced the set of Walkabout emulators that are generated from declarative machine specifications written in SLED and SSL, and we talked about some of Walkabout’s implementation details. Instrumented Walkabout emulators serve as a vehicle for our dynamic optimizer Yirr-Ma throughout the remainder of this thesis.

2 Dynamic Optimizations for Machine Emulators

Nicht dort ist die Tiefe der Welt und ihrer Geheimnisse, wo die Wolken und ihre Schärze sind, die Tiefe liegt im Klaren und Heiteren.

(Hermann Hesse)

2.1 Introduction

In the previous chapter we revisited and formalized the concepts of machine emulation, and we introduced the set of generated Walkabout emulators. In principle, a machine emulator implements the state of the emulated source machine and interprets source machine instructions over this state. Because the interpretation of a program is, compared to its native execution, usually slower, the dynamic optimization of frequently interpreted source machine traces should, at least notionally, increase the overall performance of an interpreted program.

In this chapter we focus on different dynamic optimization methods for machine emulators, and we illustrate and formalize those methods by extending the formalisms presented in the previous chapter. In particular, we derive and motivate dynamic binary translation as a special dynamic optimization technique for machine emulators. Major parts of this chapter are also dedicated to the review of previous and related work in the area of dynamic optimization.

2.2 Formalization of Dynamic Optimization Techniques

We showed in equation 1.1 on page 17 that a machine emulator implements a source machine state s_n^S on a given host machine through the state mapping φ_S , and that source machine instructions are interpreted by host machine instructions through the instruction mapping φ_I . In turn, the host machine instructions are then interpreted over the host machine state, thus computing the new state $\varphi_S(s_{n+1}^S)$. In order to improve the performance of an interpreted source program, an emulator may apply optimizations to the state and/or instructions of both source and host machine at runtime. Such dynamic optimizations are aimed at

- improving the emulated state through the state optimization function $\omega_{S_S} : S_S \rightarrow S_S$, or improving the implementation of the emulated state through the state optimization function $\omega_{S_H} : S_H \rightarrow S_H$
- generating an optimized version of a source machine trace for subsequent interpretation through the trace optimization function $\omega_{I_S^*} : I_S^* \rightarrow I_S^*$
- generating an optimized version of a host machine trace that improves the subsequent interpretation of source machine instructions through the trace optimization function $\omega_{I_H^*} : I_H^* \rightarrow I_H^*$.

The state optimization function ω_{S_S} manipulates, i.e. may rearrange or rewrite, emulated data and other selected parts of the emulated machine state, such that the modification of equation 1.1 holds:

$$\varphi_S(\gamma_S^*(t^S, s_n^S)) = \gamma_H^*(\varphi_I(t^S), \varphi_S(\omega_{S_S}(s_n^S))).$$

Similarly, ω_{S_H} manipulates the mapped data and the implementation of the emulated machine state, such that

$$\varphi_S(\gamma_S^*(t^S, s_n^S)) = \gamma_H^*(\varphi_I(t^S), \omega_{S_H}(\varphi_S(s_n^S)))$$

holds. The optimization of source machine instructions is described by the trace optimization function $\omega_{I_S^*}$. A source machine trace is rewritten into an equivalent yet more performant trace, such that

$$\varphi_S(\gamma_S^*(t^S, s_n^S)) = \gamma_H^*(\varphi_I(\omega_{I_S^*}(t^S)), \varphi_S(s_n^S))$$

holds. Similarly, the host machine instructions that result from an instruction mapping are optimized by the trace optimization function $\omega_{I_H^*}$, such that

$$\varphi_S(\gamma_S^*(t^S, s_n^S)) = \gamma_H^*(\omega_{I_H^*}(\varphi_I(t^S)), \varphi_S(s_n^S))$$

holds. Note that the state consistency is preserved in all cases above, denoted by the equality of the left-hand side of the equations with the modified right-hand sides, respectively.

The different dynamic optimizations can be applied individually or in combination with one another. However, the application of dynamic optimizations consumes time and memory resources that compete with the interpretation of the source program. Optimizations are therefore only applied to *selected* partial states and traces *on demand*. The decision about what and whether to optimize, and to what extent, is speculative and based on heuristics. As shown in e.g. [Knu71], many programs spend 90% of their

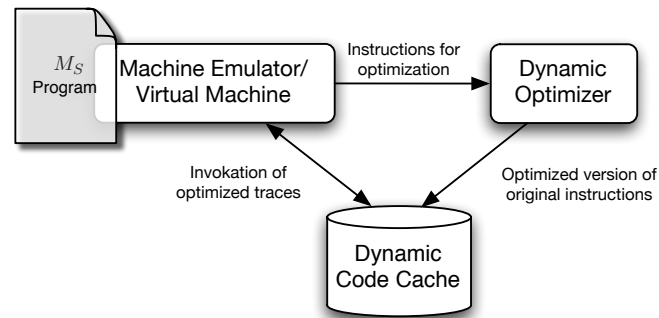


Figure 2.1:

The general architecture of a dynamic optimization framework usually consists of three major components: an interpreter that interprets, observes, and intercepts the source program, and that selects traces for the dynamic optimizer; a dynamic optimizer that generates an optimized version of a selected source machine trace; and the code cache where optimized traces are stored for their subsequent invocation by the interpreter.

runtime in 10% of their code. Therefore, it must be the goal of the dynamic optimizer to improve these 10% in order to leverage the runtime resources that were invested into dynamic optimizations.

The general architecture of a dynamic optimization framework is depicted in figure 2.1. It usually consists of three major components: a machine emulator or virtual machine, the actual dynamic optimizer, and a dynamic code cache. While the machine emulator or virtual machine interprets the source program, it observes the runtime behavior of the program and selects traces of instructions and/or portions of the emulated machine state that, speculatively, are worthwhile optimizing. A selected trace, for example, is then passed to the dynamic optimizer which generates an improved version of that trace and stores it in the dynamic code cache. Subsequent branches to the now optimized trace are intercepted by the interpreter and diverted to its improved cached version, thus resulting in an increased performance of the interpreted program.

2.3 Related Work

In the following section we review dynamic optimization frameworks for different types of machine emulators or virtual machines. Each of those frameworks applies one or more of the above optimization functions and provides different implementations for those functions. For further reading, [Smi00] gives a general yet thorough introduction to both the potential and obstacles of dynamic optimization techniques.

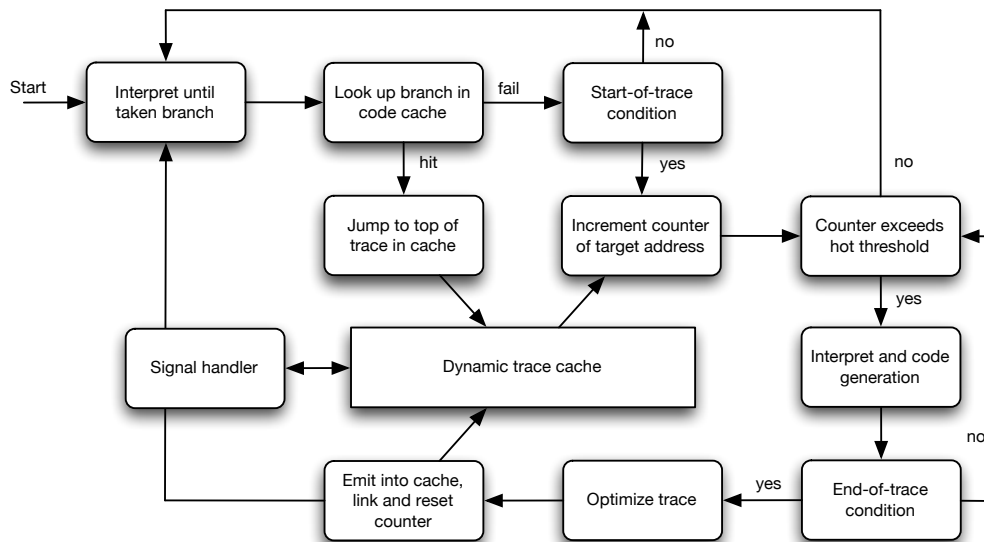


Figure 2.2:

Architecture of the original Dynamo dynamic optimizer for machine instructions. Instructions are interpreted until a hot trace is encountered. The hot trace is then collected, optimized and stored in a code cache, from where it is invoked in subsequent branches to the trace.

Dynamo

Dynamo [BDB99a, BDB99b, BDB00] is one of the most referenced frameworks for dynamic optimization of user-mode programs. The original implementation interprets PA-RISC programs and runs on PA-RISC hosts under the HPUX operating system. As a joint work with MIT, Dynamo was ported to the IA-32 architecture and now runs on both Windows and Linux operating systems [DA02].

Figure 2.2 illustrates Dynamo’s architecture. The instructions of the source program are interpreted in order to observe the runtime behavior of the source program. Upon the detection of a hot trace, i.e. a trace that was interpreted a sufficient number of times, Dynamo switches into “trace generation mode” where it also emits the interpreted instructions into a custom buffer. When an end-of-trace instruction is interpreted, the content of the buffer is optimized and then stored in the code cache. With subsequent branches to a trace that has an optimized version stored in the code cache, the interpreter branches to the optimized trace, thus switching to the native execution of dynamically optimized machine code. Dynamo thus implements the trace optimization function $\omega_{I_H}^*$.

Dynamo’s granularity for dynamic optimization is the “trace”: a trace starts with an instruction that meets the start-of-trace condition, and terminates when an end-of-trace condition is met. An instruction meets the start-of-trace condition if it is the target of a backward taken branch, or if it is the target of an exit branch of an already optimized trace in the code cache. The emission of instructions into the trace buffer continues until

one of the following end-of-trace conditions is met: another backward taken branch is interpreted, or the buffer is full.

The applied dynamic optimizations for a buffered trace are divided into conservative and aggressive techniques. Conservative optimizations do not result in modified source program memory nor register contents: elimination of redundant branches, copy propagation and constant propagation. Aggressive optimizations include redundant load and store removal, dead code elimination and loop transformation. Their application is unsafe in some situations and difficult to undo. For example, if a signal arrives during the execution of an optimized trace, the original state of the program may be difficult to restore.

Depending on the runtime behavior of the source program, Dynamo results in an average speedup of 11.4% over -O2 statically optimized native SpecINT benchmark programs. Furthermore, running a -O2 statically optimized source program under Dynamo is, on average, 4.1% faster than its -O4 optimized version running without Dynamo.

Wiggins/Redstone

Wiggins/Redstone (WR) [RDGY99] was developed at Compaq (now Hewlett Packard), and is an experimental framework for dynamic profiling, optimization and specialization of applications running on Alpha machines. WR uses hardware sampling to detect hot spots, and combines it with a software implementation for dynamic optimization.

WR works as follows. An agent and a sampler periodically collect information about the executed instructions. Driven by the system clock and Alpha's "Retired Instruction" event, the sampler collects the addresses of executed instructions, together with an associated counter. If the agent detects a hot spot, WR's trace builder begins to collect basic blocks and builds them into an instrumented trace. This goes on until heuristics or a backward branch to the beginning of the trace terminate this collection. During subsequent executions of the hot trace, the added instrumentation collects further information for the optimizer. If such an observed trace is executed frequently enough, it is transformed into a custom intermediate representation, optimized and emitted into a code cache. Like Dynamo, WR provides an implementation for $\omega_{I_H}^*$.

WR applies a series of optimizations to a trace. These include code motion and code specialization, constant propagation, common subexpression elimination and dead code elimination. The instructions of a trace are not rescheduled, nor are registers reallocated for the optimized version of the trace. The given results are comparable to those of Dynamo. However, only 30% to 40% of the overall execution time of a program was spent in optimized traces due to code cache restrictions and other limitations on the length of traces.

Mojo

The Mojo [CLCG00] dynamic optimizer was developed at Microsoft Research for the dynamic optimization of complex desktop applications. Its initial implementation targets x86 host machines running Windows 2000, and addresses issues like program exception handling (not to be confused with processor exceptions) and multi-threading.

Mojo uses the Vulcan [SEV01] instrumentation toolkit to build a tool that replaces the source program’s normal entry point with a call to a custom function. When a program is started, this function branches to Mojo’s dispatcher, thus gaining complete control over the execution of the program. The dispatcher identifies basic blocks by a lightweight disassembler. It then copies one basic block at a time into a private cache and executes it. In addition, the basic block cache collects statistical information about the contained basic blocks. Mojo uses techniques similar to those used by Dynamo to identify sequences of hot basic blocks. Once it identifies a hot spot, Mojo begins to generate a “path” incrementally.

Regardless of their original layout in the program, hot basic blocks are contiguously inserted into the path. This allows for the removal of unconditional branches and for dynamic function inlining. However, Mojo does not apply any other optimizations to the path. The optimized path is then stored in a separate path cache, and is later invoked by the dispatcher. Program exceptions, for example raised by the C++ statement `throw`, are usually implemented using a defined Windows API. Mojo thus patches this API and diverts such a call to a custom implementation. In order to handle multi-threaded programs, Mojo implements one basic block cache per thread and a single synchronized cache for all generated paths.

The given performance results show a general slowdown of interpreted programs, except for many highly repetitive ones. The introduced overhead of Mojo for some of the less repetitive programs can take up to 15% for the dispatcher, optimizer and for the cache management.

Walkabout/Pathfinder

The dynamic optimizer Pathfinder [CLU02] extends a generated Walkabout SPARC machine emulator and implements virtualization¹, combined with other dynamic optimizations on a SPARC host machine.

Pathfinder is linked with the Walkabout emulator through a custom instrumentation that allows Pathfinder to observe the interpretation of the source program. When a hot-spot is detected, the interpreted hot SPARC instructions are copied into a buffer, optimized and then stored in a code cache for subsequent invocation. A dispatcher switches between interpretation and native execution. Pathfinder does not implement

¹ Note that for virtualization $M_S = M_H$ holds. The instruction mapping φ_I is therefore an identity function, and the optimization functions $\omega_{I_S^*}$ and $\omega_{I_H^*}$ are equivalent in the case of virtualization.

advanced code caching techniques for optimized traces.

The generated Walkabout emulator interprets SPARC V8 instructions. Pathfinder rewrites some of the V8 instructions into SPARC V9 host instructions and then performs further optimizations: logically adjacent basic blocks are reordered to improve code locality, thus removing redundant branches and inverting conditional branch instructions. The generated instructions, when emitted into the code cache, are linked to previously emitted instructions.

Experimental results show a speed-up compared to the plain interpretation; however, compared to the native execution, the overhead introduced by Walkabout is hardly outperformed.

Jalapeño

Jalapeño [BCF⁺99, AFG⁺00] is a dynamic optimizing compiler for Java bytecode and was developed at the IBM T. J. Watson Research Center as a part of their JVM implementation. In contrast to other dynamic virtual machine optimizers, Jalapeño compiles every bytecode instruction into host machine instructions without interpreting them in the first place. Jalapeño consists of three different compilers that implement different optimization levels. The primary goal of Jalapeño was to deliver high performance and scalability of Java programs.

The architecture of the Jalapeño optimizing compiler resembles that of any common static compiler. Java bytecode is translated into a register based high-level intermediate representation (IR) that matches the load-store properties of the host architectures. In addition, the IR is enriched with dynamic profiling information that allows for aggressive dynamic optimizations. In subsequent steps, the high-level IR is optimized and transformed into a low-level IR, and then eventually into a host machine specific IR. Using bottom-up rewriting, host machine instructions are selected for IR instructions and emitted for execution.

Jalapeño applies different types of optimizations to the differing IRs. It also incorporates runtime information, depending on the sophistication of the optimizing compiler. Copy propagation and constant propagation are applied as well as dead code elimination, register renaming, method inlining etc. [AFG⁺00]. The given results show a roughly equivalent performance when compared to IBM's own enhanced port of Sun Microsystems' JDK-JIT compiler for small programs.

More Dynamic Optimizers for Virtual Machines

[Kis99, KF99] proposes a dynamic optimization framework for the Oberon 3 system that *continuously* re-optimizes the executing program using runtime feedback. Through the continuous observation and profiling of the source program, feedback about its changing working set is gathered and incorporated into the program's re-optimization. Individual procedures are optimized by applying constant folding and dead code elimination, code

motion, peephole optimization, register allocation and other techniques. In addition, [Kis99] gives extensive background information on other related and previous work.

An interesting technique to optimize Java bytecode programs at runtime is given in [GEK01]. Instead of interpreting Java bytecode, a static bytecode translator produces a “threaded code” representation [Bel73] which is then interpreted by a JVM compatible interpreter. Instead of decoding and switching on the opcodes of bytecode instructions, an instruction in threaded code is the address of the instruction’s interpretation function. This technique removes the overhead of decoding the bytecode instruction and switching to its interpretation function. It is also independent from the host machine and does not require a JIT compiler. A similar technique, though implemented for a Forth interpreter, is presented in [Ert01]. The paper also discusses variations on threaded code interpretation.

SELF [US87] is an interpreted, pure object-oriented programming language developed at Sun Microsystems. Its virtual machine underwent much research for static and dynamic optimizations, as well as combinations thereof [HA96]. One interesting dynamic optimization technique for SELF is described in [HCU91]: the sending of polymorphic methods is cached at the call site by a “polymorphic inline cache”, thus reducing the overhead of fully typed method dispatching. The cached type information is reused later when methods are recompiled into native code. This particular optimization technique led to a sometimes significant performance increase for a large set of SELF programs.

2.4 Dynamic Binary Translation

After discussing dynamic optimization techniques in general, we focus now on one in particular: dynamic binary translation. Dynamic binary translation is a dynamic optimization technique used by machine emulators, where selected traces of source machine instructions are re-compiled for the host machine. As is, the term “dynamic binary translation” describes what is well known from virtual machines as “just-in-time compilation”. In contrast to static binary translation, however, where both the state *and* the instructions of the source program are translated into a complete program for the target machine, a dynamic binary translator translates source machine instructions for an *implementation* of the source machine state into instructions of the target, i.e. host machine. For an overview of the problems of binary translation in general, and different related frameworks see [CM96].

A dynamic binary translator employs one of the following two methods:

- the dynamic abstraction of a source machine trace into a portable and machine independent intermediate representation which is then compiled into host machine instructions

- a fixed mapping of individual source machine instructions to host machine instructions through pattern matching, without the need for an intermediate representation.

The advantage of the first approach, compared to the second one, is that trace-global optimizations can be applied, resulting in the generation of more performant host machine code. Clearly, this comes at the expense of runtime and memory resources. On the other hand, fast code emission is achieved by the fixed mapping, whereas the quality of the generated instructions does not compare favorably to that of the first technique. The fixed mapping associates source machine instructions with sequences of host machine instructions, and is easier to implement. Regardless of the actual employed approach, dynamic binary translation implements abstract principles that we formalize in the following.

Formalization of Dynamic Binary Translation

In the context of our previously established formal framework, dynamic binary translation is an implementation of the trace optimization function $\omega_{I_H^*}$ for host machine traces in combination with the trace mapping function φ_I . The *translation function* τ for dynamic binary translation is defined as

$$\tau : I_S^* \rightarrow I_H^*,$$

such that for a given trace $t^S \in I_S^*$ of source machine instructions the following holds:

$$\tau(t^S) = \omega_{I_H^*}(\varphi_I(t^S))$$

and

$$\varphi_S(\gamma_S^*(t^S, s_n^S)) = \gamma_H^*(\tau(t^S), \varphi_S(s_n^S)). \quad (2.1)$$

Note that further improvements of the translated code can be achieved by translating an optimized source machine trace:

$$\tau(t^S) = \omega_{I_H^*}(\varphi_I(\omega_{I_S^*}(t^S))).$$

Furthermore, let

$$\tau^{\omega_{S_H}} : I_S^* \rightarrow I_H^*$$

be a *parameterized translation function* that translates a given source machine trace in the context of an ω_{S_H} -optimized state mapping, such that

$$\varphi_S(\gamma_S^*(t^S, s_n^S)) = \gamma_H^*(\tau^{\omega_{S_H}}(t^S), \omega_{S_H}(\varphi_S(s_n^S))) \quad (2.2)$$

holds.

Equation 2.2 shows that a given source machine trace t^S is translated and optimized as a black box, ensuring the consistency of the emulated state before and after the trace’s invocation. In fact, dynamic binary translation combines state and trace optimizations: given the source machine trace t^S , the state implementation of t^S is optimized by ω_{S_H} , giving rise to the application of trace optimizations $\omega_{I_H^*}$. The literature often refers to $\omega_{S_H}(\varphi_S(s_n^S))$ as a *context switch* between an emulated machine state and its mapped/optimized implementation on the host machine. When a machine emulator switches between the interpretation of the source program and the execution of a $\tau^{\omega_{S_H}}$ -translated trace thereof, the implementation of the mapped state must be synchronized with the emulated source machine state: the context s_n^S for the interpretation of t^S switches to $s_m^H = \omega_{S_H}(\varphi_S(s_n^S))$ for the execution of $t^H = \tau^{\omega_{S_H}}(t^S)$.

The quality of the translation, and therewith the quality of the dynamically generated host machine trace, is defined and limited by the state mapping φ_S and its optimization function ω_{S_H} . We show an upper bound for that code quality in section 5.2, and we discuss how this upper bound can be improved in the same section.

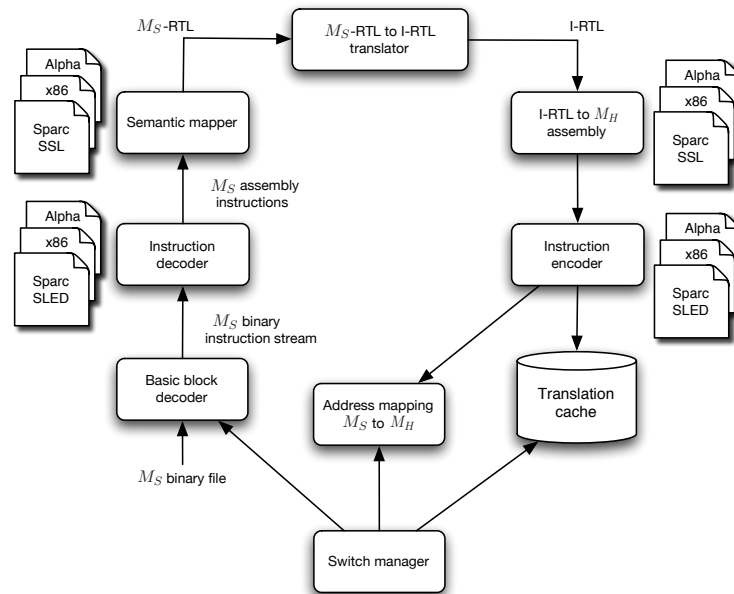
2.5 Existing Dynamic Binary Translation Frameworks

Dynamic binary translation is not a new research area, and a number of frameworks have been introduced in the past, however, most of them are written by hand. With the recent interest in JIT compilation for virtual machines, retargetable dynamic binary translation has again received more attention.

UQDBT

The University of Queensland Dynamic Binary Translator UQDBT [UC00b, UC00a] was an attempt to build a complete dynamic binary translator from specification files. Because UQDBT resembles the architecture of its static counterpart UQBT, the techniques it utilizes are fundamentally different from our Walkabout/Yirr-Ma framework. In fact, Walkabout was introduced to overcome UQDBT’s many problems.

Figure 2.3 depicts the architecture of the UQDBT dynamic binary translator. The unit of translation is a single basic block. Source machine instructions are never interpreted. Starting with the source program’s main function, instructions of a basic block are decoded and mapped to a source machine dependent intermediate representation M_S -RTL. They are then further abstracted into a machine independent intermediate representation called I -RTL. This intermediate representation abstracts from source machine peculiarities such as calling conventions, but is not as high-level as UQBT’s HRTL intermediate representation. For example, in I -RTL calls to library functions like `printf` are abstracted and translated into calls to the same function on the host machine. The flag semantics of the source machine are emulated. In a next step, the I -RTL represen-

**Figure 2.3:**

Design of the UQDBT dynamic binary translator. Similar to its static counterpart UQBT, the dynamic binary translator abstracts the source machine instructions into a high-level representation which is then encoded into host machine instructions. No emulator is incorporated, and most of UQBT's static abstraction methods are applied.

tation of a basic block is translated into host machine assembly instructions that are encoded into their binary form and stored in the translation cache.

Preliminary tests with a Pentium to SPARC and a SPARC to SPARC translator were performed with toy benchmark programs. Both translators cause a general slowdown of the source program by an average factor of 5 when compared to its native execution.

Transmeta's Crusoe Processor

An interesting approach to dynamic binary translation are Transmeta's Crusoe and, recently, Efficion processors [Kla00, DGB⁺03]. On top of Transmeta's custom VLIW processor sits the Code Morphing Software (CMS) which combines an interpreter, dynamic binary translator, optimizer and runtime system for x86 source machines only. Figure 2.4 shows the Crusoe architecture.

Crusoe's CMS is a system-level implementation of a complete translation framework, and must consequently interpret and translate source machine instructions reliably, implement precise exceptions and handle self-modifying code. The general structure of the CMS resembles that of any common dynamic optimization framework: an interpreter decodes and interprets the x86 source machine instructions and observes the execution frequency of instructions. If a sequence of interpreted instructions exceeds a certain threshold, the CMS passes the address of that sequence to the translator. The transla-

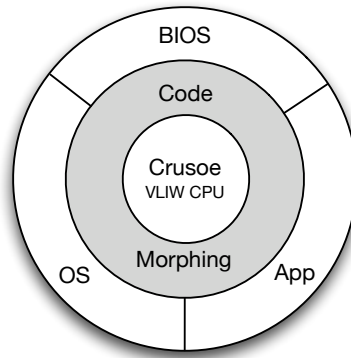


Figure 2.4:

The Crusoe processor is a VLIW processor, on top of which operates the Code Morphing Software (CMS). The CMS implements an x86 interpreter and an optimizing dynamic binary translator. Firmware, operating system and user applications run on top of the CMS.

tor then produces a native VLIW implementation, of the x86 instructions which is then stored into the translation cache.

In order to translate x86 instructions into native VLIW instructions, the CMS implements speculations about the interpreted instructions. These speculations (e.g. two load and store instructions reference non-overlapping memory) are exploited by the translator and verified at runtime by a combination of hardware and software mechanisms. Crusoe registers holding the x86 state are “shadowed” for a translation, i.e. the CMS contains a copy of the register file in addition to the actual working registers. Upon termination of a translation, a special commit shadows the working registers. A failed speculation inside of a translation triggers a native exception that rolls back to the last committed state, and then invokes the interpreter to guarantee precise, though much slower, x86 semantics. Furthermore, the CMS deals with self-modifying code by protecting x86 memory pages against write accesses whenever instructions from that page are translated. An attempt to write to a protected page is intercepted and results in discarding the affected translations.

Unfortunately, available materials do not disclose many implementation details, nor information about the performed optimizations, nor actual performance results.

Dynamite

Dynamite [Tec00] is a commercial dynamic binary translator for applications, distributed by Transitive. It provides support for different RISC, CISC and VLIW architectures such as MIPS, x86, ARM/Xscale, PowerPC and Itanium. Because both Dynamite’s front-end and back-end are written by hand, the migration to a new source or host

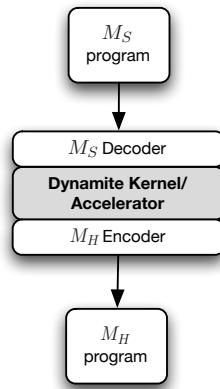


Figure 2.5:

Transitive’s Dynamite dynamic binary translator consists of different front-end and back-end modules. Source machine instructions are abstracted into a machine independent intermediate representation that is first optimized and then compiled into host machine instructions by the Dynamite Kernel.

architecture takes “as little as 6 – 9 months”².

The architecture of Dynamite is depicted in figure 2.5. The machine specific front-end decodes source machine instructions into a machine independent intermediate representation. The intermediate representation is then optimized and translated into host machine instructions by a host machine specific back-end. Little detail is given about the actual implementation of Dynamite’s kernel, nor the optimizations that it performs. Transitive claims a speed-up of RISC to RISC and CISC to CISC translated source programs, though they do not provide actual results.

DAISY

The DAISY (Dynamically Architected Instruction Set from Yorktown) framework [EA97, EAGS01] is an experimental dynamic binary translator developed at the IBM T. J. Watson Research Center. As the target of their VLIW compiler, DAISY proposes a simple VLIW architecture that is a superset of features of common existing machines and functions. Source machine instructions are translated into VLIW instructions that in turn are interpreted by a VLIW emulator. The primary focus of DAISY was to develop a fast VLIW compiler for common machine instructions which is able to harvest inherent instruction level parallelism, and to experiment with architectural features of a flexible VLIW architecture.

An experimental implementation of DAISY translates RS/6000 machine instructions into VLIW instructions. The VLIW itself is represented by a tree which contains primi-

² In comparison to Transitive’s quote, it took less than two weeks to write a complete set of specification files (SLED, SSL and instrumentation definition) for the ARM machine, and to build an experimental ARM frontend for Walkabout/Yirr-Ma.

tives which closely resemble the original RS/6000 source instructions. A VLIW tree, in principle, corresponds to an unlimited multi-way branch with multiple targets and an unlimited set of primitive operations. Nodes of the tree relate to conditional branches, edges relate to primitive operations. The VLIW machine contains 64-bit integer and floating point registers. DAISY also allows for experimentation with different VLIW widths: large VLIW instructions contain 24 fixed point instructions and 7 conditional branches, while smaller VLIW instructions issue 8 fixed point instructions and 3 conditional branches.

The dynamic compiler schedules RS/6000 source machine instructions directly into the VLIW tree representation at runtime, and preserves the precise source machine exception semantics. A single path in a VLIW tree is then selected for execution and the primitive operations of that path are sequentially interpreted by the VLIW interpreter.

Experimental results of DAISY focus on the quality of CISC and RISC translations into VLIW instruction format: they compare dynamically compiled VLIW instructions to the results of traditional VLIW compilers. On average, the DAISY compiler underperforms a traditional VLIW compiler by less than 25%. Further studies and experiments with the generated VLIW code give information about the potential of instruction level parallelism in common benchmark programs, cache performances and the actual compilation overhead.

bintrans

The bintrans framework [Pro01, Pro02, PKS02] is an attempt to generate a dynamic binary translator from specification files. However, the actual instruction decoder and encoder are written by hand, and the specification files are required to be enriched in order to support the actual translation. Bintrans supports PowerPC as source machine and Alpha as the only host machine, both of which run Linux.

Machine specification files for bintrans are written in LISP and define (to some extent) the syntax of machine instructions and their operational semantics. Bintrans generates an interpreter and a dynamic binary translator for the specified machine. During generation, it decomposes source machine instructions into effects and matches these effects to host machine instructions. Thus, a fixed mapping from source to host machine instructions is generated. The matching of effects is implemented using LISP's symbolic list processing features and generates a C source code file. At runtime, when a source machine instruction is decoded, all statically decomposed effects of the instruction are directly emitted as host machine instructions.

Preliminary results are given for two small benchmark programs interpreted by a PowerPC to Alpha translator. They indicate a general slowdown by a factor of 3 to 4 compared to their native execution.

Open Source Projects

Some open source dynamic binary translators have evolved over the past few years, two of which are particularly interesting.

QEMU. The QEMU machine emulator and dynamic binary translator [Bel04] is an emulation project that was started by Fabrice Bellard. It provides experimental implementations for x86, ARM, SPARC and PowerPC source machines running Linux. QEMU manages to run stable on x86 and PowerPC host machines, and experimental implementations exist for Alpha, SPARC, ARM, S390 and IA-64. All of the machine dependent modules are written by hand. As with some of the previously mentioned frameworks, QEMU translates all source machine instructions instead of combining interpretation with the dynamic translation of selected traces.

The QEMU dynamic binary translator implements a technique similar to Yirr-Ma's Inliner (cf. chapter 4). Statically, source machine instructions are split into simpler pseudo instructions. These pseudo instructions are then implemented by C functions and are compiled on a host machine. The unit of translation is a source basic block. The dynamic binary translator uses the compiled pseudo instructions to retrieve a native implementation of the source instructions by concatenating the implementations of the pseudo instructions. The so generated host machine instructions are stored in a code cache and linked to already translated basic blocks. QEMU does not implement any optimizations nor advanced code caching techniques.

The results shown suggest a general slowdown of the executed program by an average factor of 4.1 for a x86 to x86 translation, or by factor 5.0 for a x86 to PowerPC translation.

GNU Lightning. The Lightning library [Bon03] provides an implementation of a virtual instruction set that closely resembles a RISC machine. Lightning is not a translation framework but implements a portable instruction encoder. Several hand-written sets of macros handle the instruction mapping from the virtual instructions to the host machine instructions, and also the encoding of host machine instructions into their binary form. The current version of Lightning supports x86, SPARC and PowerPC host machines and can be ported to new host machines by implementing the virtual instruction set with macros for an actual host ISA.

Generating host machine code with Lightning works as follows. The user of the Lightning library includes the macros for his/her desired host machine into the source code. Instead of selecting and emitting instructions for the actual host machine, however, the user selects and emits instructions for the Lightning virtual machine. During the compilation of the user's code generator, the macros, implementing the emission of virtual

Lightning instructions, are expanded into source code that implements the emission of actual host machine code.

The GNU Smalltalk JIT compiler, for example, uses Lightning for its portable code emission backend.

2.6 Summary and Outlook

In this chapter we revisited and formalized the concepts of dynamic optimizations for machine emulators. We showed that dynamic optimizations can be applied at different levels of the machine emulator, and in combination with one another. Dynamic binary translation is one particular optimization technique that first produces an optimized mapping of the emulated machine state, and then translates interpreted source machine instructions in the context of this optimized state mapping.

Dynamic binary translators are tailored for source/host machine pairs, and they usually implement a fixed set of optimizations. They are traditionally hand-written and therefore difficult to port or extend. In the following chapter 3, we introduce our generic dynamic optimizer Yirr-Ma, an extension to Walkabout machine emulators. Yirr-Ma allows us to experiment with differing hot-spot detection and code caching techniques, along with varying dynamic optimization techniques. In chapter 4 we introduce Yirr-Ma's dynamic partial Inliner approach to dynamic binary translation. Chapter 5 then represents the major part of this thesis: we introduce our specification-driven dynamic binary translator which is yet another module for Yirr-Ma's dynamic optimizer interface.

3 The Yirr-Ma Framework

“Pooh,” said Rabbit kindly, “you haven’t any brain.”
“I know,” said Pooh humbly.

(A.A.Milne)

3.1 Introduction

We showed in the previous chapter that dynamic optimizations can be applied to a given partial state and/or to selected traces of both the source and host machines. The application of such dynamic optimizations aims at improving the overall runtime performance of an interpreted program. We also reviewed the architecture of any common dynamic optimization framework, depicted and explained in figure 2.1.

This chapter introduces the generic dynamic optimization framework Yirr-Ma, an extension to a generated Walkabout machine emulator. Walkabout/Yirr-Ma serves as a vehicle for the experimentation with hot-spot and code caching techniques, but we use Yirr-Ma in particular to experiment with, and to implement, two different dynamic binary translation techniques. The first of the two techniques translates hot source machine traces using dynamic partial inlining of interpretation functions. The second technique translates hot source machine traces by JIT compiling an intermediate representation of the trace’s dynamic operational semantics. We elaborate on the two techniques in depth in chapters 4 and 5, respectively.

In the following sections we introduce the Yirr-Ma framework and its components, and we describe their interface definitions and some selected implementations.

3.2 An Extension to Walkabout: Yirr-Ma

In order to experiment with hot-spot detection and code caching techniques, including their retargetability to different host machines, we designed and implemented the Yirr-Ma framework. Yirr-Ma is an extension to generated Walkabout emulators, and it is linked with the actual emulator core through a custom instrumentation. Figure 3.1 illustrates the general architecture of our framework together with Walkabout. In addition to the syntax (SLED) and semantics (SSL) specification files, a custom instrumentation definition is used to build an instrumented Walkabout emulator. The generated C/C++ source code of the instrumented machine emulator is then compiled and linked with

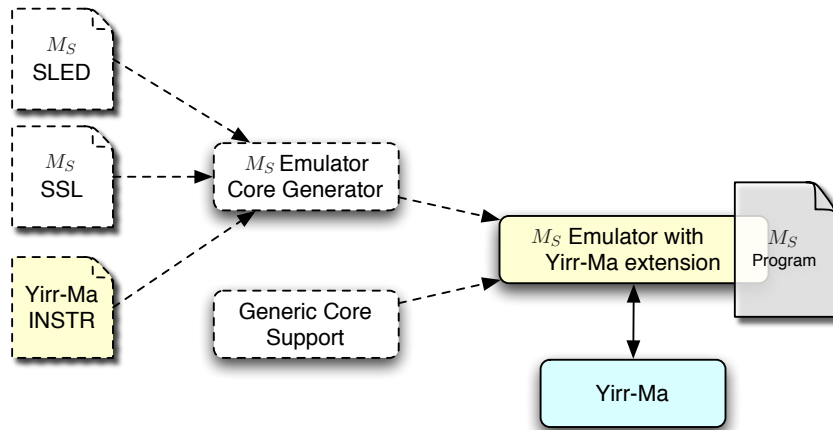


Figure 3.1:

Using Walkabout (white), SLED and SSL specification files, as well as a custom instrumentation description, an instrumented machine emulator is generated. The emulator is then compiled and linked with generic core support modules and with Yirr-Ma (green). Through its added instrumentation, Yirr-Ma observes and intercepts the interpretation of a source program. Yellow components are shared between Walkabout and Yirr-Ma.

both Walkabout’s generic core support modules and with the Yirr-Ma library. Using its custom instrumentation, Yirr-Ma observes the interpretation of the source program, and it selects source machine traces for dynamic optimization. Optimized traces are emitted in the form of *dynamic functions*: ordinary host machine functions that replace the calls to the individual interpretation functions with one single interpretation function for the entire trace. The generated dynamic functions are then stored in a code cache for subsequent invocation.

Yirr-Ma consists of several components, as depicted in figure 3.2. The *Collector* observes the interpretation of an emulated source program and, with every interpretation of a CTI, queries the *Hot-Spot Detector* to decide whether the target address of the branch instruction (either taken or fall-through) is hot. If the target instructions are indeed hot, the Collector switches into collection mode and begins collecting semantic information about the interpreted instructions, starting with the instruction at the branch target address. Semantic information is collected until a CTI that defines the end of a trace (see the next section) is interpreted, thus terminating the current hot trace. The collected information is then passed to the *Emitter* which, in turn, produces an optimized version of the interpreted trace in form of a dynamic function. The Emitter returns the dynamic function to the Collector that stores it in Yirr-Ma’s code cache. If a subsequent branch to a hot trace is interpreted, the *Dispatcher* intercepts the interpretation and invokes the generated dynamic function instead of interpreting the original trace. The Hot-Spot Detector, the Emitter and Code Cache are abstract classes defining software

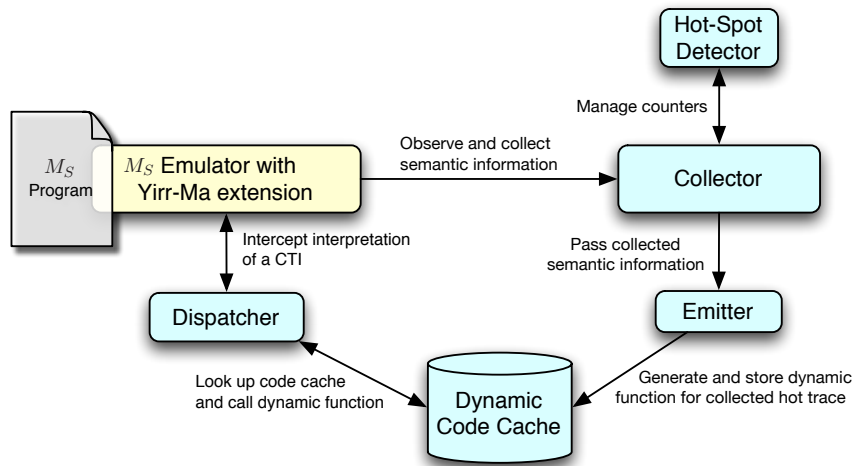


Figure 3.2:

Internal architecture of the Yirr-Ma framework. Semantic information about an interpreted hot trace is collected, and then emitted in form of a dynamic function. These dynamic functions are stored in a dynamic Code Cache, from where they are invoked by the Dispatcher.

interfaces, thus allowing for the experimentation with different implementations.

3.3 The Collector and Dynamic Semantic Information

Both the Collector and the Dispatcher are Yirr-Ma’s only two components exposed to the Walkabout emulator by a custom instrumentation (cf. sections 4.3 and 5.5). This instrumentation allows the Collector to observe the interpretation of the source program, and the Dispatcher to intercept its interpretation. If the emulator interprets a CTI, the Collector, using Yirr-Ma’s instrumentation, passes the addresses of the CTI and its target instruction to the Hot-Spot Detector. When the target instruction becomes hot, the Collector switches into collection mode and begins to collect semantic information about the interpreted instructions of the hot trace until the end of a trace is encountered. Similar to basic blocks that are terminated by a CTI, the end of a trace is defined by

- a conditional branch instruction with static target addresses
- an unconditional CTI with no static target addresses, e.g. a return or computed branch instruction.

Note that unconditional CTIs with a static target address do *not* terminate a trace, i.e. a function call or a branch-always instruction are considered a part of the trace rather than its terminating instruction. The collected semantic information is then passed to the Emitter.

The semantic information about a trace is collected per interpreted instruction and consists of two different types. Similar to a function definition and its function application in functional languages, we divide the semantic information into

- the static *instruction implementation* that represents the operational semantics of a source machine instruction
- the dynamic *instruction application* that represents the actual operand values for a particular instruction implementation.

This differentiation reflects the implementation of the Walkabout emulator. Instruction implementations are the instruction interpretation functions, and instruction applications are the actual parameter values for interpretation functions.

Depending on the implementation of Yirr-Ma’s Emitter interface, the semantic information, and with it both the instruction implementation and the instruction application, represents slightly different information which is tailored for the Emitter. We discuss the respective differences and the instrumentation of Walkabout’s interpretation functions in the according sections later.

3.4 The Hot-Spot Detector

The optimization of a source machine trace at runtime consumes time and memory resources that compete with the interpretation of the source program. A hot-spot detector thus implements methods, often based on heuristics, which select traces for optimization such that resources spent on that optimization are more likely to be amortized.

The static control-flow of a program, i.e. its possible execution paths, is represented by a control-flow graph [Sco99]. Nodes represent basic blocks, whilst edges represent branches from one basic block to another. A hot-spot, at runtime, consists of frequently and in temporal proximity interpreted instructions of one or more logically adjacent basic blocks, for example the body of a loop. The literature describes two different techniques to detect hot-spots in a program’s dynamic control-flow:

- *node based counting*: a counter is associated with every basic block, and each interpretation of that basic block increments the counter
- *edge based counting*: a counter is associated with the edge between two basic blocks, and each branch along that edge increments the counter.

Yirr-Ma defines the implementation of its hot-spot detector by an interface, shown in the UML class diagram (cf. [FS99]) in figure 3.3. We currently implement only node based hot-spot detection. The Collector queries the hot-spot detector whenever a CTI is interpreted, passing it the relevant information about the dynamic control-flow of the

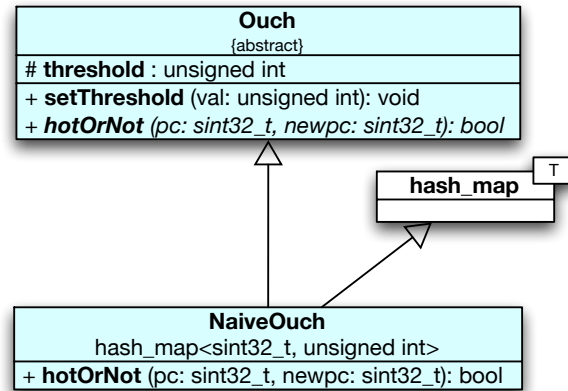


Figure 3.3: Class diagram of Yirr-Ma's hot-spot detector class `Ouch`. Currently, only one sub-class implements the interface.

interpreted program. In turn, the hot-spot detector manages that information and all of its related counters. If a counter exceeds a specified threshold, the hot-spot detector signals this to the Collector, thus triggering the collection of semantic information.

The counter threshold that defines a node (i.e. a basic block) or an edge (i.e. a branch) as “hot” is critical during the interpretation of a source program, and is an individual value for every different program. If the threshold is too low¹, too much semantic information is collected too frequently, forcing Yirr-Ma to spend much time with translation, hot-spot management and code cache maintenance. This, in turn, causes an overall slowdown of the interpreted source program. In contrast, if the threshold is too high, Yirr-Ma does not translate potentially worthwhile traces, thus spending more time with a program's interpretation without harvesting its full potential. In general, the fine-tuning of hot-spot detectors for dynamic optimization is an important and much researched problem (cf. [Kis99] for the dynamic analysis of a program's behavior, and the continuous application of dynamic optimizations).

Listing 3.1 shows actual debug output of Yirr-Ma: two actual hot traces of an interpreted SPARC source program. The first hot trace, taken from Solaris' `ld.so` dynamic linker, is also a single basic block which serves as a reference example throughout the remainder of this thesis. The second hot trace was taken from the `compress95` benchmark program from the SpecINT'95 benchmark suite, and illustrates how unconditional CTIs with a fixed target address are considered part of a trace rather than a termination criterion. Additional debug information is given for both traces: the taken and fall-through addresses of the trace, as well as the hexadecimal encoding of the instruction and their

¹ The term “too low” is relative. Its dynamic value is defined by heuristics during the runtime of a program, but is not a statically determined and fixed constant.

```

1  trace starting at 0xfe02a24
2  out edges: taken 0xfe02a24 fallthrough 0xfe02a34
3  0FE02A24: e2 04 00 00      LDreg      %l0, %g0, %l1
4  0FE02A28: 80 90 00 11          tst        %l1
5  0FE02A2C: 12 bf ff fe          BNE        0xfe02a24
6  0FE02A30: a0 04 20 04          inc        4, %l0

```

```

1  trace starting at 0x100821ac
2  out edges: taken 0x1001d7f4 fallthrough 0x1001d71c
3  100821AC: 90 10 00 12      mov_      %l2, %o0
4  100821B0: b8 27 20 01      dec        1, %i4
5  100821B4: 40 00 e3 d6      call__    0x100bb10c
6  100821B8: 92 04 62 67      ADDimm    %l1, 615, %o1
7  100BB10C: 01 00 00 00      NOP
8  100BB110: 30 80 0f 3d      BA,a      0x100bee04
9  100BEE04: 82 10 00 0f      mov_      %o7, %g1
10 100BEE08: 7f fd 7a 41      call__    0x1001d70c
11 100BEE0C: 9e 10 00 01      mov_      %g1, %o7
12 1001D70C: 81 82 00 00      WRyreg    %o0, %g0, %y
13 1001D710: 98 aa 20 0f      ANDNccimm %o0, 15, %o4
14 1001D714: 02 80 00 38      BE        0x1001d7f4
15 1001D718: 1b 3f ff c0      sethi     %hi(0xffff0000), %o5

```

Listing 3.1:

Two examples of interpreted hot SPARC traces. Note that SPARC has delayed CTI semantics, i.e. the instruction after a CTI is usually executed before the actual branch takes place. The first of the two traces serves as an example trace throughout the remainder of this thesis.

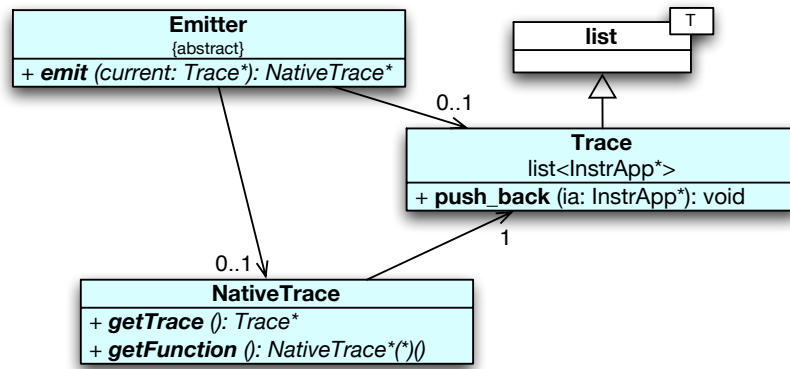
addresses.

The Collector collects semantic information, i.e. the instruction implementation and instruction application, for every detected hot-spot of an interpreted program, and passes that information to the Emitter for dynamic optimization.

3.5 The Emitter Interface and Dynamic Functions

Yirr-Ma's implementation of a dynamic optimizer is defined by the Emitter interface: it defines the generation of dynamic functions from selected hot traces. The class diagram in figure 3.4 illustrates the interface. Chapters 4 and 5 present two implementations of the Emitter interface: a dynamic partial inliner and a dynamic binary translator, respectively. Given the collected semantic information for a hot trace, an implementation of the Emitter interface must produce and return a dynamic function. Dynamic functions are Yirr-Ma's unit of code generation and they are treated by Yirr-Ma as black boxes. They abstract from the actual host machine implementation of the hot trace.

The functioning of Yirr-Ma's dynamic functions is simple. Recall that the Walkabout emulator generator generates a set of interpretation functions, each of which receives the operands of a source machine instruction as actual parameter values and then interprets the source machine instruction. Similarly, the Yirr-Ma Emitter generates a single

**Figure 3.4:**

Class diagram of Yirr-Ma’s Emitter interface. The interface defines one public virtual method `Emitter::emit()` that must be implemented by sub-classes. This method takes as its only parameter the `Trace` of collected semantic information (i.e. a list of instruction applications and their associated instruction implementations), and generates a dynamic function, implemented by the `NativeTrace` class.

dynamic function at runtime which interprets the entire hot trace, thus short-cutting the fetch-and-execute cycle of the Walkabout emulator. Both the Walkabout emulator and Yirr-Ma guarantee the consistency of the emulated machine state at the function’s boundaries: Walkabout before and after every interpreted instruction, and Yirr-Ma before and after every executed dynamic function. This approach is sufficient enough to interpret most user-level applications and exploits the inherent potential for dynamic optimizations. This encapsulation of a trace into a black box for a dynamic function, however, hampers precise and detailed machine emulation, because the mapping from generated host machine instructions back to the original interpreted instructions and their consistent state can not be performed without additional bookkeeping information.

Dynamic functions are in fact *anonymous* functions: they do not have a symbolic name or any other information associated. They are implemented by the `NativeTrace` class (cf. figure 3.4). An object of the `NativeTrace` class contains the actual code for one dynamic function and other bookkeeping information; the prototype of the dynamic function itself is defined as:

```
NativeTrace* noname(void); // return pointer to itself
```

The invocation of a dynamic function takes no parameters and returns a pointer to the dynamic function’s `NativeTrace` object (see below). The Emitter generates a dynamic function, wraps it in a `NativeTrace` object and returns this object to the Collector which stores it in Yirr-Ma’s code cache.

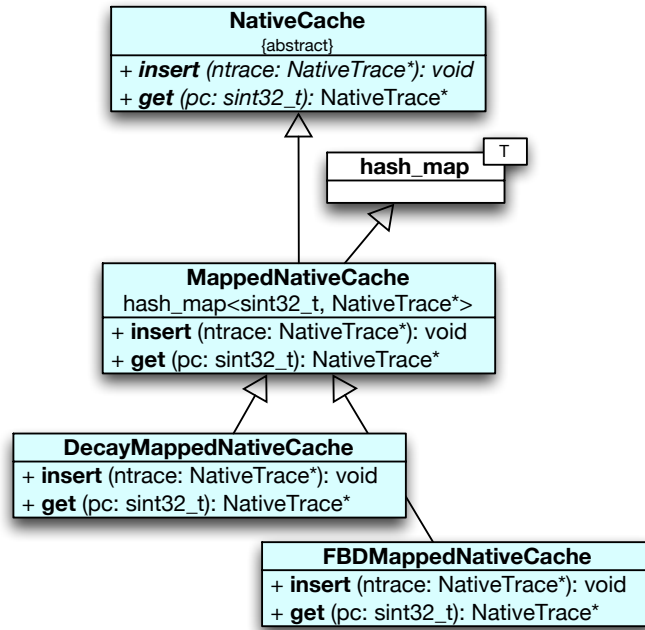


Figure 3.5:

Yirr-Ma defines dynamic code caches by an interface that allows for the experimentation with different implementations, and thus code caching and eviction techniques. Our current code cache `MappedNativeCache` is implemented by a hash map.

3.6 The Code Cache

`NativeTrace` objects, and therewith dynamic functions, are stored in Yirr-Ma’s code cache that associates the source machine address of an interpreted source machine trace with its implementing dynamic function. Because the properties and behavior of dynamic code caches are outside of the scope of our research, the current implementation of our code cache is a simple unlimited cache that does not implement any eviction strategies. An elaborate investigation of different dynamic code caches is given in [HS02].

Dynamic functions that implement logically adjacent hot traces can be linked² with one another once they are inserted into the code cache. Linking is a technique which further increases the performance of the interpreted program by avoiding unnecessary dynamic dispatches and context switches between interpretation and invocation of dynamic functions. A link from one dynamic function to another is implemented by a conditional branch right behind the target function’s prologue, and is appended to the end of the dynamic function. Whether the branch is taken or not depends on the boolean value of the conditional branch instruction at the end of the interpreted trace. In fact, the link models the interpreted control-flow exactly. The execution of several linked

² Some authors refer to the dynamic linking of independently generated instruction sequences as “chaining”.

dynamic functions appears to the host machine as one single function call.

Our current implementation of Yirr-Ma's code cache interface implements four different linking techniques:

- *no linking*: generated dynamic functions are not linked with one another, thus causing the repeated invocation of the same or logically adjacent dynamic functions
- *local linking*: the dynamic function is only linked with itself during its generation; this works for closed source machine traces where the target of the terminating CTI is the beginning of the trace itself (e.g. our first example trace in listing 3.1)
- *demand linking*: if the Dispatcher invokes two independent dynamic functions one after the other it links both, thus saving the code cache lookup time during subsequent invocations; however, in order to identify the last dynamic function in a list of already linked dynamic functions, and to link that last function to its new successor, every dynamic function must return a pointer to itself (cf. the prototype of a dynamic function in the previous section)
- *full linking*: a dynamic function, when added to the code cache, is compared to all other existing entries in the code cache and linked with logically adjacent dynamic functions; using this technique, not only is the inserted dynamic function linked with its successors, but it is also linked with existing logically preceding dynamic functions.

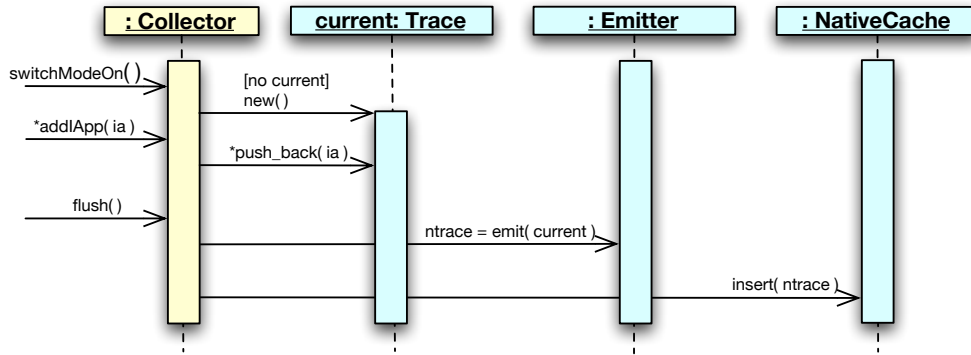
Whereas the first linking technique does not add any overhead to the emulator when entries are added to the code cache, so does the last one add significant overhead for our unlimited cache, particularly when programs constantly change their working set. However, the last technique guarantees a minimal number of code cache lookups and dispatches compared to the first one, where every invocation of a dynamic function requires a dynamic dispatch. Didum³.

Whenever the Walkabout emulator interprets a CTI, it invokes the Yirr-Ma's Dispatcher. The Dispatcher searches the code cache for a dynamic function that implements the target trace of the CTI, and then invokes that function.

3.7 The Dispatcher

The interpretation of a machine instruction, in particular that of a CTI, modifies the value of the emulated `%pc` register and sets it to the address of the instruction which is interpreted next. This address, however, may be the beginning of a cached hot trace. Therefore, Yirr-Ma's Dispatcher looks up the code cache for a dynamic function that is

³ The phrase "didum", having no intended meaning, is of no relevance for this thesis, whatsoever.

**Figure 3.6:**

Sequence diagram that illustrates the successful generation of a dynamic function. When the Collector switches into collection mode, it begins to collect semantic information into a trace. The Emitter generates the dynamic function from that trace and the Collector then stores into the code cache.

associated with that target address⁴. If no dynamic function was found, the Walkabout emulator continues to interpret source machine instructions; otherwise, the dispatcher invokes that function, thus transparently advancing the emulated source machine state.

The invocation of a dynamic function for a hot trace has the same effect on the emulated machine state as the continuous invocation of interpretation functions for that trace's instructions: the consistency of the emulated machine state is always guaranteed at the function's boundaries. Furthermore, as the code cache populates with dynamic functions, the chance of linking dynamic functions increases. This allows the Walkabout emulator to run *without* fetching, decoding and interpreting source machine instructions, *whilst* maintaining a consistent emulated machine state. When a dynamic function (linked in a list of same) returns, the emulated `%pc` register contains the address of the instruction that is to be interpreted next. The Walkabout emulator now continues to interpret the source program just like before and without further context switch.

3.8 Implementation

Figure 3.6 shows the sequence diagram for a successful generation of a dynamic function and its insertion into the code cache. In collection mode, Yirr-Ma's Collector continuously adds the instruction applications and instruction implementations of the interpreted instructions, the hot trace's semantic information, to the Trace object `current`. The interpretation of a trace terminating CTI instruction then causes the trace to be flushed and passed to the Emitter which first generates the dynamic function, and then

⁴ Note that we do not regard unconditional CTI with a static target address as a CTI (although they are per definition); they thus do not trigger a cache lookup.

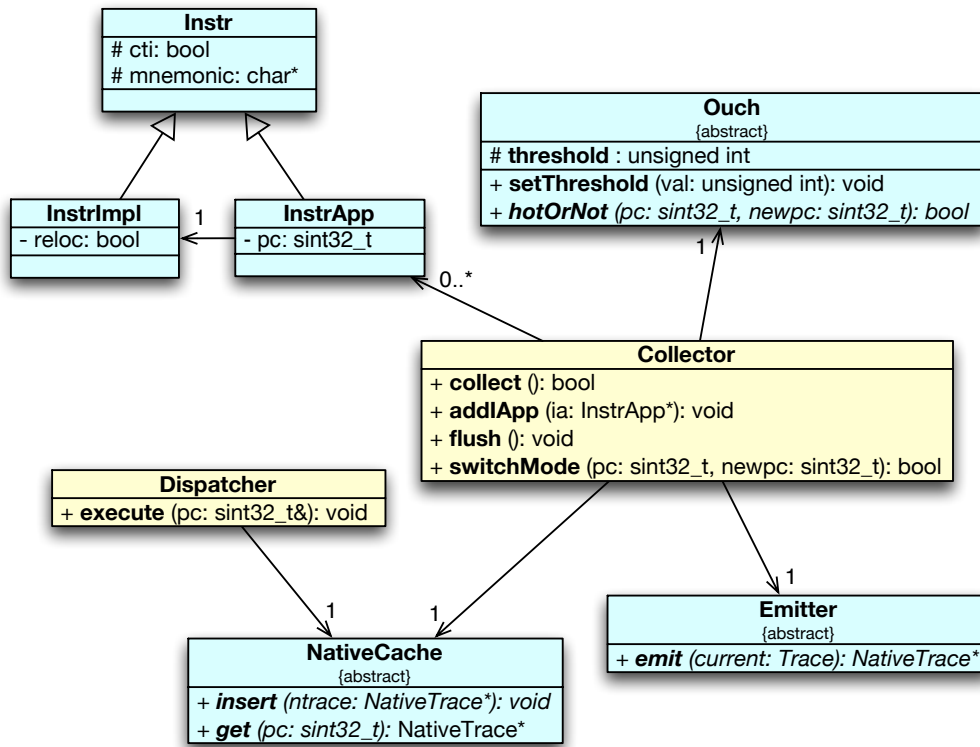


Figure 3.7: Class diagram of Yirr-Ma's main classes and their respective associations. Most of Yirr-Ma's modules are defined by abstract classes, such that different implementations of the modules can replace one another.

returns it to the Collector. The Collector then inserts the generated dynamic function into the code cache.

The Collector

The design of Yirr-Ma's Collector and its associated main classes is shown in the class diagram in figure 3.7. Each of Yirr-Ma's components is defined and implemented by a single class, reflecting Yirr-Ma's modular architecture. The hot-spot detector, the code cache and the Emitter are defined by interfaces, or abstract classes in C++, such that we are able experiment with different implementations using sub-classes. The Collector invokes these components through their interface methods, which we describe in more detail below. Only the Collector and the Dispatcher are exposed to the Walkabout emulator core, and we discuss how the Walkabout emulator and the Yirr-Ma framework are linked together in the following.

The Collector performs a number of tasks. Amongst the most important are the management and detection of hot-spots, and the collection of semantic information for an interpreted hot trace. With every interpreted CTI the emulator passes control-

flow information to the Collector by calling the `Collector::switchMode()` method. The collector then queries the hot-spot detector to determine whether or not that control-flow information defines a hot trace by calling the `Ouch::hotOrNot()` method. If this method returns true, the collector triggers the collection mode. If in collection mode, indicated by `Collector::collect()` returning true, the Collector collects semantic information about the interpreted instructions. The semantic information is implemented by the `InstrImpl` class for the instruction implementation, and by the `InstrApp` class for the instruction application. For an interpreted hot instruction, the Collector collects the instruction application which has an association with its related instruction implementation. Both types of objects are created by the instrumentation code added to the interpretation functions.

The semantic information of an interpreted instruction is passed to the Collector by invoking its `Collector::addIApp()` method for every interpreted instruction. The Collector then builds a trace of semantic information by storing the individual instruction applications in a linked list. Upon interpretation of a trace terminating CTI, the Collector sets the static out-edges for the trace and prepares the generation of a dynamic function for the collected trace. A call from the emulator core to the Collector's `Collector::flush()` method finally passes the collected trace information to the Emitter.

Hot-Spot Detector

Whenever it interprets a CTI, the emulator core passes control-flow information to Yirr-Ma's Collector. The Collector then queries the hot-spot detector by calling its `Ouch::hotOrNot()` method. The `Ouch` interface defines this method: it takes two parameters, and returns a boolean value. The first parameter is the address of the interpreted CTI instruction itself, and the second parameter is the address of the next interpreted instruction (either taken or fall-through). An implementation of the interface may thus support either node-counting or edge-counting detection techniques. The method returns either true (hot) or false (not) to the Collector that, in turn, switches the collection mode on or off accordingly.

As can be seen in the class diagram in figure 3.3, Yirr-Ma currently implements only one node-based detector using an unlimited hash map, instantiating the STL template class `hash_map` (cf. [Jos99]). Without further maintenance of the hot-spot information, our implementation of the hash map grows big and slow rapidly because *every* interpreted CTI becomes an entry in the hash map. In fact, the dynamic constraints of a hot-spot detector are similar to those of a dynamic code cache, and thus outside the scope of this thesis. Furthermore, entries in our current implementation do not expire, nor are they replaced: Yirr-Ma does not support the detection of *temporal* hot-spots. As a consequence, traces that are not part of a temporal hot-spot but happen to be interpreted sufficiently often over time are falsely detected as "hot".

The Code Cache

Like both the hot-spot detector and the Emitter, so is the dynamic code cache defined by an interface named `NativeCache`. We implemented our code cache by instantiating the template class `hash_map` in our `MappedNativeCache` class, as can be seen in figure 3.5. The template class is parameterized such that it maps from a source machine address (the beginning of an interpreted hot trace) to a dynamic function (the hot trace’s improved version). The code cache has unlimited size and allows for the storage of an unlimited number of cache entries. This, in turn, poses a number of performance constraints on the emulator.

As the code cache populates, the insertion of new entries and the lookup of existing ones takes longer, and thus linking becomes a more time consuming process. Interpreted programs that continuously change their working set perform particularly poor.

For proof of concept and experimental purposes, we implemented two sub-classes of our simple mapping cache: a decaying code cache and a feedback-directed code cache. The decaying code cache `DecayMappedNativeCache` is parameterized with the number of cache entries which it is able to store. It uses a Least-Recently-Used replacement strategy to evict existing cache entries, if new ones are added to a full code cache. Because dynamic functions stored in this cache may be removed at any time, the applicable linking strategies are restricted to local linking or no linking at all. For a more detailed discussion on dynamic code caching techniques and their impact on dynamic trace linking see [HS02]. The second implementation is a feedback-directed cache named `FBDMappedNativeCache`. Upon object destruction, this code cache produces a C/C++ source code file from its content. The generated file can be compiled and optimized statically on the host machine. The code cache then loads its optimized content for subsequent executions of the source program upon object construction of the code cache. We revisit this particular code cache in more detail in section 5.2 when we investigate and discuss the quality of dynamically translated machine instructions.

Linking of Dynamic Functions

When a dynamic function is inserted into a code cache, it can be linked with other logically adjacent functions. The linking of dynamic functions is host machine specific and as such implemented by methods of an Emitter interface implementation. Depending on the chosen linking technique, dynamic functions must be prepared for linking by inserting placeholder instructions during generation. Placeholder instructions are host `nop` instructions, which are overwritten with actual branch instructions, when the code cache links two dynamic functions. Links follow the control-flow of the interpreted program and therefore they depend upon the value of the `r_COND` pseudo register. This register is not part of the emulated machine, rather it is added to support Yirr-Ma by extending the source machine’s state specification. Interpreted CTIs evaluate their

```
1 0x100b9cdc: lis      r9,4107      ; load address of r_COND
2 0x100b9ce0: ori      r9,r9,35602
3 0x100b9ce4: lbz     r9,0(r9)   ; value of r_COND
4 0x100b9ce8: cmpwi   r9,1       ; if not set, skip taken
5 0x100b9cec: bne-    0x100b9d00
6 0x100b9cf0: nop
7 0x100b9cf4: nop
8 0x100b9cf8: nop
9 0x100b9cfc: nop
10 0x100b9d00: lis     r9,4107    ; load address of r_COND
11 0x100b9d04: ori     r9,r9,35602
12 0x100b9d08: lbz    r9,0(r9)   ; value of r_COND
13 0x100b9d0c: cmpwi  r9,0       ; if set, skip fallthrough
14 0x100b9d10: bne-   0x100b9d24
15 0x100b9d14: nop
16 0x100b9d18: nop
17 0x100b9d1c: nop
18 0x100b9d20: nop
19 0x100b9d24: ...
```

Listing 3.2:

Empty link for dynamic functions on a PowerPC host machine. When the dynamic function is linked, the `nop` instructions are overwritten by an unconditional branch into the target dynamic function.

branch condition into this register⁵ and branch, i.e. set the emulated `%pc` register, accordingly. Likewise must the link of a dynamic function test the value of `r_COND`, and then branch to the dynamic function that implements either the taken or fall-through source machine trace.

Listing 3.2 shows the PowerPC implementation of an empty link with placeholder instructions that is generated alongside the dynamic function. The instructions in lines 1-5 load the value of the emulated `r_COND` register into register `r9` and test it against true or false (lines 10-14). When a dynamic function is inserted into the code cache and linked with another dynamic function, the `nop` placeholder instructions for the taken (lines 6-9) or fall-through (lines 15-18) link are overwritten with an unconditional branch right behind the prologue of the target dynamic function⁶:

```
0x100b9cf0: lis      r9,4107 ; load address of target function
0x100b9cf4: ori      r9,r9,39324
0x100b9cf8: mtctr   r9      ; and branch
0x100b9cfc: bctr
```

Because the implementation of our code cache is unlimited in its size, links must always be implemented as absolute 32-bit branches instead of faster `%pc`-relative branches.

Whereas the above links are patched (either during demand linking by the Dispatcher or full linking by the code cache itself) *after* the dynamic function was generated and

⁵ The branch condition of unconditional CTIs always evaluates to either true or false.

⁶ Inserting instructions into existing code and overwriting placeholder instructions is a technique sometimes referred to as “code stitching” or “patching”.

inserted into the code cache, a local link is emitted *during* the generation of the dynamic function. A local link is a `%pc`-relative backwards branch to the beginning of the same dynamic function and is implemented as follows:

```

0x100b9cdc:  lis      r9,4107      ; load address of r_COND
0x100b9ce0:  ori      r9,r9,35602
0x100b9ce4:  lbz      r9,0(r9)    ; value of r_COND
0x100b9ce8:  cmpwi   r9,1        ; test the branch condition
0x100b9cec:  beq+    0x100b999c  ; branch if true

```

Again, the value of the `r_COND` register is tested and, if the emulated branch condition is true, the backward branch to the beginning of the dynamic function is taken.

With dynamic linking switched off, Yirr-Ma does not emit any placeholder instructions for the generated dynamic functions.

Walkabout's Emulator and Yirr-Ma's Dispatcher

A Walkabout emulator implements the fetch-and-execute cycle of the source machine: a source machine instruction is fetched from the emulated memory segment, decoded and then interpreted over the emulated machine state. The invocation of a dynamic function (in preference to the interpreted fetch-and-execute cycle) can, at any time, replace the interpretation of source machine instructions, thus making the dynamic function transparent to the Walkabout emulator.

Using a custom instrumentation, Yirr-Ma's Dispatcher extends the fetch-and-execute cycle of a Walkabout emulator. The interpretation of a CTI sets the value of the `r_CTI` pseudo register to true, indicating to the Dispatcher that the last interpreted instruction was a CTI (cf. listing 3.3). The Dispatcher takes the current value of the `%pc` register (i.e. the starting address of a new trace) and searches the code cache for a dynamic function that implements the new trace. Listing 3.4 shows the implementation of the Dispatcher's main method `Dispatcher::execute()`. If no function was found, the method returns and the interpretation of the source program continues. If a dynamic function was found in the code cache, it is invoked and advances the state of the emulated source machine. When the dynamic function returns, the interpretation of the source program continues with the new state.

3.9 Summary and Contribution

We introduced the generic dynamic optimization framework Yirr-Ma that is linked with, and extends, a generated Walkabout emulator using a custom instrumentation. Yirr-Ma defines its main components by interfaces, allowing us to experiment with different implementations and techniques for hot-spot detection, code caching of generated dynamic functions and with various dynamic optimizations.

Using the Emitter interface in particular, we can now implement and experiment with

```
1 // Walkabout's fetch-and-execute cycle is implemented here
2 void run()
3 {
4     while(true)
5         executeOneInstruction();
6 }

```

```
1 // interpret one instruction: compute a new state for the
2 // emulated source machine
3 void executeOneInstruction()
4 {
5     regs.r_CTI = 0; // reset the CTI bit
6     execute(regs.r_pc); // fetch, decode and interpret the
7                         // current instruction
8
9     // if the instruction was a CTI then the r_CTI register
10    // is true and we can invoke the Yirr-Ma dispatcher
11    if (regs.r_CTI)
12        dispatcher.execute(regs.r_pc);
13 }
```

Listing 3.3:

Yirr-Ma's Dispatcher is invoked by a Walkabout emulator whenever it interprets a CTI. If a dynamic function for the current %pc value exists in the code cache, it is executed and changes the state of the emulator consistently. No explicit context switch is required here. Note that the shown dispatch does not support delayed CTI semantics for source machines; however, an appropriate extension is trivial.

```
1 void Dispatcher::execute(sint32_t& pc) const
2 {
3     // look up the code cache for a dynamic function that
4     // implements the trace starting at pc
5     while (NativeTrace* ntrace = cache.getNativeTrace(pc))
6         (ntrace->getFunction())();
7
8     return;
9 }
```

Listing 3.4:

The Dispatcher's code cache lookup and invocation of dynamic functions is implemented by the `Dispatcher::execute()` method. Given the current value of the emulated program counter, i.e. the beginning of a trace, the Dispatcher finds an implementing `NativeTrace` object and invokes its dynamic function.

different dynamic optimizers. An implementation of the Emitter interface must generate a dynamic function, i.e. it has to return a `NativeTrace` object. When executed, this dynamic function consistently advances the emulated machine state, but may temporarily neglect the state consistency during its execution. Using the Yirr-Ma framework as described in this chapter, we implemented two different techniques for retargetable dynamic binary translation that we elaborate upon in the following two chapters.

4 Dynamic Code Generation using Partial Inlining

Det tar noen millioner år å skape et menneske. Og det tar noen sekunder å dø.

(Jostein Gaarder)

4.1 Introduction

A generated Walkabout machine emulator is linked with our generic dynamic optimization framework Yirr-Ma by a custom instrumentation. Yirr-Ma detects hot traces by observing the interpretation of a source program, it provides an interface for the generation of dynamic functions from selected hot traces, and it uses different code cache implementations to manage the generated dynamic functions. Yirr-Ma's dynamic functions interpret hot traces, and thus advance the emulated machine state transparently and consistently. Implementations of the Emitter interface must generate dynamic functions from selected hot traces of semantic information that is provided to the implementation by Yirr-Ma.

In this chapter we introduce a dynamic binary translation technique which employs *dynamic partial inlining* of Walkabout's instruction interpretation functions to generate dynamic functions. This work, along with an earlier version of Yirr-Ma, was published at the Workshop on Binary Translation in 2002 [TG02]. The dynamic Inliner class extends Yirr-Ma's Emitter interface, while host machine specific sub-classes of the Inliner class are hand written. First, we discuss the semantic information that is required for our dynamic inlining technique, and we show how this information is collected by an instrumented Walkabout emulator for SPARC source programs. Second, we illustrate our approach to dynamic partial inlining with actual implementations for SPARC, PowePC and Pentium host machines. Experimental results conclude this chapter. The insights and experiences gathered from this work motivate our specification-driven dynamic binary translator in chapter 5.

The approach taken in this chapter is similar to the concept of *superoperators* for C interpreters as described in [Pro95], and to the selective dynamic inlining of threaded code for virtual machines [PR98]. In contrast to those two, however, Yirr-Ma's Inliner

handles interpreted machine instructions of arbitrary source machines that run on different host machines, and functions more as a case study and a proof of concept for the generic dynamic optimization framework Walkabout/Yirr-Ma.

4.2 The Semantic Information

The generation of a dynamic function, i.e. an optimized version of the hot trace, requires the semantic information of a hot trace. As discussed in section 3.3 we distinguish the semantic information into instruction implementation and instruction application. By instrumenting the interpretation function of *every* source machine instruction, Yirr-Ma is able to collect both

- the implementation of a source machine instruction that is surrounded by first-class labels inside the interpretation function
- the actual parameter values of an interpretation function that are wrapped for the instruction application.

The individual instruction application objects are associated with their respective instruction implementation and then passed to Yirr-Ma's Collector.

The key idea behind this dynamic binary translation approach lies in the differentiation of the semantic information. As shown in equation 2.1 on page 37, a trace t^S of source machine instructions is translated into a trace of host machine instructions through the application of the translation function $\tau(t^S)$. In Walkabout, the SSL specification of a machine defines the operational semantics of machine instructions over the machine state. The emulator generator then transforms the source machine specification into a C/C++ implementation: data structures abstract the source machine state, and interpretation functions implement the operational semantics of machine instructions. By compiling the generated source code, the operational semantics is translated into host machine instructions. The static C/C++ compiler therefore *implements* the injective trace mapping function φ_I for individual instructions (i.e. traces of length one), and Yirr-Ma utilizes the results of that translation at runtime. By generating a dynamic function, Yirr-Ma produces an optimized version of the original calling sequence of interpretation functions. Therefore, it implements the optimization function $\omega_{I_H^*}$ in the above equation. The data required to produce a dynamic function is provided by the instrumentation code which passes the necessary semantic information about the trace to the Emitter.

```

1  table9 [ "ANDREG", "ORREG", "XORREG", "ANDCCREG", "ORCCREG",
2          ...
3          ]
4
5  table9 rs1, rs2, rd
6  {
7      // the PC of the interpreted instruction
8      sint32_t _pc = SSL(%pc);
9
10     // force actual parameter values into host machine registers
11     register sint8_t reg_rs1 asm (REG1) = PARAM(rs1);
12     register sint8_t reg_rs2 asm (REG2) = PARAM(rs2);
13     register sint8_t reg_rd  asm (REG3) = PARAM(rd);
14
15     // instruction implementation begins here
16     Lstart:
17     {
18         // make values available to the implementation
19         sint8_t PARAM(rs1) = reg_rs1;
20         sint8_t PARAM(rs2) = reg_rs2;
21         sint8_t PARAM(rd) = reg_rd;
22
23         // actual instruction implementation
24         SSL_INST_SEMANTICS
25     }
26     Lend:
27
28     // collect instruction implementation and instruction
29     // application of the interpreted instruction for the current
30     // trace
31     if (collector.collect()) {
32         static const InstrImpl ii(&&Lstart, &&Lend);
33         InstrApp* ia = new InstrApp(_pc, &ii, Param(PARAM(rs1)),
34                                   Param(PARAM(rs2)), Param(PARAM(rd)));
35         collector.addInstrApp(ia);
36     }
37 }

```

Listing 4.1:

Instrumentation of interpretation functions for SPARC's logical instructions. The actual implementation of the source machine instruction is enclosed by the first-class labels `Lstart` and `Lend`. Yirr-Ma's Collector collects the actual parameter values of the interpretation function, together with the value of the emulated `%pc` register and the instruction implementation.

4.3 Instrumenting a Walkabout Emulator for Yirr-Ma's Inliner

The Walkabout emulator generator allows for the instrumentation of arbitrary interpretation functions (cf. section 1.7.3). For our purposes, we instrument each of the emulator's interpretation functions for Yirr-Ma's Inliner. Listing 4.1, as an illustrating example, shows the instrumentation of SPARC's logical instructions with three register operands.

The generated interpretation functions take two types of actual parameter values: either an integer value which is the index into the emulated register file, or the immediate operand value of the interpreted instruction. These actual parameter values are forced into host machine registers (lines 11-13), and they are then passed to the actual implementation of the interpreted instruction (lines 19-21). When the dynamic function is generated at runtime, a prologue for the inlined instruction semantics writes the collected actual parameter values into the dedicated host machine registers. The implementation of the host machine instructions itself is enclosed by the first-class labels `Lstart` and `Lend` (lines 16 and 26). Using these labels, the instruction implementation is encapsulated into a constant `InstrImpl` object, as shown in line 32. When Yirr-Ma is in collection mode (line 31), the Collector wraps the actual parameter values for the interpretation function using `Param` objects, and stores them, together with a pointer to their respective instruction implementation and the emulated `%pc` of the interpreted instruction, in an `InstrApp` object (lines 33-34). It then appends the instruction application, and therewith the complete semantic information for the interpreted instruction, to the trace of collected semantic information (line 35).

The interpretation of a CTI may initiate or terminate the collection of semantic information and, upon termination, may trigger the generation of a dynamic function through the Emitter. Therefore, most interpreted CTIs require additional instrumentation, as shown in listing 4.2. If the Collector is in collection mode (line 23), the fixed addresses of both out edges of the currently collected trace are stored as additional trace information (line 30). This information enables the linking of dynamic functions upon their storage in the code cache. The actual generation of a dynamic function is explicitly triggered by calling the `Collector::flush()` method in line 31. Note that the generation may need to be delayed by one instruction if, as shown in our example, the source architecture implements CTIs with delayed semantics. Finally, in order to switch the collect mode on or off, the `Collector::switchMode()` method in line 36 passes relevant control flow information to the Collector. Note that CTIs without a fixed target address always switch *off* collection mode, whereas CTIs with a fixed target address actually control the collection of the semantic information.

When the complete semantic information about a hot trace is collected, the trace is flushed, thus triggering the generation of the dynamic function.

```

1  ctitable1 [ "BNE", "BE", "BG", "BLE", "BGE", "BL", "BGU",
2             ...
3             ]
4
5  ctitable1 reloc
6  {
7      // the PC of the emulated instruction
8      sint32_t _pc = SSL(%pc);
9
10     // force actual parameter value into host registers
11     register sint32_t reg_reloc asm (REG1) = PARAM(reloc);
12     Lstart:
13     {
14         sint32_t PARAM(reloc) = reg_reloc;
15
16         // actual instruction implementation
17         SSL_INST_SEMANTICS
18     }
19     Lend:
20
21     // collect implementation and application of the
22     // emulated instruction for the current trace
23     if (collector.collect()) {
24         static const InstrImpl ii(&&Lstart, &&Lend, true);
25         InstrApp ia = new InstrApp(_pc, &ii, Param(PARAM(reloc)));
26         collector.addInstrApp(ia);
27
28         // set the static out edges for this trace, and trigger
29         // the delayed generation of the dynamic function
30         collector.setOutedges(PARAM(reloc), _pc+8);
31         collector.flush(true);
32     }
33
34     // pass control flow information to the collector, which
35     // may or may not switch the collection mode (true=delayed)
36     collector.switchMode(_pc, SSL(%npc), true);
37 }

```

Listing 4.2:

CTIs are instrumented like any other source machine instruction, but in addition control Yirr-Ma's Collector. The interpretation of most CTIs adds control-flow information to the trace, triggers the generation of a dynamic function, and initiates or terminates the collection of semantic information by switching Yirr-Ma's collection mode on or off.

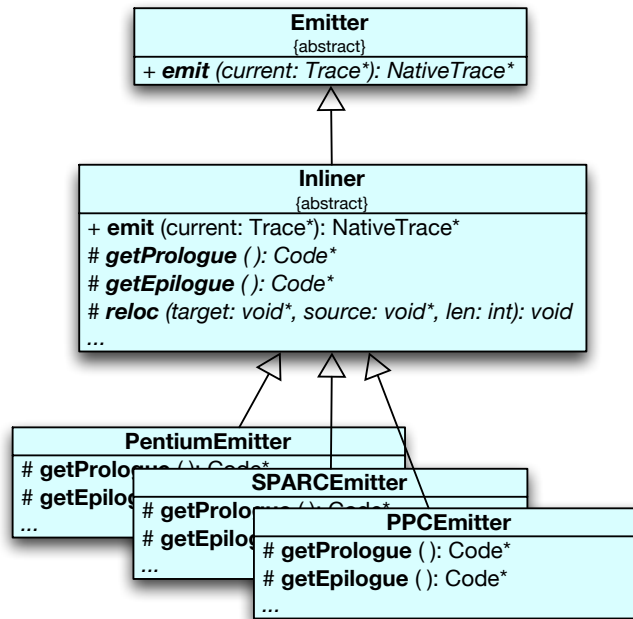


Figure 4.1:

Class diagram of the `Emitter` interface and implementing sub-classes for Yirr-Ma's dynamic partial Inliner. The `Emitter` defines the interface for Yirr-Ma's code generators that generate a dynamic function from a given trace. The `Inliner` class implements the actual generation of code by its `Inliner::emit()` method which, in turn, uses `Code` objects created by host-specific sub-classes.

4.4 Generation of Dynamic Functions

Similar to a statically compiled function, so contains the dynamic function a prologue that allocates a stack frame for local variables, an epilogue to remove that stack frame and return, and the function's body. Yirr-Ma's `Inliner` abstracts the host specific implementation of function prologues and epilogues into `Code` objects: a black box of host machine instructions that can be freely relocated. The class diagram in figure 4.1 illustrates the hierarchy of Yirr-Ma's dynamic partial Inliner, as well as its super- and sub-classes. Because `Code` objects are host machine dependent, they are created by a host machine specific sub-class of the `Inliner`, but they are then handled by the `Inliner` itself. Analogous to the `Code` objects wrapping prologue and epilogue, the operational semantics of interpreted instructions, enclosed in first-class labels above, is wrapped into `Code` objects for relocation.

The generation of a dynamic function is implemented by the `Inliner::emit()` method and works as follows. First, the `Inliner` allocates memory for the dynamic function. Second, it relocates `Code` objects into the allocated memory, incrementally constructing the dynamic function. The `Code` objects, however, are created by host-specific sub-classes

of the Inliner. The order in which `Code` objects are relocated into a dynamic function is as follows:

1. copy the content of the `Code` object implementing the function prologue into the body of the dynamic function
2. for every interpreted source machine instruction of the collected trace
 - 2.1. generate and copy an instruction prologue that passes the actual parameter values (from the instruction application) of the interpretation function into dedicated host machine registers
 - 2.2. relocate the instruction implementation from its original location inside of the interpretation function
3. optionally copy placeholder instructions of dynamic linking (cf. section 3.6)
4. copy the content of the `Code` object implementing the function epilogue.

The content of a given `Code` object is copied without additional relocation using the host machine's `memcpy` function. However, because the interpretation of some source machine instructions requires support from the Walkabout emulator core (e.g. the interpretation of `trap` instructions calls Walkabout's system call mapper) or from external library functions (e.g. the interpretation of SPARC instructions for 64-bit integer division on Pentium hosts invoke `libc` functions), these `Code` objects must be relocated properly. The relocation process is transparent to the Inliner and is implemented by the host specific sub-classes. When an instruction implementation is copied into the body of a dynamic function (see step 2.2 above), the Inliner's sub-class checks the relocation flag of the `InstrImpl` object and either simply `memcpy`'s, or actually relocates, the host machine instructions accordingly.

Listing 4.3 shows the dynamic function on a Pentium host machine that was generated from the SPARC example trace in listing 3.1. Lines 1 and 20-21 represent the prologue and epilogue of the dynamic function, respectively. The body of the dynamic function contains the implementations of the four source machine instructions that were inlined from their respective interpretation functions. For example, because the interpretation function of the `LDreg` instruction takes three actual parameter values (`LDreg` defines three operands), the three actual parameter values are stored into host machine registers by the instruction prologue in lines 2-4. Lines 5-7 represent the first three lines of the instrumentation (lines 19-21 in listing 4.1), where the actual parameter values of the interpretation function are copied from forced host machine registers into local variables. The dots in line 8 represent the actual compiled implementation of the source machine instruction that was inlined and relocated into the body of the dynamic function. The remaining three source machine instructions of the trace are inlined in the same manner.

```

1 0x080a7c08: enter  $0x40,$0x0 ; standard function prologue
2 0x080a7c0c: mov    $0x11,%ecx ; LDreg operands/instr prologue
3 0x080a7c11: mov    $0x0,%ebx
4 0x080a7c15: mov    $0x10,%eax
5 0x080a7c1b: mov    %al,-17(%ebp) ; start of LDreg semantics
6 0x080a7c1e: mov    %bl,-18(%ebp)
7 0x080a7c21: mov    %cl,-19(%ebp)
8 ... ; inlined interpreted function
9 0x080a7c7f: mov    $0x11,%eax ; operand of TST
10 0x080a7c84: mov    %al,-5(%ebp) ; start of TST semantics
11 ...
12 0x080a7ce9: mov    $0xfe02a24,%eax ; operand of BNE
13 0x080a7cee: mov    %eax,-4(%ebp) ; start of BNE semantics
14 ...
15 0x080a7d41: mov    $0x10,%ebx ; operands of INC
16 0x080a7d46: mov    $0x4,%eax
17 0x080a7d4b: mov    %eax,-4(%ebp) ; start of INC semantics
18 0x080a7d4e: mov    %bl,-8(%ebp)
19 ...
20 0x080a7db9: leave ; standard function epilogue
21 0x080a7dba: ret

```

Listing 4.3:

Generated dynamic function for the SPARC example trace on a Pentium host machine. The function consists of a prologue, partially inlined interpretation functions and the function epilogue.

4.5 Implementation

Yirr-Ma's Emitter defines an interface for the differing implementations of dynamic optimizers. The `Inliner` sub-class implements the host machine independent methods for the actual generation of dynamic functions, and it defines the interface for the host machine dependent sub-classes (cf. the class diagram in figure 4.1). Listing 4.4 shows the implementation of the `Inliner::emit()` method that generates a dynamic function from a given trace.

First, the `Inliner` allocates memory for the new dynamic function (line 4). It then copies the function's prologue into the allocated memory (line 7) and then iterates over the collected semantic information of the trace. For every collected instruction, the `Inliner` generates an instruction prologue which is copied into the dynamic function (line 12), and then inlines the significant part of the instruction's interpretation function into the body of the dynamic function (line 13). At last, it copies optional placeholder instructions for dynamic linking (line 17) and the function's epilogue (line 20). The generated dynamic function is then returned to the `Collector`.

In order to port Yirr-Ma's `Inliner` to a new host machine, a host machine specific sub-class of the `Inliner` interface must be implemented. The new class must provide implementations for the pure virtual methods that the `Inliner` interface defines and invokes. These methods are:

```

1 NativeTrace* Inliner::emit(Trace* t)
2 {
3     // allocate a new dynamic function
4     _trace = new NativeTrace(t, this);
5
6     // emit the function prologue
7     emitPrologue( getPrologue() );
8
9     // copy the host code into the dynamic function
10    Trace::iterator it;
11    for (it = t->begin(); it != t->end(); it++) {
12        emitInstrPrologue(*it);
13        emitInstr(*it);
14    }
15
16    // emit local linking, or placeholder code
17    emitLinks();
18
19    // emit the function epilogue
20    emitEpilogue();
21
22    return _trace;
23 }

```

Listing 4.4:

The `Inliner::emit()` method allocates a new dynamic function object, emits the prologue, the collected semantic information, placeholder instructions and the epilogue. Then it returns the generated dynamic function to the Collector.

- the `getPrologue()` and `getEpilogue()` methods each return `Code` objects that contain a host machine specific function prologue/epilogue with a fix-sized local stack frame
- the `getInstrPrologue()` method takes an `InstrApp` object and produces a `Code` object which writes the actual parameter values of this instruction application into host machine registers
- the `getLinks()` method returns a `Code` object that contains either a valid local link or placeholder instructions for subsequent linking
- the methods `linkAsTaken()` and `linkAsFallthrough()` implement the actual linking of two dynamic functions
- the `relocCode()` method `memcpy`'s and, if required, relocates a block of host machine instructions from one address to another

The implementation of these methods requires only a small amount of knowledge about the new host machine. For instance, it took less than three man-days to implement a complete PowerPC sub-class for the Inliner, including the time spent learning about the PowerPC ISA.

Relocation of Host Machine Instructions

The relocation of host machine instructions is transparent to the Inliner, and a host-specific sub-class must implement the virtual `Inliner::relocCode()` method. Relocation of a `Code` object is necessary because the implementation of some interpretation functions contains `call` instructions. The `%pc`-relative target address of these `call` instructions must be adjusted to the instruction's new location in the dynamic function. In order to locate a `call` instruction, `Inliner::relocCode()` scans the copied instructions, re-computes the distance to the target address and then adjusts the operand of the `call` instruction.

Whether or not the implementation of a source machine instruction requires proper relocation is defined in Yirr-Ma's instrumentation file for the emulated machine. We found that the instrumentation of an emulator needs to be fine-tuned for every host machine. For example, the interpretation of the SPARC `SDIV` and `UDIV` instructions is implemented by a library call on PowerPC and Pentium host machines and, therefore, requires proper relocation. In contrast, SPARC host machines do not require a library function call and, therefore, no relocation of `Code` objects for these two instruction implementations is needed.

4.6 Results

We tested Yirr-Ma's Inliner with some SpecINT'95 benchmark programs that were compiled for SPARC v8/Solaris source machines: `compress`, `li`, `go` and `perl` running the "train" input sets. The three different host machines are standard desktop computers: a SPARC v9/Solaris, an Intel Pentium/Linux and a PowerPC G4/Linux box.

Table 4.1 compares the relative execution times of the four benchmark programs interpreted by an uninstrumented Walkabout emulator with a Walkabout emulator with Yirr-Ma's Inliner. Yirr-Ma collected its traces using a hot-threshold of 23, and used full-linking of dynamic functions for an unlimited code cache. The performance improvements or penalties depend on the actual interpreted program, or more precisely, its runtime behavior. Whereas `compress` with its highly repetitive runtime behavior amortizes the cost of Yirr-Ma's overhead, `go` performs worse under our dynamic optimization. This is because of `go`'s constantly changing working set and Yirr-Ma's overhead that is required to manage the ever growing number of dynamic functions and hot-spots for the unlimited code cache.

Table 4.2 lists the performance results of Yirr-Ma for different linking techniques used for an unlimited cache. As can be seen locality properties of the interpreted traces reflect in performance of the interpreted program. Whereas the overhead of *full linking* is amortized for `compress` and `perl`, its overhead almost outweighs *no linking* for `go`.

	SPARC	PowerPC	Pentium
	compress.v8 < test.in		
Walkabout	0m 20.3s	0m 12.0s	0m 9.7s
Walkabout/Yirr-Ma	0m 5.0s	0m 2.8s	0m 2.2s
	li.v8 train.lsp		
Walkabout	10m 25.0s	6m 14.4s	5m 0.1s
Walkabout/Yirr-Ma	9m 46.9s	5m 53.1s	4m 13.8s
	go.v8 50 9 2stone9.in		
Walkabout	4m 23.8s	2m 32.2s	2m 6.34s
Walkabout/Yirr-Ma	4m 33.0s	4m 20.3s	2m 41.1s
	perl.v8 jumble.pl < jumble.in		
Walkabout	18m 39.0s	12m 40.1s	10m 0.1s
Walkabout/Yirr-Ma	17m 23.6s	10m 35.2s	8m 26.4s

Table 4.1:

Performance results for some SpecINT'95 benchmark programs compiled for SPARC/Solaris and running on SPARC, PowerPC and Pentium host machines. The table shows the runtimes of the benchmark programs running on an uninstrumented Walkabout emulator and the Walkabout/Yirr-Ma dynamic Inliner.

	full linking	no linking	local linking	demand linking
compress	0m 2.8s	0m 3.1s	0m 4.6s	0m 3.1s
li	6m 14.4s	6m 44.4s	7m 20.9s	6m 39.7s
go	4m 20.3s	4m 22.8s	3m 23.8s	4m 23.4s
perl	12m 40.1s	13m 44.8s	14m 14.1s	13m 33.2s

Table 4.2:

Walkabout/Yirr-Ma interprets some SpecINT'95 benchmark programs using an unlimited code cache. The performance results are for different linking techniques, and they also provide insights into the code locality of detected hot-spots.

	M_S	SPARC	PowerPC	Pentium
compress.v8 < test.in				
Cached traces		431	433	433
Cache size in bytes		851.628	887.936	425.707
Shortest trace	1	47	62	29
Average trace	6	513	494	287
Longest trace	51	3694	3686	2488
li.v8 train.lsp				
Cached traces		882	881	881
Cache size in bytes		1.745.468	1.762.480	867.363
Shortest trace	1	47	62	22
Average trace	5	459	501	280
Longest trace	38	3131	2772	1753
go.v8 50 9 2stone9.in				
Cached traces		5368	5368	5368
Cache size in bytes		11.990.464	12.720.264	6.113.771
Shortest trace	1	47	62	22
Average trace	7	559	593	319
Longest trace	63	4015	4278	2143
perl.v8 jumble.pl < jumble.in				
Cached traces		1025	1023	1021
Cache size in bytes		1.687.104	1.748.748	843.725
Shortest trace	1	47	62	22
Average trace	4	412	428	237
Longest trace	38	2726	2722	1753

Table 4.3:

Actual code cache sizes and trace lengths of Yirr-Ma running some benchmark programs. The first two rows give information about the number of cached traces, and how much memory they consume. The three bottom rows show statistics about the size of traces by means of interpreted source machine instructions and generated host machine instructions.

It is noteworthy that *local linking* outperforms any other linking technique for the go benchmark program. This indicates that most hot traces are closed, i.e. their taken branch is the beginning of the trace itself.

Table 4.3 gives statistical information about an unlimited code cache at runtime. It shows total code cache sizes in bytes, and compares the number of instructions for interpreted hot traces to the number of instructions for their generated dynamic functions. The first two rows give, for every benchmark program, the number of generated dynamic functions and the total size of the code cache in bytes. For example, for the li benchmark program Yirr-Ma's Inliner generated 881 dynamic functions which consumed a total of 1.68 MByte on PowerPC or 847 kByte on Pentium host machines. The other three rows give, again for every interpreted program, the ratio of the number of source

machine instructions to the number of generated host machine instructions for differing traces. It shows the number of instructions for the shortest, the arithmetical average and the longest interpreted hot trace and for the generated dynamic functions. On average, for instance, an interpreted hot trace of the `perl` benchmark is 4 SPARC machine instructions long, which then results in dynamic functions of an average of 412 SPARC instructions, 428 PowerPC instructions and 237 Pentium instructions. The table is also an illustration of the different instruction densities of RISC and CISC ISAs.

In summary and conclusion, Yirr-Ma's Inliner improves the performance of a Walkabout emulator for most interpreted programs on all tested host machines. However, owing to Yirr-Ma's lazy code cache and hot-spot management, the added overhead is too big for source programs that have changing working sets, thus slowing down the performance of the emulator itself.

4.7 Summary, Contribution and Outlook

In this chapter we introduced one implementation of Yirr-Ma's Emitter interface: a dynamic binary translator that employs partial inlining. Dynamic functions are generated at runtime by inlining selected parts of instrumented interpretation functions, thus harvesting the translation effort of a static compiler and translating source machine instructions to the host machine at no cost. The Inliner itself defines an interface which, in order to port the Inliner to new host machines, must be implemented by hand. Performance tests expectedly showed a general speed up of Walkabout/Yirr-Ma compared to the plain interpretation by an uninstrumented Walkabout emulator.

The work in this chapter demonstrates Yirr-Ma's contribution as a machine independent and flexible extension for generated Walkabout emulators. It is also a case study of retargetable dynamic binary translation and of a portable dynamic optimization framework and its components. The experiences that we gathered with generated Walkabout machine emulators and with Yirr-Ma made way for our specification-driven dynamic binary translator. This dynamic binary translator again implements Yirr-Ma's Emitter interface, and is the major aspect and contribution of this thesis. We introduce its approach, techniques and implementation in much detail in the following chapter.

5 Specification-Driven Dynamic Binary Translation

What upsets us most is the difference between imagination and reality.

(Paul Gleeson)

5.1 Introduction

We introduced the architecture of the dynamic optimization framework Yirr-Ma in chapter 3, and detailed the functioning, design and implementation of its dynamic partial Inliner in chapter 4. Yirr-Ma implements the host machine instructions, generated from a given source machine trace t , in the form of a dynamic function. Dynamic functions contain an optimized version of t which is then invoked in preference to the instruction-wise interpretation of t . In the context of our formal framework from chapter 1, Yirr-Ma's Inliner, for example, implements the trace optimization function $\omega_{I_H}^*$ that improves the interpretation of a trace of source machine instructions.

We introduce another implementation of Yirr-Ma's Emitter interface in this chapter: a *specification-driven dynamic binary translator*. The techniques and algorithms presented are the main contribution of this thesis. In contrast to other existing dynamic binary translators, our translator is portable to different source and host machines *entirely* through machine specification files: machine specific components of both frontend and backend are generated from machine specification files that define relevant properties of the respective machine. The translator is invoked from Yirr-Ma and works as follows. First, it constructs an intermediate representation from a given trace of semantic information using the operational semantic specification of the source machine. Then, in a second step, the translator maps relevant parts of the emulated source machine state of the intermediate representation to the host machine, thus implementing an optimized state mapping $\omega_{S_H}(\varphi_S(s^S))$. The implementation of the state mapper is generic and parameterized with the host machine's state specification. At last, the φ_S -mapped and ω_{S_H} -optimized intermediate representation is further optimized and then compiled into a dynamic function for the host machine. Unlike the parameterized state mapper, the applied dynamic optimizations are machine independent. The instruction selector is generated from the operational semantics specification for the host machine,

and the instruction encoder is generated from a SLED specification using the NJMC Toolkit. This specification-driven generation of dynamic functions implements the parameterized translation function $\tau^{\omega_{SH}}(t^S)$, introduced with equation 2.2 in section 2.4. Note that we do not address combinations of different optimization functions yet.

A vital implication of equation 2.2 is the definition of an upper bound for the quality of dynamically translated machine instructions. The first part of this chapter therefore focuses on the derivation of this upper bound, we illustrate it with comprehensive examples, and we discuss how the quality can be improved, thus enhancing the overall performance of the machine emulator. The major part of this chapter is dedicated to the in depth discussion of our specification-driven dynamic binary translator. We discuss the properties of the specification files, and we show how they are used for dynamic binary translation. We also demonstrate the practical applicability of our approach and illustrate every step of a translation with a number of examples. Some initial toy benchmark tests finally conclude this chapter.

5.2 The Upper Bound for the Quality of Translated Machine Instructions

A dynamic binary translator, like any other dynamic optimizer, is constrained by both time and memory resources. It is hardly possible to apply the same sophisticated optimizations to the intermediate representation and the generated code that a static compiler can resort to. However, because the context of a dynamic binary translator is different to that of a static compiler, it may exploit the available runtime information to improve the performance of the source program interpretation.

The quality of a dynamic binary translator is often measured by the source to host machine instruction ratio, i.e. how many host machine instructions, on average, does a single source machine instruction translate into. But the matter in reality is more complicated, though not simply because of the diversity of processor architectures.

To understand and measure the quality of dynamically translated machine instructions, we investigated the relationship between

- the implementation of an emulated source machine on a host machine
- the mapping of the source machine state to the host machine state during a translation
- the generated host machine instructions.

In order to investigate these relationships, we performed an experiment that supports and illustrates our reasoning.

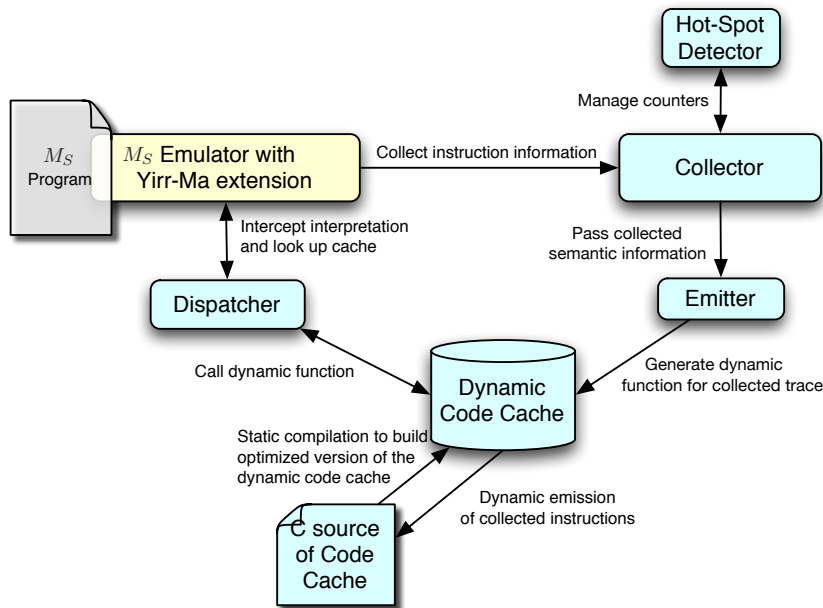


Figure 5.1:

Yirr-Ma and its extended, statically compiled code cache. Upon its destruction, the code cache saves its content into a single C/C++ function that is compiled by a common static compiler. During subsequent runs of the same program, the precompiled cache content is loaded into the code cache, thus improving the performance of the emulator.

5.2.1 An Experiment

We used the static compiler `gcc` to examine and define the upper bound, therewith ignoring the above mentioned runtime constraints of a dynamic compiler. The static compiler translates the same information that is available to a dynamic binary translator. By applying sophisticated compilation/translation and optimization techniques in this “pseudo-dynamic” environment we gather insights into the theoretically achievable upper bound of the code quality of translated machine instructions.

The implementation for our experiment extends Yirr-Ma’s dynamic code cache such that the code cache writes the semantic information of all collected traces into a C/C++ source code file when Yirr-Ma terminates. The class diagram in figure 3.5 documents the extended code cache, and figure 5.1 illustrates the extended Yirr-Ma framework. The generated source code file contains the same information that is available to the dynamic binary translator at runtime. This information is sufficient to generate host machine instructions, both statically and at runtime, under similar conditions. Through the static compilation of the collected traces, the C/C++ compiler implements both the optimized state mapping $\omega_{S_H}(\varphi_S(t^S))$ and the parameterized translation function $\tau^{\omega_{S_H}}$.

Yirr-Ma, using the extended code cache, works now as follows. In an initial run, the

code cache is populated with hot traces as usual: when dynamic functions are inserted into the code cache, they are fully linked with one another, thus creating linked traces of hot instructions¹. Upon destruction of the code cache, the cache generates a single C/C++ function that implements the entirety of the cache's contents. The prototype of this cache function (for an emulated SPARC source machine) is declared as:

```
bool invokeFunction(sint32_t addr, // current %pc
                   char* mem, // base address of the emulated memory
                   RegisterFile& regs, // register file
                   float32_t* regs_fds[], // shared float registers
                   float64_t* regs_fdd[],
                   float128_t* regs_fdq[]);
```

The cache function takes six parameters: the value of the emulated %pc register, usually the target of a CTI, a pointer to the emulated memory segment, a reference to the emulated integer register file, and pointers to the floating point register files overlapping one another (cf. SPARC's SSL state specification). The body of the cache function maps the given %pc value to a label which is the beginning of the statically compiled source machine instructions, therewith implementing a performant cache lookup. Following each label, the semantic information of the related dynamic function is emitted as a sequence of macros. For instance, the example trace from listing 3.1 produces the following C/C++ source code:

```
_lbl_fe02a24: // label = start of trace
  LDREG(16,0,17); // macros that implement the source machine
  TST(17); // instructions of the hot trace
  BNE(0xfe02a24);
  INC(4,16);
  if (regs.r_COND) // implement interpreted control-flow
    goto _lbl_fe02a24; // taken link
  else
    goto _lbl_end; // fall-through link (exit the cache)
```

A macro implements a source machine instruction of the collected hot trace, and the instruction's operands, that is to say the actual parameter values to the interpretation function, are the macro's parameters. Links between dynamic functions are emitted as goto statements that branch to the label of the target sequence of macros. For example, as can be seen in table 4.3, Yirr-Ma collects 431 dynamic functions for the compress benchmark program. The generated cache function therefore contains 431 macro sequences and 2×431 goto statements branching either to logically adjacent sequences or that exit the code cache.

The implementation of the macros themselves is generated from the SSL specifications, together with the rest of the source machine specific part of Walkabout's emulator core. For example, the LDREG macro generated from the specification of SPARC's LD instruction

¹ Note the similarities to the FX!32 x86 emulator and static binary translator (cf. section 1.3). It statically translates x86 "profiles", collected at runtime, into Alpha host machine instructions.

	compress.v8	li.v8	go.v8	perl.v8
C/C++ file (macros) in bytes	136.454	267.719	1.809.798	284.223
Expanded C/C++ file	916.993	1.924.886	12.127.205	1.827.964
gcc compile time with -O2	2m 9.0s	7m 51.3s	n/a	8m 14.0s
Compiled cache size in bytes	122.924	236.012	n/a	245.852
Dynamic cache size in bytes	851.628	1.745.468	11.990.464	1.687.104
Interpretation	0m 20.3s	10m 25s	4m 23.8s	18m 39.0s
Yirr-Ma Inliner	0m 5.0s	9m 46.9s	4m 33.0s	17m 23.6s
Compiled code cache	0m 3.1s	6m 28.3s	n/a	10m 52.7s

Table 5.1:

File sizes and compilation times for the generated C/C++ cache functions for different benchmark programs. The precompiled code cache increases the performance of the emulator for a particular program, compared to plain interpretation and our dynamic partial Inliner.

(cf. listing 1.1) looks as follows:

```
#define LDREG(op_rs1,op_rs2,op_rd) { \
    regs.rd[0] = 0;\
    regs.rd[op_rd] = \
        getMemsint32_t(regs.rd[op_rs1] + regs.rd[op_rs2]);\
    regs.r_pc = regs.r_npc;\
    regs.r_npc = (regs.r_npc + 4);\
}
```

The generated C/C++ source code file implements the content of the code cache, and thus linked traces of source machine instructions. These traces operate over the emulated source machine state implemented by the emulator.

The cache content is then compiled into a highly optimized dynamic linker library. During subsequent runs, Yirr-Ma's modified code cache loads this library and populates the cache with hot traces that are optimized by the static compiler. As usual, Yirr-Ma uses its Dispatcher to invoke the hot traces, thus saving Yirr-Ma the dynamic translation overhead.

Table 5.1 shows the sizes of the C/C++ source code files that are generated for some SpecINT'95 benchmark programs, their compile time and cache sizes, and it compares execution times of the different machine emulators. All benchmark tests were performed on the same SPARC v8 host machine using the same static compiler. The first two rows show the source code sizes of the generated cache function before and after the macro expansion. Row three shows the static compile time for the cache functions using the compiler's -O2 optimization switch, and rows four and five show the sizes of the compiled cache function compared to the size of the code cache with dynamically collected traces, respectively. As can be seen is the size of the compiled cache function significant smaller, when compared to the original cache size. Notably, due to its complexity, we were not

able to compile the generated cache function for the `go` benchmark program². The last three rows compare the execution times of the benchmark programs using plain interpretation by the Walkabout emulator, interpretation combined with Yirr-Ma's Inliner, and interpretation combined with the statically compiled code cache. As expected the invocation of the statically compiled code cache outperforms both plain interpretation and our dynamic Inliner, sometimes drastically.

5.2.2 Examination of the Generated Code

The input for both the static C/C++ compiler and the dynamic binary translator is the same: traces of semantic information collected at runtime. However, in contrast to the dynamic binary translator, the static compiler is not constrained by time or memory resources, and thus it is free to apply sophisticated optimizations during compilation. Listing 5.1 shows the SPARC host instructions that were generated for our example source trace in listing 3.1. Note that this particular example illustrates a SPARC to SPARC translation rather than virtualization. As a further illustration, appendix A.1 shows the same SPARC example trace compiled for Pentium and PowerPC host machines, respectively.

The generated SPARC trace consists of 35 instructions, 21 of which are required to load and store values from and to the emulated register file and the emulated memory segment. The remaining 14 instructions implement the operational semantics of the interpreted source machine instructions:

- the `ld` instruction in line 5 implements the interpreted `ld` instruction, including its redirection to the emulated memory segment
- the instructions in lines 7, 8, 10 and 11 implement SPARC's synthesized `TST` instruction, including the explicit computation of the emulated machine flags which are stored to individual host machine registers
- the instructions in lines 14 and 21-28 implement the conditional branch instruction `bne`, i.e. they modify the emulated `%pc` and `%npc` registers according to the values of emulated flags
- the `add` in line 32 implements the synthesized `INC` instruction.

Of these 14 instructions, only the 2 instructions in lines 9 and 31 implement secondary effects that advance the emulated `%pc` and `%npc` registers. Note, however, that the

² After almost 4 hours, `gcc` crashed with a segmentation fault. It had already allocated more than 1 GByte of memory, and the allocation of another 2 GByte failed. Further attempts to compile the file on bigger machines with other versions of `gcc` were fruitless and we filed the bug for `gcc` under [ID 13716](#). The bug was acknowledged and attributed "memory hog".

```

1      ; ptr to emulated register file is in %g4
2  ld    [%fp+72], %l0 ; emulated membase to %l0
3  st    %g0, [%g4]   ; store 0 to %g0S
4  ld    [%g4+64], %o0 ; %l0S ↦ %o0
5  ld    [%l0+%o0], %o1 ; %l1S ↦ %o1, %o1 := [membase + %l0S]
6  ld    [%g4+304], %o5 ; %npcS ↦ %o5, %o5 := 0xfe02a28
7  subcc %g0, %o1, %g0 ; %g0-%o1 sets/clears %CF
8  subx  %g0, -1, %o2 ; %ZFS ↦ %o2, %o2 := %l1S=0 ? 1 : 0
9  add   %o5, 8, %o3  ; %npcS ↦ %o3, %o3 := %npcS + 8
10 xor   %o2, 1, %o4  ; %CONDS ↦ %o4, %o4 := ~%ZFS
11 srl  %o1, 31, %o0  ; %NFS ↦ %o0, %o0 := %l1S@[31:31]
12 stb  %o0, [%g4+280] ; store %NFS into register file
13 st   %o3, [%g4+308] ; store %pcS
14 cmp  %o4, 0       ; compare %o4 (%CONDS) register with 0
15 st   %o1, [%g4+68] ; store %l1S
16 st   %o1, [%g4]   ; store %l1S to %g0
17 st   %o3, [%g4+304] ; store %npcS
18 stb  %o2, [%g4+300] ; store %ZFS
19 stb  %g0, [%g4+281] ; store (clear) %DFS
20 stb  %g0, [%g4+257] ; store (clear) %CFS
21 bne  .LL2831
22 stb  %o4, [%g4+258] ; delay slot! store %CONDS
23 b    .LL2832
24 add  %o5, 12, %o3  ; delay slot! %npcS ↦ %o3
25      ; %o3 set to 0xfe02a34
26 .LL2831:
27 sethi %hi(266348544), %o0
28 or    %o0, 548, %o3 ; %npcS ↦ %o3, %o3 set to 0xfe02a24
29 .LL2832:
30 ld    [%g4+64], %o1 ; %l0S ↦ %o1
31 add   %o3, 4, %o2   ; %npcS ↦ %o2
32 add   %o1, 4, %o1   ; increment %o1 (%l0S) by 4
33 mov   1, %o0       ; %CTIS ↦ %o0
34 stb  %o0, [%g4+259] ; store %CTIS
35 st   %o1, [%g4+64] ; store %l0S
36 st   %o2, [%g4+304] ; store %npcS
37 stb  %g0, [%g4+264] ; store (clear) %DLYAS
38 st   %g0, [%g4]   ; store (clear) %g0S
39 st   %o3, [%g4+308] ; store %pcS

```

Listing 5.1:

The static compilation of a SPARC source machine trace leads to much improved quality of the generated host machine trace. The static compiler is not limited by runtime constraints, and thus employs sophisticated optimization techniques. Note that $\%r_S \mapsto \%r'$ denotes the register mapping from an emulated SPARC register $\%r_S$ to an actual SPARC host register $\%r'$.

instructions in lines 9, 13 and 17 are dead and could be removed. In fact, the 4 SPARC instructions of the interpreted hot trace are effectively translated into 12 host machine instructions, which is an average ratio of 1:3 without overhead for this particular example trace. The remaining 23 instructions implement emulated secondary effects that also account for the implementation of the emulated source machine state. The actual source to host instruction ratio is roughly 1:9. We show in the following that this ratio is close to the upper bound of the quality for the translation in the given context.

The static compiler handles the emulated memory segment and the emulated register files as global and mutable data, and it applies its optimizations accordingly. However, the example trace, and hence every dynamically translated source machine trace, holds more potential for optimization if information about certain source machine properties and their implementation by the emulator are made available to either of the compilers. We elaborate on those in section 5.9.

5.2.3 On the Quality of Dynamically Translated Machine Instructions

Equation 2.2 on page 37 describes dynamic binary translation as a parameterized translation function $\tau^{\omega_{S_H}}$ that maps a trace t^S of interpreted source machine instructions to a trace of host machine instructions in the context of an optimized state mapping. The host machine instructions are then interpreted by the host machine and advance the implementation of the emulated source machine state consistently:

$$\varphi_S(\gamma_S^*(t^S, s^S)) = \gamma_H^*(\tau^{\omega_{S_H}}(t^S), \omega_{S_H}(\varphi_S(s^S))).$$

The equation is applied in practice in section 3.5, where we demonstrated that the invocation of a dynamic function may replace the interpretation of individual source machine instructions, provided that the emulated machine state is maintained consistently.

The *static* compilation of source machine traces illustrates the implication of the above equation on the emulated state: the translation of a source machine trace t^S is always parameterized, and therefore constrained, by the state mapping. More precisely, the quality of (statically or dynamically) translated machine instructions depends on

- the quality of φ_S , i.e. the quality of the implementation of the source machine state
- the mapping/optimization ω_{S_H} of the emulated machine state to the host machine for a particular trace t^S
- the instruction translation, improved under the optimized state mapping.

In fact, the upper bound of the quality of dynamically translated machine instructions is *primarily constrained* by the implementation of an emulated source machine state,

and only secondary by the applied translation and optimization techniques.

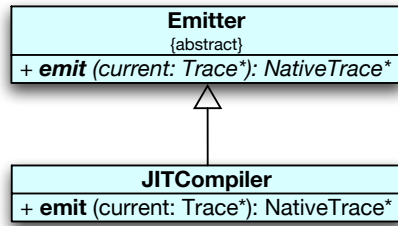
When a source machine instruction is translated into a (sequence of) host machine instruction(s), the operational semantics of the translated source machine instruction must be preserved. The operational semantics of an instruction is always defined over a machine state, and the translation of the source machine instruction must therefore operate on an implementation or mapping of the source machine state. The quality of this state mapping from source to host machine constrains the quality of the translated host machine instructions: both the implementation of the emulated machine state by the emulator, and the machine characteristics of the source and host machines are factors defining this constraint. The above experiment is an illustration of how sophisticated compilation techniques improve the quality of the translated state and instructions, and thus lead to a much improved performance of the machine emulator. However, in contrast to the static compilation, the dynamic translation of machine instructions is subject to time and memory constraints that limit the applicable optimization techniques.

The state mapping from the emulated source machine to the host machine becomes more difficult if both machines have very different characteristics. Whereas, say, a SPARC to SPARC translation is trivial and can lead to very performant host machine code, the translation of CISC machine instructions to a RISC host machine (or vice versa) proves to be more difficult. This is because of the different register files and the semantics of the individual registers and flags, and/or the representation of data (big and little endian).

Two Examples

A Walkabout emulator implements the state of the emulated machine by means of a register file, a compound data structure stored in the host machine's memory, and a relocated memory segment. Both implementations add a level of indirection to the interpretation of source machine instructions: emulated register operands require instructions that load the operands from the emulated register file into host machine registers, and translated load and store instructions must be diverted to the emulated memory segment by adding an offset to every memory access. The translated source machine instructions therefore require additional instructions that account for these indirections. In the above example trace, the emulated source machine state accounts for 21 instructions, which is an added overhead of approximately 290%.

The complete contrast to a Walkabout emulator are Transmeta's Crusoe and Efficion processors. They translate only x86 instructions into a custom VLIW format. The state of the "emulated machine" here *is* the x86 host machine state and thus the state mapping φ_S is an identity function. The added overhead to handle the source machine state is therefore 0 which leads to generated host machine instructions without additional instructions implementing indirections introduced by the emulated machine state.

**Figure 5.2:**

Class diagram of Yirr-Ma’s Emitter interface and our specification-driven dynamic binary translator which provides an implementation for the interface.

5.2.4 Improving the Quality

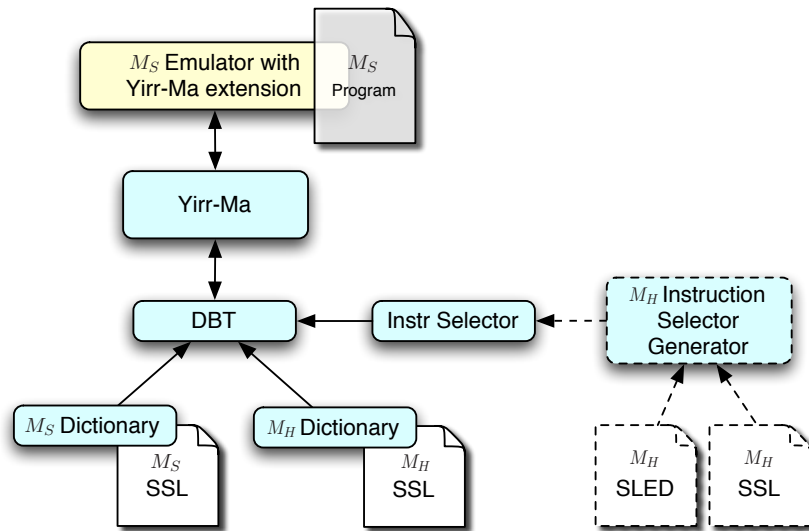
Improving the quality of generated host machine traces requires not only the efficient translation and optimization of the source machine instructions, but also requires consideration of the fact that the implementation of the emulated machine state on the host machine constrains the quality of the instruction translation. In particular for specification-driven machine emulation and dynamic binary translation both the quality (i.e. granularity) of the machine specifications *and* the quality of the generators define the upper bound of the quality of the translated machine instructions.

Put into the context of our formal framework, the optimizing state translation function $\omega_{S_H}(\varphi_S(t^S))$ must be optimized towards the identity function. The overhead which parameterizes the instruction translation function $\tau^{\omega_{S_H}}$ is then ideally 0, allowing an instruction mapping without the compensation for the source machine state implementation. Note that if $\omega_{S_H}(\varphi_S(t^S))$ is the identity function and source and host machine are equal, i.e. if $(S_S, I_S, \gamma_S) = (S_H, I_H, \gamma_H)$ holds, the source program is in its native environment. On a side note, virtualization is an interesting case where the instruction mapping φ_I is the identity function, and overhead is introduced by the state implementation φ_S (the context switch) to maintain state consistency for the emulated machine.

5.3 Yirr-Ma’s Dynamic Binary Translator

Our specification-driven dynamic binary translator is another implementation of Yirr-Ma’s Emitter interface, illustrated by the class diagram in figure 5.2. According to the interface definition, the implementing `JITCompiler::emit()` method takes the collected semantic information of an interpreted source machine trace as a parameter, generates a dynamic function and returns that dynamic function to the Collector.

Figure 5.3 shows the architecture of our specification-driven dynamic binary translator in relation to the Walkabout emulator. To support experimentation with specification files we chose a hybrid approach for our framework: its three major components, source

**Figure 5.3:**

Architecture of our specification-driven dynamic binary translator. Machine dictionaries are parameterized by SSL specification files: the source machine dictionary contains the operational semantics of all interpreted machine instructions, and the host machine dictionary the state definition of the host machine. In contrast, the instruction selector is generated from SLED and SSL specification files for the host machine.

machine dictionary, host machine dictionary and the instruction selector, are either generic and parameterized by machine specification files, or their implementation is generated from a machine specification.

The two machine dictionaries are generic and load a SSL machine specification upon object construction, thus parameterizing their generic implementation with information about the respective machine. The source machine dictionary loads the operational semantics specification of all interpreted source machine instructions and adapts it to the emulator's runtime implementation of the source machine state, whereas the host machine dictionary loads the state specification of the host machine. In contrast to the dictionaries, both the instruction selector and encoder are generated from the host machine's SLED and SSL specifications.

Given a trace of semantic information about a selected trace, the translator constructs an intermediate representation using the information from the source machine dictionary. This representation consists of the dynamic operational semantics of the interpreted machine instructions. Using the host machine dictionary, the intermediate representation is mapped to the host machine state, i.e. emulated source machine registers are mapped to equivalent host machine registers. The mapped intermediate representation is then optimized and, in a last step, host machine instructions are selected and encoded, thus generating the dynamic function for the given source machine trace. Note that the

generated dynamic functions are as yet not linked with one another, neither locally nor with logically adjacent dynamic functions. The extension, however, is trivial as already shown in section 3.6.

5.4 The Semantic Information

The semantic information that is collected for the dynamic binary translator, and which is then passed from the Collector to the translator, is again divided into instruction implementation and instruction application. In contrast with Yirr-Ma's Inliner, however, the collected information now contains

- as the instruction implementation: a reference to the dynamic operational semantics of an interpreted instruction stored in the source machine dictionary
- as the instruction application: either actual immediate operand values of the interpreted instruction, or host machine addresses of emulated operand registers.

Similar to Yirr-Ma's Inliner, every interpretation function is instrumented such that objects wrap the function's semantic information and are then passed to the Collector. This requires a slightly different instrumentation specification for all source machine instructions than we used for the Inliner.

5.5 Instrumenting a Walkabout Emulator for Yirr-Ma's DBT

As described in sections 3.3 and 4.3, the semantic information about a hot source machine trace is collected during the interpretation of that trace. Because the dynamic binary translator requires only slightly different information to translate the collected trace, we modified the instrumentation previously used for Yirr-Ma's Inliner accordingly.

Listing 5.2 shows the instrumentation of logical source machine instructions, similar to the instrumentation for the Yirr-Ma Inliner in listing 4.1. An interpretation function for a source machine instruction contains the implementation of the instruction's operational semantics (line 11) and again Yirr-Ma's instrumentation code. When the Collector detects a hot trace and switches into collection mode (line 15), the interpretation of instructions causes the Collector to collect the semantic information about those instructions.

The operands of an interpreted source machine instruction, i.e. the actual parameter values of the interpretation function, are again wrapped into `Param` objects. But in contrast to the Inliner a wrapped parameter now contains either a pointer to an emulated operand register, or the immediate operand value (lines 24-26). The collected instruction application in line 27 consists of the value of the emulated `%pc` register of the current

```

1  table9 [ "ANDREG", "ORREG", "XORREG", "ANDCCREG", "ORCCREG",
2          ...
3          ]
4
5  table9 rs1, rs2, rd
6  {
7      // the PC of the interpreted instruction
8      sint32_t _pc = SSL(%pc);
9
10     // actual instruction implementation
11     SSL_INST_SEMANTICS
12
13     // if in collect mode, collect the instruction implementation
14     // and instruction application of an interpreted instruction
15     if (collector.collect()) {
16
17         // instruction implementation is a reference into dictionary
18         static map<string,TableEntry,StrCmp>::const_iterator
19             dict_it = JITCompiler::sourceMachineDict->
20                 idict.find(string("PARAM(table9)"));
21
22         // instruction application
23         JITCInstrApp* ia = new JITCInstrApp(_pc, dict_it,
24             Param((void*) &PARAM(rs1), REGISTER),
25             Param((void*) &PARAM(rs2), REGISTER),
26             Param((void*) &PARAM(rd), REGISTER));
27         collector.addInstrApp(ia);
28     }
29 }
30 ...

```

Listing 5.2:

Instrumentation of interpretation functions for Yirr-Ma's dynamic binary translator. In contrast to the Inliner is the instruction implementation now a reference into the dynamic source machine dictionary, and the instruction application contains pointers to operand registers and/or immediate values. Hot Spot detection and trace collection, however, functions in the same manner.

instruction, the instruction implementation `dict_it` and the `Param` objects. For the dynamic binary translator, the instruction implementation is implemented by a reference into the dynamic source machine dictionary, and is retrieved the first time a particular instruction is collected (line 18).

The semantic information about instructions is again collected into a trace that is then passed to the dynamic binary translator. The translator constructs an intermediate representation using the source machine dictionary, and maps the state of the intermediate representation to the host machine using the host machine dictionary. Both machine dictionaries are essentially runtime representations of the SSL machine specification files.

5.6 Incorporation of Specification Files

Walkabout’s emulator generator uses SLED specification files to generate the emulator’s instruction decoder, and SSL machine specification files to generate the instruction interpretation functions for the emulator core. We thus decided to experiment with annotated versions of SSL specification files rather than introducing yet another specification language. Because the SSL language has some drawbacks when it is used in a dynamic environment and for our purposes, this decision caused problems during later stages of the dynamic binary translation process. To translate the interpreted source machine instructions into host machine instructions, our dynamic binary translator requires the specification of the syntax and operational semantics of the instructions for both machines. Because the emulator already maps the binary encoding of source machine instructions to their respective semantics for interpretation, the translator requires only the specification of the operational semantics of the source machine and the specification of the syntax and operational semantics of host machine instructions for translation.

Our specification-driven dynamic binary translator uses machine dictionaries to represent SSL specification files at runtime. The specification files are loaded into such dynamic machine dictionaries before the Walkabout emulator begins with the interpretation of the source program, thus parameterizing the generic dictionary implementation with actual machine specific information. The use of dictionaries allows for the experimentation with different annotations of SSL files without having to build new machine specific components. In a production environment, however, the information stored in a machine dictionary should be used to generate “hard-wired” implementations of machine specific components, thus removing the added overhead at runtime.

5.6.1 The Source Machine Dictionary

The *static* operational semantics, defined in a SSL file, specifies the effects of instructions over a symbolic and abstract state definition. In contrast, the *dynamic* operational

```

1  ADDFLAGS(op1, op2, result) {
2    *1* %NF := result@[31:31]
3    *1* %ZF := [result = 0 ? 1 : 0]
4    *1* %OF := ((op1@[31:31]) & (op2@[31:31]) &
5               ~(result@[31:31])) | (~(op1@[31:31]) &
6               ~(op2@[31:31]) & (result@[31:31]))
7    *1* %CF := ((op1@[31:31]) & (op2@[31:31])) |
8               (~(result@[31:31]) & ((op1@[31:31]) |
9               (op2@[31:31])))
10 }
11
12 ADDCCimm rs1, simm13, rd
13 *32* %g0 := 0
14 *32* r[tmp] := r[rs1]
15 *32* r[rd] := r[rs1] + sgnex(13,32,simm13)
16 *32* %pc := %npc
17 *32* %npc := %npc + 4;
18 ADDFLAGS(r[tmp], sgnex(13,32,simm13), r[rd]);

```

Listing 5.3:

Static operational semantics of the SPARC `ADDCCimm` instruction. The semantics is defined over the symbolic and abstract SPARC state and operates on formal operands and actual symbolic registers. Flag computation is defined by the macro `ADDFLAGS`.

semantics specifies the effects of instructions over the actual *implementation* of the source machine state on the host machine. Yirr-Ma thus loads the static operational semantics specification of the source machine’s ISA into the source machine dictionary and rewrites it for the actual emulated source machine state, such that the source machine dictionary contains the dynamic operational semantics of all interpreted source machine instructions.

The source machine dictionary is rewritten per instruction, and rewriting is applied to each effect of an instruction individually. This process is comprised of the following steps:

- expand macros at their instantiation site into the sequence of actual effects, such that the operational semantics of all instructions is self-contained and complete
- remove effects that modify pseudo-registers used only by the interpreter (currently, these are effects of the form $CTI_S := n$ and $DLY_A := m$ for CTIs used by the interpreter to trigger dynamic dispatching)
- simplify complex SSL operators by rewriting them using basic SSL operators
- adjust operand locations to the actual runtime implementation of the emulated source machine state.

Listing 5.3 shows the static operational semantics of the SPARC `ADDCCimm` instruction, as defined in SPARC’s SSL file. In addition to the five effects that define the

```

1  ADDCCIMM  rs1,simm13,rd
2  *32* m[102e3988]{32} := 0
3  *32* m[102e1d70]{32} := r[rs1]
4  *32* r[rd] := r[rs1] + sgnex(13,32,simm13)
5  *32* m[102e3abc]{32} := m[102e3ab8]{32}
6  *32* m[102e3ab8]{32} := m[102e3ab8]{32} + 4
7  *1* m[102e3aa0]{1} := (r[rd] & ((2 << 31) - 1)) >>A 31
8  *1* m[102e3ab4]{1} := (r[rd] = 0) ? 1 : 0
9  *1* m[102e3aa1]{1} := (((m[102e1d70]{32} & ((2 << 31) - 1))
10 >>A 31) & ((sgnex(13,32,simm13) & ((2 << 31) - 1))
11 >>A 31)) & ~((r[rd] & ((2 << 31) - 1)) >>A 31)) |
12 (~((m[102e1d70]{32} & ((2 << 31) - 1)) >>A 31) &
13 ~((sgnex(13,32,simm13) & ((2 << 31) - 1)) >>A 31)) &
14 ((r[rd] & ((2 << 31) - 1)) >>A 31))
15 *1* m[102e3a89]{1} := ((m[102e1d70]{32} & ((2 << 31) - 1))
16 >>A 31) & ((sgnex(13,32,simm13) & ((2 << 31) - 1))
17 >>A 31)) | (~((r[rd] & ((2 << 31) - 1)) >>A 31) &
18 ((m[102e1d70]{32} & ((2 << 31) - 1)) >>A 31) |
19 ((sgnex(13,32,simm13) & ((2 << 31) - 1)) >>A 31))

```

Listing 5.4:

Complete, not optimized dynamic operational semantics of the `ADDCCimm` instruction, generated from the static operational semantics in listing 5.3. The `ADDFLAGS` macro is inlined, complex operators are simplified, emulated registers are replaced by typed memory addresses of their implementation and emulated memory accesses have an offset added.

instruction’s semantics (lines 11-15), a macro abbreviates the computation of the flags for `ADDCCimm`. Whereas the effects in lines 11, 14 and 15 operate on actual registers, the register operands of the primary effect in line 13 are determined at runtime. The flags themselves are computed by a sequence of effects that are defined in the body of the `ADDFLAGS` macro. The macro is instantiated as a last effect in line 16, thus adding four more effects to the operational semantics of `ADDCCimm`.

Listing 5.4 shows the generated dynamic operational semantics for the `ADDCCimm` instruction as it is stored in the source machine dictionary. The expansion of the `ADDFLAGS` macro in line 16 is trivial: symbolic parameter values at the instantiation site replace formal parameters in the body of the macro, and the body of the macro then replaces the macro’s instantiation (lines 7-19). There are no effects for the support of the emulator in this example. Line 7 gives an example of how complex SSL operators are simplified and rewritten using basic operators. Currently, our translator rewrites two different operators into simpler expressions:

- expressions of the form $expr@[a:b]$ are rewritten into $(expr \& (2^a-1)) \gg b$
- the logical negation of an expression $!expr$ is rewritten into $(expr=0 ? 1 : 0)$.

Rewriting complex SSL operators into expressions with basic operators prepares the dynamic operational semantics for instruction selection later. Furthermore, to adapt the symbolic and abstract state definition of an instruction to the actual dynamic state

implementation, fixed registers, e.g. `%g0` in line 11 in listing 5.3, are replaced by the typed runtime host memory address of their implementation, for example `m[102e3988]{32}` in line 2. In addition, actual emulated memory accesses like load and stores are adjusted to the runtime implementation by adding the base address of the emulated memory segment to every address computation: static SSL expressions of the form `m[a]` are rewritten into their dynamic equivalent `m[(a) + base]`.

For our experimental machine emulators we instantiated SPARC, Pentium and ARM source machine dictionaries, and examples of the rewritten operational semantics can be found in sections A.2 and A.3 of the appendix, respectively.

5.6.2 The Host Machine Dictionary

The host machine dictionary is parameterized with the SSL state specification describing the host machine. It specifies the host machine registers that are available for the state mapper, their types as well as the endianness used to store values on the host machine. We annotated the SSL specification of the host machine such that the general purpose registers available for register allocation are explicitly marked. Yirr-Ma then instantiates a host machine dictionary by building a pool of available and typed host machine registers from the SSL specification. After the intermediate representation (IR) of an interpreted hot trace is generated, the state mapper uses this register pool to find suitable host machine registers for emulated source machine registers, thus optimizing the implementation of the source machine state for the host machine.

A SSL specification file describes both the type and semantics of registers. The type of a register is defined as a tuple in $\{1, 8, 16, 32, 64\} \times \{int, float\}$: register types are defined as a Cartesian product of the register's bit-width and its value type. The semantics describes properties such as overlapping and sharing of partial registers. Our current implementation of the host machine dictionary does not yet consider such properties during the state mapping.

Listing 5.5 shows an excerpt of the extended SSL state specification of the PowerPC integer register file. Lines 3 to 6 define the set of general purpose integer registers and enumerate them from 0 to 31. The `%sp` and `%fp` registers are aliased to registers number 1 and 31, respectively, to make the specification more readable. Registers that are available for use by the generic state mapper and that are not otherwise reserved by the host machine operating system are aliased as `%un`. These are the registers that are stored into the host machine dictionary, and which are used to implement an optimized source machine state.

Currently, we support only PowerPC and the Lightning virtual machine [Bon03] as host machines for both state mapping and instruction selection. However, we are able to generate mapped and optimized IRs of ARM and Pentium source machines for PowerPC

```

1  # define set of 32-bit integer registers
2  INTEGER
3  [ %r0 , %r1 , %r2 , %r3 , %r4 , %r5 , %r6 , %r7 ,
4    %r8 , %r9 , %r10 , %r11 , %r12 , %r13 , %r14 , %r15 ,
5    %r16 , %r17 , %r18 , %r19 , %r20 , %r21 , %r22 , %r23 ,
6    %r24 , %r25 , %r26 , %r27 , %r28 , %r29 , %r30 , %r31 ] [32] -> 0..31,
7
8  # register aliases
9  %sp [32] -> 1, # %r1 is the %sp
10
11 %u0 [32] -> 14, # available registers for the state mapper
12 %u1 [32] -> 15, # are aliased as %un
13 %u2 [32] -> 16,
14 ...
15 %u16 [32] -> 30,
16
17 %fp [32] -> 31; # %r31 is the frame pointer

```

Listing 5.5:

Excerpt of the SSL state specification of PowerPC's integer register file. General purpose registers that are available to the state mapper are explicitly aliased.

host machines, and we give more examples in appendix A.

5.7 The Intermediate Representation

With both source and host machine dictionaries initialized, the Walkabout emulator begins to interpret and observe a source program, until it detects a hot-spot for translation and collects the semantic information. Then, given the collected semantic information of the hot trace, our specification-driven dynamic binary translator generates a dynamic function in three major steps:

1. it retrieves an IR that represents the dynamic operational semantics of the given trace using the source machine dictionary
2. it optimizes the IR through a state mapping and other common code improvement algorithms
3. it selects and encodes host machine instructions for effects of the IR or parts thereof.

The following sections are dedicated to the discussion of these steps in detail. We illustrate these different steps by translating our SPARC example trace, again shown in listing 5.6, to a PowerPC host machine.

Retrieving the Intermediate Representation

Most common dynamic binary translators abstract source machine instructions into a machine independent and abstract IR that is oriented on classical compiler IRs. In con-

```

1 trace starting at 0xfe02a24
2 out edges: taken 0xfe02a24 fallthrough 0xfe02a34
3 0FE02A24: e2 04 00 00      LDreg      %10, %g0, %11
4 0FE02A28: 80 90 00 11      tst        %11
5 0FE02A2C: 12 bf ff fe      BNE       0xfe02a24
6 0FE02A30: a0 04 20 04      inc       4, %10

```

Listing 5.6:

Selected SPARC trace that serves as example to illustrate the binary translation to a PowerPC host machine throughout the next sections.

trast, Yirr-Ma does not construct such an IR through abstraction: it uses the collected semantic information of a hot trace and the representation of the dynamic operational semantics from the source machine dictionary, as derived from the source machine specification file.

Recall that the semantic information consists of references into the source machine dictionary, representing the instruction implementations, and the actual parameter values of the interpretation functions, representing the instruction applications or operands for the interpreted instructions. Using this information, the IR is constructed by instantiating the instruction implementation with its respective instruction application, and then concatenating the instantiated effects of the individual instructions of the hot trace.

Listing 5.7 shows the static operational semantics of the example trace from listing 5.6 above. The collected semantic information of this trace contains, for each of the four instructions, a reference into the source machine dictionary and the instruction application. In order to generate the IR representing the dynamic operational semantics of the trace, the translator

- binds actual operand values of the instruction application to the free variables in the instruction implementation, thus instantiating the dynamic operational semantics for a particular interpreted instruction
- concatenates the effects of the instantiated instruction implementation of all instructions into a list of effects, the actual IR.

This list of effects represents the complete and self-contained dynamic operational semantics of the original trace of interpreted source machine instructions. Note that the semantics of flags is made explicit in our IR, and is not kept as implicit side-effects. This is because different ISAs define differing semantics for instructions and their flags which is hard to translate implicitly, and because a Walkabout emulator implements flags registers and their computation explicitly.

The generated IR for the example trace is shown in listing 5.8. Formally, the generation of the IR for a given source machine trace t^S is an implementation of the instruction mapping φ_I and the state mapping φ_S . Because Walkabout emulators implement source

```
1  # LDreg %l0, %g0, %l1
2      *32* %g0 := 0
3      *32* %l1 := m[%g0 + %l0]{32}
4      *32* %pc := %npc
5      *32* %npc := %npc + 4
6
7  # ORCCreg %g0, %l1, %g0
8      *32* %g0 := 0
9      *32* %g0 := %g0 | %l1
10     *32* %pc := %npc
11     *32* %npc := %npc + 4
12     *1* %NF := %g0@[31:31]
13     *1* %ZF := [%g0 = 0 ? 1 : 0]
14     *1* %OF := 0
15     *1* %CF := 0
16
17  # BNE 0xfe02a24
18     *1* %COND := ~ %ZF
19     *32* %pc := %npc
20     *32* %npc := [(%COND = 0){1} ? %npc+4 : 0xfe02a24]
21     *1* %CTI := 1      # emulator specific effect
22     *1* %DLYA := 0     # emulator specific effect
23
24  # ADDimm %l0, 4, %l0
25     *32* %g0 := 0
26     *32* %tmp := %l0
27     *32* %l0 := %l0 + sgnex(13,32,4)
28     *32* %pc := %npc
29     *32* %npc := %npc + 4
```

Listing 5.7:

The static operational semantics for the example trace from listing 5.6 defines the manipulation of a symbolic and abstract machine state. Note that the effects in lines 21 and 22 are removed from the dynamic source machine dictionary.

```

1  # LDreg %l0, %g0, %l1
2  m[102e3988]{32} := 0
3  m[102e39cc]{32} := m[(m[102e3988]{32} + m[102e39c8]{32})
4                      + 0x30032000]
5  m[102e3abc]{32} := m[102e3ab8]{32}
6  m[102e3ab8]{32} := (m[102e3ab8]{32} + 4)
7
8  # DRCCreg %g0, %l1, %g0
9  m[102e3988]{32} := 0
10 m[102e3988]{32} := (m[102e3988]{32} | m[102e39cc]{32})
11 m[102e3abc]{32} := m[102e3ab8]{32}
12 m[102e3ab8]{32} := (m[102e3ab8]{32} + 4)
13 m[102e3aa0]{1} := ((m[102e3988]{32} & -1) >>A 31)
14 m[102e3ab4]{1} := ((m[102e3988]{32} = 0) ? 1 : 0)
15 m[102e3aa1]{1} := 0
16 m[102e3a89]{1} := 0
17
18 # BNE 0xfe02a24
19 m[102e3a8a]{1} := ~ m[102e3ab4]{1}
20 m[102e3abc]{32} := m[102e3ab8]{32}
21 m[102e3ab8]{32} := ((m[102e3a8a]{1} = 0) ? (m[102e3ab8]{32} + 4)
22                      : 0xfe02a24)
23 # ADDimm %l0, 4, %l0
24 m[102e3988]{32} := 0
25 m[102e1d70]{32} := m[102e39c8]{32}
26 m[102e39c8]{32} := (m[102e39c8]{32} + sgnex(13, 32, 4))
27 m[102e3abc]{32} := m[102e3ab8]{32}
28 m[102e3ab8]{32} := (m[102e3ab8]{32} + 4)

```

Listing 5.8:

Complete dynamic operational semantics of the example trace from listing 5.6. The effects of individual instructions of the trace are instantiated and concatenated, thus generating a complete and compilable IR.

machine registers using a compound data structure that is stored in the host machine's memory, the generated IR consists of effects which operate on values stored in the host machine's memory. Note the primary effect of the `LDreg` instruction in listing 5.7 in line 3: when the Walkabout emulator relocates an emulated memory segment, it must account for the relocation in its dynamic operational semantics. Therefore, the base address `0x30032000` of the emulated memory segment is added to `LDreg`'s address calculation (listing 5.8 line 3).

5.8 State Mapping

The generated IR can already be compiled for the host machine³ which would lead to a poor quality of the generated host machine instructions. The main advantage of an IR is its potential for improvement through the application of different dynamic optimizations. This is especially so for long traces. Yirr-Ma handles every hot trace as a black box within which optimizations are applied at will, as long as the emulated machine state is consistent at the boundaries of the generated dynamic function. In the scope of the dynamic function the emulated state is a global data structure.

The first optimization that the translator applies to the IR is a *state mapping*. In fact, this state mapping is the implementation of the dynamic optimization function ω_{S_H} which optimizes the implementation of the emulated source machine state for the host machine. In this context, ω_{S_H} is a mapping of emulated source machine registers to equivalent actual host machine registers. In contrast to traditional register allocation, our state mapping is implemented by a lightweight algorithm that is guided by the allocated registers of the interpreted instructions. Additionally, in order to maintain a consistent source machine state for a mapped trace, a prologue and an epilogue must be added to the IR: the prologue consists of a sequence of effects that load the content of emulated registers into their mapped counterparts on the host machine, and the epilogue stores their content back in the emulated register file. Listing 5.9 depicts the IR mapped to actual PowerPC host machine registers, and listing 5.10 shows the same IR mapped to Pentium host machine registers.

The current implementation of our state mapper generates a fixed and trace-global register mapping by performing two linear scans over the original IR. During the first scan it generates the set of defined source machine registers. If the left-hand side of an effect is an emulated register, i.e. a memory reference of the form $m[a]_{type}$ ⁴, then the state mapper assigns a host machine register of the same type to that address and

³ In fact, for a fast instruction translation the generation of an IR can be omitted completely. The effects that specify the dynamic operational semantics of the hot source machine instructions can be compiled instructions wise and directly into host machine instructions (cf. also section 5.10).

⁴ Recall that actual emulated memory accesses to an emulated address a have the base address of the emulated memory segment added, thus are of the form $m[a + 0x30032000]$.

```

1  r[17] := m[102e3abc]{32} # %pc ↦ r17
2  r[14] := m[102e3ab8]{32} # %npc ↦ r14
3  r[16] := m[102e39cc]{32} # %l1 ↦ r16
4  r[19] := m[102e39c8]{32} # %l0 ↦ r19
5  r[15] := m[102e3988]{32} # %g0 ↦ r15
6  r[18] := m[102e1d70]{32} # %tmp ↦ r18
7  r[15] := 0 # LDreg %l0, %g0, %l1
8  r[16] := m[(r[15] + r[19]) + 0x30032000]
9  r[17] := r[14]
10 r[14] := (r[14] + 4)
11 r[15] := 0 # DRCCreg %g0, %l1, %g0
12 r[15] := (r[15] | r[16])
13 r[17] := r[14]
14 r[14] := (r[14] + 4)
15 m[102e3aa0]{1} := ((r[15] & -1) >>A 31)
16 m[102e3ab4]{1} := ((r[15] = 0) ? 1 : 0)
17 m[102e3aa1]{1} := 0
18 m[102e3a89]{1} := 0
19 m[102e3a8a]{1} := ~ m[102e3ab4]{1} # BNE 0xfe02a24
20 r[17] := r[14]
21 r[14] := ((m[102e3a8a]{1} = 0) ? (r[14] + 4) : 0xfe02a24)
22 r[15] := 0 # ADDimm %l0, 4, %l0
23 r[18] := r[19]
24 r[19] := (r[19] + sgnex(13,32,4) )
25 r[17] := r[14]
26 r[14] := (r[14] + 4)
27 m[102e1d70]{32} := r[18] # epilogue
28 m[102e3988]{32} := r[15]
29 m[102e39c8]{32} := r[19]
30 m[102e39cc]{32} := r[16]
31 m[102e3ab8]{32} := r[14]
32 m[102e3abc]{32} := r[17]

```

Listing 5.9:

As a first optimization, the IR state is mapped to actual host machine registers, PowerPC in this example. Typed memory addresses are assigned to host registers of the same type, thus performing a strict mapping of emulated machine registers to actual host machine registers. A prologue and an epilogue are added to the IR which load the content from an emulated register into a host machine register, and then write the computed values back into the emulated register file.

```

1  r[ebx] := m[102c295c]{32} # %pc ↦ %ebx
2  r[eax] := m[102c2958]{32} # %npc ↦ %eax
3  r[edx] := m[102c286c]{32} # %l1 ↦ %edx
4  r[edi] := m[102c2868]{32} # %l0 ↦ %edi
5  r[ecx] := m[102c2828]{32} # %g0 ↦ %ecx
6  r[esi] := m[102c0c80]{32} # %tmp ↦ %esi
7  r[ecx] := 0 # LDreg %l0, %g0, %l1
8  r[edx] := m[(r[ecx] + r[edi]) + 0x30032000]
9  r[ebx] := r[eax]
10 r[eax] := (r[eax] + 4)
11 r[ecx] := 0 # ORCCreg %g0, %l1, %g0
12 r[ecx] := (r[ecx] | r[edx])
13 r[ebx] := r[eax]
14 r[eax] := (r[eax] + 4)
15 m[102c2940]{1} := ((r[ecx] & -1) >>A 31)
16 m[102c2954]{1} := ((r[ecx] = 0) ? 1 : 0)
17 m[102c2941]{1} := 0
18 m[102c2929]{1} := 0
19 m[102c292a]{1} := ~ m[102c2954]{1} # BNE 0xfe02a24
20 r[ebx] := r[eax]
21 r[eax] := ((m[102c292a]{1} = 0) ? (r[eax] + 4) : 0xfe02a24)
22 r[ecx] := 0 # ADDimm %l0, 4, %l0
23 r[esi] := r[edi]
24 r[edi] := (r[edi] + sgnex(13,32,4) )
25 r[ebx] := r[eax]
26 r[eax] := (r[eax] + 4)
27 m[102c0c80]{32} := r[esi] # epilogue
28 m[102c2828]{32} := r[ecx]
29 m[102c2868]{32} := r[edi]
30 m[102c286c]{32} := r[edx]
31 m[102c2958]{32} := r[eax]
32 m[102c295c]{32} := r[ebx]

```

Listing 5.10:

IR state mapped to a Pentium host machine. In this case, the Pentium host provided the exact amount of 32-bit integer registers required to map all emulated SPARC registers. Note that the implementation of the emulated register file is located at a different address, thus changing the dynamic operational semantics of the IR trace.

records the mapping in the mapping table:

$$\begin{aligned} m[102e3ab8]_{32i} &\mapsto r[14], & m[102e3988]_{32i} &\mapsto r[15], & m[102e39cc]_{32i} &\mapsto r[16], \\ m[102e3abc]_{32i} &\mapsto r[17], & m[102e1d70]_{32i} &\mapsto r[18], & m[102e39c8]_{32i} &\mapsto r[19]. \end{aligned}$$

A suitable register is allocated from the host machine dictionary. In the above example listing 5.9, the typed memory address $m[102e3ab8]\{32\}$, representing a 32-bit SPARC integer register, is mapped to the 32-bit PowerPC integer register $r14$. If the host machine dictionary does not contain a register of the required type, the emulated register is not mapped. The emulated flag registers in lines 15 to 19, for example, are not mapped to actual host machine registers because the host machine does not provide 1-bit integer registers. On the other hand, if the state mapper runs out of registers because the trace uses more registers than the host machine is able to provide, it leaves memory references in the IR, thus having a negative impact on the quality of the generated host machine code. During its second scan, the state mapper replaces all occurrences of typed memory addresses with their respective host machine register and generates and adds both prologue effects (lines 1-6) and epilogue effects (lines 27-32) to the IR.

Improving the State Mapper

Our initial implementation of the state mapper performs rather poorly for a trace that defines many emulated source machine registers on a host machine with less(er) available registers. For example, the state mapper can run out of registers quickly when it maps an interpreted SPARC trace to a Pentium host machine. However, this is a general problem of RISC to CISC state mapping, and a common compiler task when many local variables (i.e. virtual registers) must be allocated to few actual registers.

One improvement over our technique is a load-based register mapping. The frequency of register uses are weighted against one another, and only the most frequently used source machine registers are then mapped to actual host machine registers. The remaining emulated registers are not mapped, resulting in the continuous execution of load and store instructions. Special care must be taken when shared registers (e.g. Pentium or M68k) are mapped to a host machine that does not support these register properties. In that case, the outermost register of a set of shared registers can be mapped, and shared sub-registers are then emulated by inserting effects for masking these sub-registers into the IR.

The most appropriate solution, obviously, is a register allocator as utilized by common (static) compilation techniques [Sco99]. Instead of a direct register mapping as implemented by our current state mapper, emulated registers are mapped to equivalent virtual registers that are then allocated to host machine registers. Liveness analysis and spills of register values for the reuse of registers result in better code quality, but require more runtime resources for their generation.

5.9 Dynamic Optimization of the Mapped IR

An IR representing the dynamic operational semantics over a *mapped* source machine state holds significant potential for further dynamic optimizations. Formally, the dynamic optimization and the subsequent compilation of the IR into host machine instructions implements the parameterized translation function $\tau^{\omega_{SH}}$ from section 2.4. YirmMa’s translator implements dynamic optimization techniques that may be applied to the IR itself. They are as yet independent from the actual host machine, and host machine specific optimizations are not implemented. Experimentation with different optimization techniques showed that some optimizations are better suited for certain source machines than others, and we elaborate on that later on. The optimization algorithms we use in the following are adapted from traditional static compilation, as described for example in [Sco99]. In contrast to the static techniques, they incorporate dynamic feedback information not available to a static compiler. The application of dynamic optimizations should be considered carefully because they introduce additional overhead for the generation of host machine code. In the following we describe our optimizations, and illustrate their application using the mapped IR from listing 5.9.

The Walkabout emulator does not support exceptions raised by interpreted machine instructions, and it terminates immediately when a host machine exception occurs during program interpretation. The dynamic optimization of the IR reorders and/or removes effects, thus effectively blurring the boundaries of interpreted machine instructions within the translated trace. The precise location of a source machine instruction that raises an exception is therefore hard for dynamically optimized traces.

5.9.1 Pseudo Constants

For every single interpreted machine instruction, the value of the emulated `%pc` register, and for SPARC source machines both the values of the `%pc` and `%npc` registers, are local *pseudo constants* in the scope of the one instruction’s operational semantics, and can therefore be handled as such. For the example trace in listing 5.9, we add an effect right after the trace prologue that assigns the value of the emulated `%npc` register for the first interpreted instruction to the mapped host machine register:

```
r[14] := 0xfe02a28    # recall that %npcS ↦ r14
```

Subsequent optimizations, namely constant folding and constant propagation, exploit this added information. For all and in particular long traces this optimization improves the IR drastically by introducing a constant which causes the elimination of almost all secondary effects that modify the emulated `%pc` and `%npc` registers.

5.9.2 Constant Propagation

A constant is an immediate value that is either introduced by an immediate operand of the interpreted source machine instruction, or as part of the instruction’s dynamic operational semantic specification. If an effect writes a constant into a host machine register, subsequent uses of this register can be replaced by the constant itself, until another effect redefines that register.

```
r [15] := 0
r [16] := m[(r [15] + r [19]) + 0x30032000]
```

The IR shows the first two effects of the dynamic operational semantics of the interpreted LDreg %10,%g0,%11 instruction. The constant 0 is assigned to register r[15], and r[15] is then used in the next effect. The constant 0 can therefore be propagated, such that

```
r [15] := 0
r [16] := m[(0 + r [19]) + 0x30032000]
```

become the new effects.

Constant propagation is implemented as one linear sweep over the entire IR. Registers that get a constant assigned are marked and their subsequent uses are replaced with that constant. When a marked register is redefined, the propagation of the associated constant terminates. Constant propagation

- introduces new opportunities for constant folding (i.e. the premature evaluation of expressions)
- leaves the effects that assign a constant to a register “dead” in the context of the subject trace, thus allowing the removal of the dead effect.

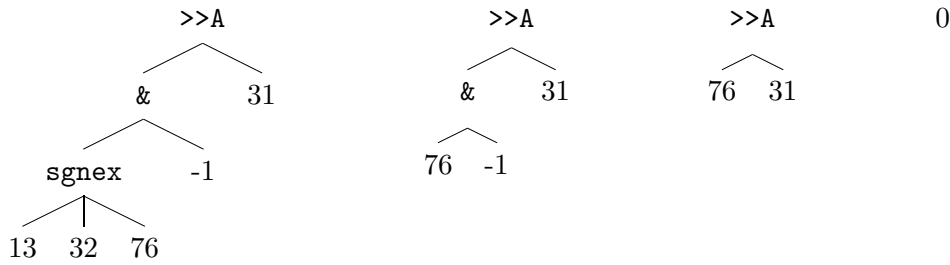
We discuss the combination of constant propagation together with constant folding and dead effect elimination in the following.

5.9.3 Constant Folding

Constant folding is applied to individual effects, in combination with constant propagation, during the linear sweep over the IR. Every effect represents an expression tree which is evaluated bottom-up, thus simplifying the entire effect.

As mentioned above, constants are introduced into an effect either through immediate operands of the interpreted source machine instructions, as part of the operational semantics of an instruction, or through constant propagation. The following two effects, as continuation of the above example, are retrieved by folding an expression of the form r_n+0 into r_n :

```
r [15] := 0
r [16] := m[r [19] + 0x30032000]
```

**Figure 5.4:**

Bottom-Up constant folding of a sub-tree of a given SSL effect. Operators with constant operands are evaluated, thus replacing an entire sub-tree with an immediate value, the result of the evaluation. Note the type inference flow in this example. The 13-bit integer value 76 is sign extended into a 32-bit value, thus propagating the 32-bit signed integer type further up into the tree. The SSL operators $\&$ and $\gg A$ are polymorphic and thus receive type information from their typed operands.

Especially effects that compute the value of emulated flag registers can be simplified immensely through constant folding. For instance, the instantiated dynamic operational semantics defining the %OF flag for the emulated SPARC instruction `SUBccimm %o0,76,%g0`, with the state mapping $\%o0 \mapsto r[21]$ and $\%g0 \mapsto r[15]$, looks as follows:

```
m[102e3aa1]{1} := (((((r[21] & -1) >>A 31) & ~((sgnex(13,32,76)
& -1) >>A 31)) & ~((r[15] & -1) >>A 31)) | ((~((r[21]
& -1) >>A 31) & ((sgnex(13,32,76) & -1) >>A 31)) &
((r[15] & -1) >>A 31)))
```

The semantics of emulated flags is in general defined by complex boolean expressions. Given the immediate operand value 76 of the `SUBccim` instruction, the above complex expression is folded into

```
m[102e3aa1]{1} := ((r[21] >>A 31) & ~ (r[15] >>A 31))
```

which results in a much higher quality of the generated host machine code.

Both sides of an assignment effect are evaluated through a recursive depth-first descent into the expression tree, and constants are then evaluated bottom-up. Figure 5.4 illustrates the rewrite of a sub-tree of the example effect. The resulting constant 0 is then used further to simplify the expression tree.

Folding constants simplifies, and thus modifies, the representation of the dynamic operational semantics of an instruction. This can cause the translation of source machine instructions into different host machine instructions. For example, SPARC's `ANDN` instruction with an immediate value operand has this primary effect specified:

```
*32* r[rd] := r[rs1] & ( ~sgnex(13,32,simm13) )
```

The instantiation of this instruction, for example with `ANDN %g1,7,%l0`, introduces the sub-expression $\sim\text{sgnex}(13,32,7)$ which is folded to -8 , resulting in the effect

```
*32* %l0 := %g1 & ( ~sgnex(13,32,7) )  $\rightarrow$  %l0 := %g1 & -8
```

On our PowerPC host machine, for example, this effect is then compiled into an `andi` instruction.

5.9.4 Dead Effect Elimination

Similar to dead assignment elimination, an effect is said to be “dead” if the left-hand side location of the effect is redefined without being used in between the two definitions. For example, the prologue of our example trace can be reduced to a single load effect:

```
r[19] := m[102e39c8]{32} # %l0 ↦ r19
```

The prologue effects in lines 1, 2, 3, 5 and 6 are dead (cf. the mapped IR in listing 5.9). Without any intermediate use, register `r[17]` in line 1 is redefined in line 9, `r[14]` in line 2 is redefined by the introduced pseudo constant for the mapped `%npc` register, `r[16]` is redefined in line 8, and so are `r[15]` in line 7 and register `r[18]` in line 23. These load effects can therefore be removed. More dead effects are the assignments to the mapped `%pc` register in lines 13 and 20.

This optimization is especially valuable for CISC source machines. Most CISC instructions compute flag values, which are usually not used by subsequent instructions. Therefore, an IR of a CISC source machine trace, for example Pentium, contains numerous dead effects which are removed.

5.9.5 Idiom Replacement

To take advantage of some idiosyncrasies of the host machine, sub-expressions or entire effects can be replaced with semantically equivalent sub-expressions or effects that improve the quality of the generated host machine code. Idioms are host machine specific and can be safely applied as a last step before instruction selection; idioms for a particular machine are usually found in the architecture manuals.

For experimentation, Yirr-Ma’s translator implements idiom replacement in a separate linear sweep over the IR. For performance reasons, however, idiom replacement could be applied right after constant folding during the same sweep. For example, the effect

```
r[15] := 0
```

is replaced by the idiom

```
r[15] := r[15] - r[15]
```

for the PowerPC host machine.

Furthermore, if the host architecture provides a register `r0` which always reads the value 0, the above effect can alternatively be idiomized into

```
r[15] := r0
```

In addition, constants 0 in already folded effects can be replaced with the register `r0`. Our current implementation does not define any other PowerPC idioms.

```

1  r[19] := m[102e39c8]{32}           # prologue: load %l0s
2  r[16] := m[r[19] + 0x30032000]     # LDreg %l0, %g0, %l1
3  r[15] := r[16]                     # tst %l1
4  m[102e3aa0]{1} := (r[15] >>A 31)
5  m[102e3ab4]{1} := ((r[15] = 0) ? 1 : 0)
6  m[102e3aa1]{1} := 0
7  m[102e3a89]{1} := 0
8  m[102e3a8a]{1} := ~ m[102e3ab4]{1} # BNE 0xfe02a24
9  r[14] := ((m[102e3a8a]{1} = 0) ? 0xfe02a34 : 0xfe02a24)
10 r[15] := (r[15] - r[15])           # inc 4, %l0
11 r[18] := r[19]
12 r[19] := (r[19] + 4)
13 r[17] := r[14]
14 r[14] := (r[14] + 4)
15 m[102e1d70]{32} := r[18]           # epilogue
16 m[102e3988]{32} := 0
17 m[102e39c8]{32} := r[19]
18 m[102e39cc]{32} := r[16]
19 m[102e3ab8]{32} := r[14]
20 m[102e3abc]{32} := r[17]

```

Listing 5.11:

The application of dynamic optimizations simplifies the IR trace: effects are evaluated and dead effects removed. That, in turn, improves the quality of the emitted host machine instructions.

5.9.6 More Dynamic Optimizations

The optimized IR of the example trace produced by Yirr-Ma is shown in listing 5.11. As can be seen, the generated IR is compacter than its original version. Most effects, especially those computing emulated flags, are simplified, and some are even removed from the IR. In the next step host machine instructions are selected for this IR.

In the following we discuss two more optimizations for the IR. Although both improve the IR of a source machine trace, we have not yet implemented them.

Expression Propagation

With expression propagation, the right-hand side expression of an assignment effect replaces occurrences of the left-hand side location in one or more subsequent effects, until the left-hand side is redefined. With respect to code redundancy, this is useful only if there is no more than one use of the location for complex expressions. On the other hand, with effects of the form $r_n := r_m$ the right-hand side register r_m can replace any number of occurrences of r_n until either r_n or r_m is redefined.

For example, the value of register `r[16]` is assigned to register `r[15]` in listing 5.11 line 3, and can therefore replace the two occurrences of register `r[15]` in lines 4 and 5. Also, the effect in line 9 defines the new value for the emulated `%npc` register, mapped to `r[14]`. Because `r[14]` is used only once before its redefinition, the right-hand side expression of the effect can be propagated into line 13.

Similar to constant propagation, the original defining effect is left dead and can be

```

1  r[19] := m[102e39c8]{32}
2  r[16] := m[r[19] + 0x30032000]
3  m[102e3aa0]{1} := (r[16] >>A 31)
4  m[102e3ab4]{1} := ((r[16] = 0) ? 1 : 0)
5  m[102e3aa1]{1} := 0
6  m[102e3a89]{1} := 0
7  m[102e3a8a]{1} := ~ m[102e3ab4]{1}
8  r[19] := (r[19] + 4)
9  r[17] := ((m[102e3a8a]{1} = 0) ? 0xfe02a34 : 0xfe02a24)
10 r[14] := (r[17] + 4)
11 m[102e39c8]{32} := r[19]
12 m[102e39cc]{32} := r[16]
13 m[102e3ab8]{32} := r[14]
14 m[102e3abc]{32} := r[17]

```

Listing 5.12:

The application of further dynamic optimizations results in a very compact IR. The original IR is stripped of almost all secondary effects, reducing the IR to 14 effects, 5 of which emulate SPARC flags and another 5 function as prologue and epilogue, respectively. The remaining 4 effects implement the original SPARC instructions.

removed (lines 3 and 9 in the above example trace). The propagation of expressions also introduces new potential for constant folding if the expression is propagated such that new sub-expressions are created which can be evaluated in turn.

Machine and Emulator Peculiarities

With additional knowledge about the properties of the emulated register file and the specification of the operational semantics of instructions, the epilogue can be improved: store effects of the registers `r[18]` in line 15 (the emulated `%tmp` register) and `r[15]` in line 16 (the emulated `%g0` register which can always be replaced with the constant 0) can be removed.

These two registers are used only in the scope of a single source machine instruction and hence are irrelevant for the consistency of the emulated source machine state. Removing the stores leaves the effects in lines 10 and 11 dead for removal.

Listing 5.12 finally shows the optimized intermediate representation for the original example trace. The IR now contains 14 effects, including the prologue in line 1 and the epilogue in lines 11-14. The remaining 9 effects implement the 4 original SPARC instructions: `LDreg` in line 2, `TST` in lines 3-7 where each of the effects implements an emulated flag register, `BNE` in line 9 and `INC` in line 8.

Host machine instructions can now be selected for the individual effects of the optimized IR, thus completing the dynamic binary translation.

5.10 Code Generation

In a last step, the translator generates binary host machine code from the IR by

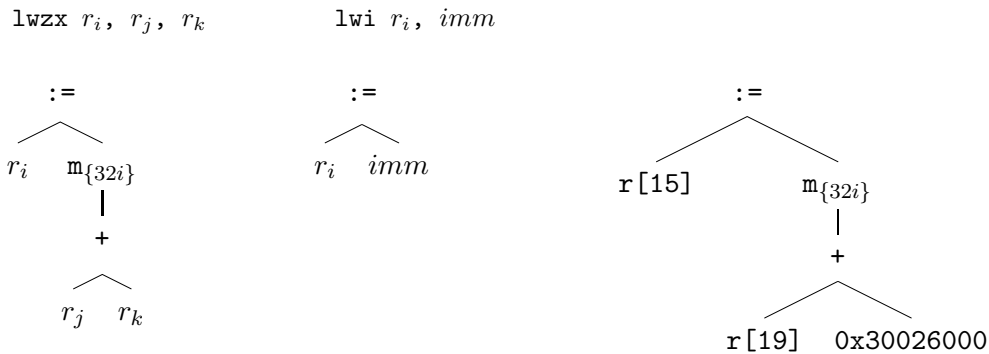
- selecting a suitable host machine instruction, or a sequence thereof, for every single effect in the IR
- encoding the selected instructions into their binary representation and writing them to the body of the dynamic function.

Recall that the operational semantics of an instruction in SSL is specified by a list of effects: one primary effect representing the instruction’s main function, and a sequence of secondary effects representing flags and `%pc` computation. The IR of an interpreted trace of source machine instructions is then constructed from all effects, thus specifying the dynamic operational semantics of the entire trace. For every single effect of the IR, the instruction selector now selects a host machine instruction whose *primary* effect matches the given IR effect, either partially or completely. Instruction selection thus implements the actual “binary translation” step and therewith the mapping φ_I .

Both Yirr-Ma’s instruction selector and instruction encoder can be generated from SSL and SLED specifications, respectively. Unfortunately, due to the time constraints towards the end of our research, we were unable to implement the instruction selector generator itself. We do, however, give a formal description of the instruction selection process that we implemented by hand, and present the algorithms to generate such an instruction selector from a SSL specification file.

Many instruction selectors are generated from machine specification files that define an annotated bottom-up rewrite grammar [FHP92, FSW94, GL97]. In contrast, SSL specifies the operational semantics of instructions by a sequence of effects, one of which represents the primary effect of the instruction. An effect, in turn, is an assignment expression which is represented by a tree, where nodes are SSL operators and leaves are operands: actual host machine registers, formal operand symbols or immediate values. Therefore, instead of a rewrite grammar, our instruction selector must be generated from a set of such trees, and it must emit the instruction associated with a particular selected tree. Note that SSL does neither weigh instructions nor attribute primary effects with costs which makes *optimal* instruction selection harder.

The instruction selection of host machine instructions per IR effect is implemented by a modified *maximal munch algorithm* [Cat78] that first consumes an IR effect top-down, and then covers IR sub-trees bottom-up with primary effect trees of host machine instructions, thus selecting appropriate host machine instructions. The selected host machine instructions are then encoded into their binary representation by an instruction encoder that is, similar to Walkabout’s instruction decoder for the emulated source machine, generated from the SLED specification of the host machine using the NJMC Toolkit.

**Figure 5.5:**

Tiles for two PowerPC instructions defining the primary effects in the SSL specification of the host machine (left and middle), and an actual IR effect (right). The two tiles together match (or cover) the IR effect completely, thus selecting host machine instructions that implement the given IR tree.

5.10.1 Instruction Selection Using Tiling

In a SSL file, the semantics of available instructions of a host machine is specified by a set of trees or *tiles*: a tile represents the primary effect of an instruction. A tile is an expression tree defined over SSL operators and formal operands, i.e. symbols for host machine registers and immediate values, and the IR effects are expression trees defined over SSL operators and actual operands, i.e. actual host machine registers and immediate values. Given the primary effect of an instruction, i.e. its tile, the instruction's mnemonic and operands are associated with the tile. Instruction selection therefore amounts to

- covering a given IR effect with a (minimal) number of non-overlapping tiles
- binding the symbols of the tiles to the actual registers and immediate values of the IR effect.

For example, figure 5.5 depicts the tiles representing the two PowerPC instructions `lwzx` and `lwi`, and the derived IR effect for the SPARC `LDreg` instruction from our example trace. The PowerPC instruction `lwzx` loads a 32-bit value from a typed memory address, the sum of registers r_j and r_k , into a destination register r_i , and `lwi` loads a 32-bit integer value imm into a register r_i . The leaf nodes of the two tiles are symbols: placeholders that are bound to actual host machine registers or immediate values during instruction selection. The IR effect at the right can be covered completely by the two tiles binding the operand symbols of the tiles to actual host machine registers and immediate values of the IR effect, and thus delivering actual operands for the selected machine instruction.

The process of tiling a given IR effect is implemented by a tree pattern matcher. In order to specify the pattern matching algorithm, we first define its data types: the set

of general trees, and its two sub-types the set of tiles (i.e. primary effects in an SSL specification file), and the set of IR effects (i.e. dynamic instantiations of arbitrary SSL expressions).

Trees

Let Δ be an alphabet of ranked operator symbols, and let $(\Delta, \text{rank}_\Delta)$ be a Δ -finite set. $\text{rank}_\Delta : \Delta \rightarrow \mathbb{N}^+$ is a function that associates an operator $\delta \in \Delta$ with a number $n \in \mathbb{N}^+$, the rank of δ , denoted as $\delta^{(n)}$. The set of SSL operators is defined as $\Delta_{SSL} = \{?, ^{(3)}, :=^{(2)}, +^{(2)}, =^{(2)}, m^{(1)}, \dots\}$. For brevity, we use Δ instead of Δ_{SSL} in the following. Also, $\Delta^{(n)} \subseteq \Delta$ denotes the set of SSL operators of rank n .

Let V be an arbitrary set of leaf symbols that we define for tiles and IR effects individually. The set $T_{\Delta, V}$ of trees is the smallest set $T_{\Delta, V} \subseteq (\Delta \cup V \cup \{(\, , \,)\})^*$ such that

$$\text{if } \delta \in \Delta^{(n)} \text{ with } n \geq 1 \text{ and } t_1, \dots, t_n \in V \cup T_{\Delta, V}, \text{ then } \delta(t_1, \dots, t_n) \in T_{\Delta, V} \quad (5.1)$$

holds. A tree, therefore, consists of an operator δ of rank n , and n sub-trees (the operands of δ) that are either leaf symbols in V or other trees in $T_{\Delta, V}$.

Tiles

Let $V_s = \{imm, r_i\}$ with $i \in \mathbb{N}$ be a set of leaf symbols that represent formal operand symbols for the primary effect of an instruction specification. The set of tiles over SSL operators Δ and leaf symbols V_s is the set T_{Δ, V_s} as defined in definition 5.1 above. The SSL specification of a (host) machine implicitly defines the set T_{Δ, V_s} of tiles for machine instructions. For example, the two tiles of the PowerPC instructions from figure 5.5 are denoted as $:=^{(2)}(r_i, m^{(1)}(+^{(2)}(r_j, r_k)))$ for `lwzx` and $:=^{(2)}(r_i, imm)$ for `lwi`.

IR effects

Similarly, let $V' = R \cup \mathbb{N}$ be a set of actual (host) machine registers R and integer values \mathbb{N} ⁵. The set of trees that represent the effects of an IR (i.e. the dynamic operational semantics of a trace of source machine instructions), is the set $T_{\Delta, V'}$ as defined in definition 5.1 above. For example, the IR effect in figure 5.5 is denoted as $:=^{(2)}(r_{15}, m^{(1)}(+^{(2)}(r_{19}, 0x30032000)))$.

Instruction Selection through Tree Pattern Matching

We specify the recursive algorithm that implements the matching (i.e. covering) of a given IR effect with a number of tiles, and thus instruction selection, as follows. Let $t = \delta^{(n)}(t_1, \dots, t_n) \in T_{\Delta, V_s}$ be a tile and $t' = \delta'^{(n)}(t'_1, \dots, t'_n) \in T_{\Delta, V'}$ be an IR effect. The tile t matches the IR effect t' , iff

⁵ At the moment we do not consider floating point values. The formal extension of tiles and IR effects for their support, however, is trivial, their actual implementation for different machines not so.

- $\delta^{(n)} = \delta'^{(n)}$ (i.e. the ranked operator is the same for both trees), and
- for every $t_i \in \{t_1, \dots, t_n\}$ and $t'_i \in \{t'_1, \dots, t'_n\}$ (i.e. for every sub-tree t_i of the tile t and for every sub-tree t'_i of the IR effect t') one of the following holds:
 - $t_i = \delta_i^{(m)}(t_{i,1}, \dots, t_{i,m})$ and t_i matches t'_i (i.e. both sub-trees t_i and t'_i match recursively), or
 - $t_i = imm$ and $t'_i \in \mathbb{N}$ and $imm = t'_i$ (i.e. t_i is an operand symbol imm for an immediate value and t'_i is an actual immediate value, thus t_i matches t'_i completely and the operand symbol is bound to the immediate value), or
 - $t_i = r_n$ and
 - $t'_i \in R$ and $r_n = t'_i$ (i.e. t_i is an operand symbol r_n for registers and t'_i is an actual register, thus t_i matches t'_i completely and the operand symbol is bound to the actual register), or
 - $t'_i \in T_{\Delta, V'}$ and $r_n = r_{tmp}$ with $r_{tmp} \in R$ (i.e. the tile t leaves t'_i uncovered and t_i is thus bound to a temporary register r_{tmp}). The temporary IR effect $:=^{(2)}(r_{tmp}, t'_i) \in T_{\Delta, V'}$ is created from the uncovered sub-tree and recursively matched with tiles in T_{Δ, V_s} .

Informally, instruction selection works as follows. For every given IR effect t' the instruction selector searches the set of available tiles for a tile t that matches t' either partially or completely. The successful match results in a binding of leaf symbols (i.e. formal operand symbols V_s) of t to either an actual register in R or an immediate value in \mathbb{N} , therewith deriving operands for the selected instruction. The machine instruction that is specified by t is then emitted into the dynamic function. Because a successful match of an IR effect with a tile requires uncovered sub-trees (i.e. generated temporary IR effects) to be first matched bottom-up with other tiles, the host machine instructions are emitted in their correct order.

It is, however, the responsibility of the specification writer to provide a tile, i.e. primary SSL effect, for every SSL operator. The minimal tile operates on register operands, and therefore a complete SSL specification of the host machine must at least contain register forms of host machine instructions for every SSL operator used in the source machine specification file. See section 5.10.3 below for more details.

5.10.2 An Example

Given the optimized IR trace from listing 5.11, the IR effect for the interpreted SPARC LDreg instruction

```
r[16] := m[r[19] + 0x30032000]
```

is the tree $:=^{(2)} (r_{16}, m^{(2)}(+^{(2)}(r_{19}, 0x30032000))) \in T_{\Delta, V_s}$ (see also figure 5.5 above). The tile representing the `lwzx` instruction matches partially the IR effect with the leaf symbol binding $r_i = r_{16}$, $r_j = r_{19}$ and $r_k = r_{20}$. This partial matching leaves the sub-tree `0x30032000` uncovered, therewith generating the temporary IR effect

```
r[20] := 0x30032000
```

or $:=^{(2)} (r_{20}, 0x30032000) \in T_{\Delta, V_s}$. The temporary effect now matches the pattern that specifies the `lwi` instruction with the leaf symbol binding $r_i = r_{20}$ and $imm = 0x30032000$. This complete match of the temporary IR effect causes the emission of the instruction `lwi r20, 0x30032000`. Now that all sub-trees of the original IR effect are completely matched, the tile of the `lwzx` instruction matches and `lwzx r16, r19, r20` is emitted. The final sequence of host machine instructions is as follows (note that `lwi` is synthesized from the `lis` and `ori` instructions):

```
lis    r20, 0, 0x3003      # lwi r20, 0x30032000
ori    r20, r20, 0x2000
lwzx   r16, r19, r20
```

Such is (the primary effect of) the original SPARC LD instruction “translated” into the PowerPC instruction `lwzx`. The overhead of two instructions accounts for Walkabout’s state implementation that relocates the emulated memory segment. The relocation is compensated for by the offset `0x30032000` which is added to LD’s effective address computation.

5.10.3 Additional Remarks

The devil is in the detail, and nothing else holds for our research. This section discusses the issues and shortcomings of our instruction selector and its implementation, and it shows our approaches to resolve those problems.

Completeness of SSL

To guarantee a successful translation of IR effects into host machine instructions, the SSL specification of the host machine must be complete. That means that every SSL operator must be implemented (or used) by the primary effect of a host machine instruction, such that the use of an SSL operator in an IR effect can be matched to a host machine instruction.

IR Types

For simplicity and better readability we omitted types in the definition of the pattern matcher above. In fact, some SSL operators are overloaded and have not only a rank but also an *implicit* type (e.g. $+_{32i}^{(2)}$ for a 32-bit integer addition) associated that must be inferred from its operands. Other operators, in contrast, *are* typed but propagate their type and context information into the operand trees (e.g. $sgnex^{(3)}(from, to, expr)$

defines *expr* to be of type *from*-bit integer, and is itself of type *to*-bit integer). Pattern matching over operators and operands is therefore required to be type sensitive. The extension of our algorithm specification with types is trivial: the instruction selector implements a type sensitive tree pattern matching as indicated.

The current implementation of the SSL language is not type complete, nor is it type explicit. For example, the memory access operator `m[.]` may be annotated with a bit width such like `m[.]{32}`, representing a 32-bit memory access. However, only the (dynamic) analysis of the operator context reveals whether the memory value is interpreted as an integer value or a floating point value. The type inference adds more overhead to the instruction selector.

Complex Expressions and Temporary Registers

If a given IR effect can not be covered completely by a single tile, a temporary IR effect, with the uncovered sub-tree as the right-hand side, is generated. A temporary register is required for the left-hand side location of the temporary IR effect, forcing the covering tile into a register form. For example, the two tiles in figure 5.5 require the generation of a temporary IR effect as we have illustrated above. The instruction selector must thus allocate a suitable register when it generates such a temporary effect. If no register of the required type can be allocated, the instruction selector in its current implementation bails out, failing the translation of the entire trace.

During the covering of the temporary effect with tiles, the allocated temporary register can be re-allocated for uncovered sub-trees of the temporary IR effect. This has the effect that temporary results are computed into the same register, reducing the register pressure for the original IR effect.

Ternary ?: Operator

The only SSL operator that is not completely matched using our scheme is the ternary operator `?:` which is equivalent to the same C/C++ operator. This is because it introduces conditional branches into the otherwise linear control-flow of an IR trace. The effect

```
r[14] := ((m[102e3a8a]{1} = 0) ? 0xfe02a34 : 0xfe02a24)
```

in line 9 of our optimized example IR trace (cf. listing 5.11) implements parts of the dynamic operational semantics of the SPARC machine instruction `BNE`. Depending on the boolean value of the emulated pseudo register `%COND` (implemented at address `0x102e3a8a`) the emulated `%npc` register, mapped to the host machine register `r[14]`, gets either the value `0xfe02a34` (for the fallthrough case) or `0xfe02a24` (for the taken branch) assigned.

We extended the current implementation of the instruction selector manually. Additional host machine instructions that implement the control-flow of the `?:` operator are

```

1 0x1052178c:  lis    r20,4142          ;  m[102b95f2]{1} = 0
2 0x10521790:  ori    r20,r20,14986     ;
3 0x10521794:  lbz    r20,0(r20)        ;  load emulated flag
4 0x10521798:  andi.  r20,r20,1         ;  mask 1-bit value
5 0x1052179c:  cmpwi  cr3,r20,0
6 0x105217a0:  beq-   cr3,0x105217b0    ;  beq _then
7                                     ;  _else:
8 0x105217a4:  lis    r14,4064          ;  r[14] := 0xfe02a24
9 0x105217a8:  ori    r14,r14,10788
10 0x105217ac:  b      0x105217b8        ;  b _done
11                                     ;  _then:
12 0x105217b0:  lis    r14,4064          ;  r[14] := 0xfe02a38
13 0x105217b4:  ori    r14,r14,10804
14 0x105217b8:                                     ;  _done:

```

Listing 5.13:

The SSL operator `?:` introduces conditional control-flow to an IR effect. The PowerPC instructions implementing the control-flow are not selected through pattern matching.

inserted into the generated sequence of host machine instructions. The first operand of the `?:` operator, its condition, is matched as before using our scheme. Following the selected instructions, a conditional branch instruction is inserted. Before the second and the third operands are matched, they are rewritten into temporary IR effects that assign their respective value to the left-hand side location of the original effect. Instructions are first selected for the third operand effect, and then for the second one, such that the condition needs no negation. Following the third operand effect, an unconditional branch instruction is inserted, skipping over the implementation of the second operand effect. In such a way is the above IR effect rewritten and extended, resulting in the following logical pseudo SSL implementation:

```

    m[102e3a8a]{1} = 0    ; conditional expression
    beq _then             ; inserted/back-patched CTI
_else:
    r[14] := 0xfe02a24    ; third operand effect
    b _done              ; inserted/back-patched CTI
_then:
    r[14] := 0xfe02a34    ; second operand effect
_done:

```

The host machine instructions for the three operands of the `?:` operator are selected as described above. The SSL comparison operator is always matched with a host machine `cmp` instruction, and the conditional branch instruction is then selected according to the comparison operator. In the above example, the equality operator `=` is implemented by the `beq` instruction. Because either of the three operands may be a complex expression, the two inserted branch instructions must be back-patched when the instruction selection for the entire original IR effect is completed. The addresses of the labels are saved during instruction selection and emission, and the branch instructions are then adjusted to their correct target addresses when the actual addresses are determined.

Listing 5.13 shows the selected PowerPC instructions for the above example IR effect. The instructions in lines 1 to 5 are selected to implement the condition. The inserted `beq` instruction branches if the condition evaluates to true, or falls through otherwise. The code in lines 8-9 and 12-13 are generated to implement the third and second operand effect, respectively. The unconditional branch instruction in line 10 transfers control behind the implementation of the complete `?:` operator.

The `m[.]` Operator and Endianness Implementation

If both the source and host machines implement a different endianness for their data representation, then the load and store of emulated data must modify, i.e. byte swap, the data accordingly. Instruction selection for the memory access operator `m[.]` thus implements post-load and pre-store instructions which swap the bytes of the data.

5.10.4 Translating System Calls

As indicated earlier Walkabout does not support the interpretation of machine exceptions, except those used to invoke system calls. Walkabout interprets such exceptions by a custom system call mapper for the source machine that is implemented by hand-written classes, tailored for every host machine that the Walkabout emulator runs on.

The manual and intentional raising of an exception on SPARC (called trap) is implemented by dedicated instructions which generate such a machine trap, passing the actual trap number as the instruction's operand, and the system service number and its parameters through registers. For example, calls from a SPARC user application to the Solaris operating system are implemented by the trap number 136. The number of the invoked system service is passed in register `%g1`, and parameters for the system services are passed in registers `%o0` to `%o5`:

```
or  %g0, 1, %g1 ; syscall #1 = exit, no parameters
ta  8          ; trap 8 + 128 = 136
```

The example shows the invocation of the `exit` system service on a SPARC/Solaris machine, resulting in the normal termination of the invoking process. The number of the system service, in this case 1, is loaded into register `%g1` and an unconditional trap instruction then generates the machine trap number 136 that is caught and handled by the operating system, thus invoking the actual system service.

The operational semantics of instructions that raise an exception is specified using the SSL operator `TRAP`. For example, the unconditional SPARC trap instruction `ta` above has the following operational semantics defined in SPARC's SSL specification:

```
TAimm rs1, simm13
*32* %g0 := 0
TRAP((1 = 1), r[rs1] + sgnex(13,32,simm13) + 128)
*32* %pc := %npc
*32* %npc := %npc + 4;
```

```

1 0x104ea7b8: lis      r3,0          ; r[3] := 1
2 0x104ea7bc: ori      r3,r3,1
3 0x104ea7c0: lis      r4,0          ; r[4] := 136
4 0x104ea7c4: ori      r4,r4,136
5 0x104ea7c8: lis      r0,4104       ; Walkabout::trap()
6 0x104ea7cc: ori      r0,r0,25304
7 0x104ea7d0: mtlr     r0
8 0x104ea7d4: blrl

```

Listing 5.14:

Generated PowerPC host instructions that emulate a system call for a SPARC/Solaris source program. The generated dynamic function calls back into Walkabout's emulator core to invoke its `Walkabout::trap()` function.

The first operand of the TRAP operator is a condition. If the condition evaluates to true, then the trap, whose number is given as the second operand, is invoked. The Walkabout emulator implements trap instructions by the custom core function

```
void Walkabout::trap(bool cond, sint32_t trapnum);
```

that maps the emulated system service to the host operating system (cf. section 1.7 for more details).

Therefore, a TRAP operator in a given IR effect is compiled into a host machine call to that core function. The calling convention for the SysV ABI on PowerPC host machines, for instance, requires that parameters are passed in designated registers and over the stack. The two parameters for the `Walkabout::trap()` function are passed in the PowerPC registers `r3` and `r4` such that the given IR TRAP effect must be rewritten accordingly:

```
*32* r[3] := (1 = 1)
*32* r[4] := r[rs1] + sgnex(13,32,simm13) + 128
```

followed by a manually inserted call to Walkabout's system call mapper.

Listing 5.14 shows the generated PowerPC instructions for the interpreted system call to the `exit` system service from the above example. The instructions in lines 1-2 and 3-4 are selected using the instruction selector, and load the already evaluated trap condition and the trap number, the actual parameter values for the `Walkabout::trap()` function, into the designated host machine registers. The call to Walkabout's system call mapper is then implemented by an absolute branch-and-link instruction sequence in lines 5-8.

5.10.5 Instruction Emission

The encoding and emission of the selected host machine instructions is implemented by an instruction encoder that is generated from the SLED specification for the syntax of host machine instructions. The New Jersey Machine Code Toolkit [RF94] provides the required tools, libraries and some example specification files to build machine instruction decoders and encoders. The Toolkit itself was originally designed and implemented for

the use by static code emitters, and therefore adds significant overhead to our dynamic framework. However, the emission routines of the Toolkit can be replaced by custom ones, such that most of the static overhead is removed.

The generated instruction encoder works as follows. For every instruction defined in a SLED syntax specification, the encoder generator creates a C function that takes the operands of the instruction, i.e. the encoded registers or immediate values, as actual parameter values. The operands may optionally be checked for validity by the Toolkit, and they are then, together with the opcode, encoded and emitted into a custom buffer. This buffer is implemented by a list of linked byte arrays, providing a resizable buffer for code emission. The emission of a long enough trace of host machine instructions therefore causes a fragmentation of the emitted trace into multiple arrays.

When all effects of an IR trace are emitted into the fragmented buffer, the content of that buffer is merged and relocated into the body of Yirr-Ma's dynamic function, and the dynamic function is then returned to the Collector.

Listing 5.15 shows excerpts of the trace of PowerPC host machine instructions generated from the optimized IR trace. Because instructions are selected per effect, some code redundancy is generated for the trace's epilogue. Using a host machine specific peephole optimizer, for example, the target code can be improved further. Lines 2-4 implement the only prologue effect that loads the value of the emulated %10 register into host machine register r19. Note that through the application of the PowerPC idiom

```
*32* rn := m[addr] -> lis rn,addr@[31:16] , lwz rn,addr@[15:0](rn)
```

the instruction selector would generate one instruction less, provided that the given 32-bit operand *addr* can be decomposed into an unsigned and a signed 16-bit value. Lines 6-9 implement the interpreted LDreg instruction, and lines 13-17 compute the emulated %NF register. The `andi` instruction in line 16 is selected to implement the 1-bit data type of the typed memory location. Lines 23-35 implement the conditional branch instruction BNE, line 38 implements the emulated INC instruction and lines 36-38 and 40 implement the computation of the emulated %g0, %pc and %npc registers. Finally, the instructions in lines 41-43 are selected to store register r18 to the emulated %tmp register, followed by the remainder of the trace epilogue.

5.11 Generation of the Instruction Selector from a SSL Specification

An efficient instruction selector must search the set of available tiles in preferably sub-linear time. We therefore developed an algorithm that *generates* a maximal munch algorithm which searches a given set of tiles in logarithmic time using a generated search tree. The search tree is defined over the set of *extended tiles*: tiles whose leaves

```

1      ; r[19] := m[102e39c8]{32}
2 0x105216d8: lis      r20,4142
3 0x105216dc: ori      r20,r20,14792
4 0x105216e0: lwz      r19,0(r20)
5      ; r[16] := m[r[19] + 0x30032000]
6 0x105216e4: lis      r20,12291
7 0x105216e8: ori      r20,r20,8192
8 0x105216ec: add      r20,r19,r20
9 0x105216f0: lwz      r16,0(r20)
10     ; r[15] := r[16]
11 0x105216f4: ori      r15,r16,0
12     ; m[102e3aa0]{1} := (r[15] >>A 31)
13 0x105216f8: srawi   r20,r15,31
14 0x105216fc: lis      r21,4142
15 0x10521700: ori      r21,r21,15008
16 0x10521704: andi.   r20,r20,1
17 0x10521708: stb     r20,0(r21)
18 ... ; m[102e3ab4]{1} := ((r[15] = 0) ? 1 : 0)
19 ... ; m[102e3aa1]{1} := 0
20 ... ; m[102e3a89]{1} := 0
21 ... ; m[102e3a8a]{1} := ~ m[102e3ab4]{1}
22     ; m[102e3a8a]{1} = 0 ? ...
23 0x1052178c: lis      r20,4142
24 0x10521790: ori      r20,r20,14986
25 0x10521794: lbz     r20,0(r20)
26 0x10521798: andi.   r20,r20,1
27 0x1052179c: cmpwi   cr3,r20,0
28 0x105217a0: beq-    cr3,0x105217b0
29     ; r[14] := 0xfe02a24
30 0x105217a4: lis      r14,4064
31 0x105217a8: ori      r14,r14,10788
32 0x105217ac: b       0x105217b8
33     ; r[14] := 0xfe02a34
34 0x105217b0: lis      r14,4064
35 0x105217b4: ori      r14,r14,10804
36 0x105217b8: subf    r15,r15,r15 ; r[15] := (r[15] - r[15])
37 0x105217bc: ori      r18,r19,0 ; r[18] := r[19]
38 0x105217c0: addi    r19,r19,4 ; r[19] := (r[19] + 4)
39 0x105217c4: ori      r17,r14,0 ; r[17] := r[14]
40 0x105217c8: addi    r14,r14,4 ; r[14] := (r[14] + 4)
41 0x105217cc: lis      r20,4142 ; m[102e1d70]{32} := r[18]
42 0x105217d0: ori      r20,r20,7536
43 0x105217d4: stw     r18,0(r20)
44 ... ; m[102e3988]{32} := 0
45 ... ; m[102e39c8]{32} := r[19]
46 ... ; m[102e39cc]{32} := r[16]
47 ... ; m[102e3ab8]{32} := r[14]
48 ... ; m[102e3abc]{32} := r[17]

```

Listing 5.15:

The generated trace of PowerPC instructions for the optimized IR trace from listing 5.11. Selected instructions implement individual effects. Note that the quality of the generated instruction trace still holds much potential for improvement through host machine specific optimizations. Section A.4 in appendix A shows the complete trace of host machine instructions.

are either operand symbols in V_s or variables X_n for other tiles.

The root of the search tree is the most general extended tile, a single variable X_n , which is stepwise refined towards the leaves of the search tree that represent actual tiles in T_{Δ, V_s} from the SSL specification. The top-down refinement is formally defined by a non-deterministic rewrite relation. Given an SSL specification, i.e. the set of tiles, the search tree is constructed bottom-up using a deterministic reduction relation. Actual tiles are reduced to extended tiles until the most general extended tile X_n , the root of the search tree, is produced.

The problem of generating such an instruction selector and our solution are in some respects similar to that described in [Cat80], where the author introduces the derivation of a code generating compiler backend from a machine specification file. Whereas the therein proposed specification language extends and resembles features of SSL, both the generation and implementation of the instruction selector are very different. A formal representation of a machine is extracted from a machine description file, providing the semantics, the costs and binary encoding for machine instructions. The generated instruction selector is table driven. The available instructions are represented by a set of tree productions, where the left-hand side is a pattern and the right-hand side the actual instruction encoding information. Instruction selection is a recursive and goal-driven process, based on the derived information stored in the selector's table. Given an IR tree, the initial goal tree, the instruction selector rewrites the tree and, if required, sub-trees until the entire tree is rewritten into a sequence of instruction trees. In this context, rewriting a tree into an instruction tree is somewhat similar to the tiling of a tree in our approach. The generated sequence of instruction trees is then encoded using the right-hand sides of the tree productions. Rewriting a given tree is specified by a set of axioms, given by the writer of the machine description file in the first place. Interestingly, the completeness of the set of axioms defines the success of the instruction selection process, much like in our case.

In the beginning of this section we define the set of extended tiles and the non-deterministic top-down rewrite relation for extended tiles. We then introduce the deterministic bottom-up reduction relation for the set of actual tiles and extended tiles, producing the search tree from a given SSL specification. Finally, we show the generation of the instruction selector from the derived search tree.

Although we give a formal specification of the algorithms and their data types, we were unfortunately unable to implement the instruction selector generator.

5.11.1 Extended Tiles

Let $X = \{X_1, X_2, \dots\}$ be the set of variables of type T_{Δ, V_X} where $V_X = V_s \cup X$, and V_X denotes the set of formal operand symbols for tiles and variables in X . The set T_{Δ, V_X}

of extended tiles is the smallest set $T_{\Delta, V_X} \subseteq (\Delta \cup V_X \cup \{(\cdot, \cdot)\})^*$ such that

- $X \subseteq T_{\Delta, V_X}$, and
- if $\delta \in \Delta^{(n)}$ with $n \geq 1$ and $t_1, \dots, t_n \in T_{\Delta, V_X}$, then $\delta(t_1, \dots, t_n) \in T_{\Delta, V_X}$ and for every (if any) $t_i \in X$ with $i \in \{1, \dots, n\}$, $t_{i+k} \in X$ with $1 \leq k \leq n - 1$ holds.

Note that $T_{\Delta, V_s} \subset T_{\Delta, V_X}$ holds. The set T_{Δ, V_X} denotes the set of extended tiles, where leaf nodes are either formal operand symbols in V_s or variables in X . Furthermore, for any given extended tile, the variables $X_n \in X$ (if any) are always the rightmost subtrees. Two examples of such extended tiles are $:=^{(2)}(X_1, X_2)$ or $:=^{(2)}(m^{(1)}(X_1), X_2)$. In contrast, $:=^{(2)}(X_1, m^{(1)}(X_2))$ is not a valid extended tile because right of the subtree X_1 is the sub-tree $m^{(1)}(X_2) \notin X$. As a convention and to conform with the implementation of IR effects by SSL, immediate values in \mathbb{N} are always at the right-hand side of commutative operators.

5.11.2 Rewriting of Extended Tiles

A search tree over extended tiles is defined by the rewrite relation for extended tiles. Let $t \in T_{\Delta, V_X}$ be an extended tile. The non-deterministic rewrite relation $\Rightarrow \subseteq T_{\Delta, V_X} \times T_{\Delta, V_X}$ of an extended tile $t = \delta^{(n)}(t_1, \dots, t_i, \dots, t_n)$ with $1 \leq i \leq n$ is recursively defined as follows:

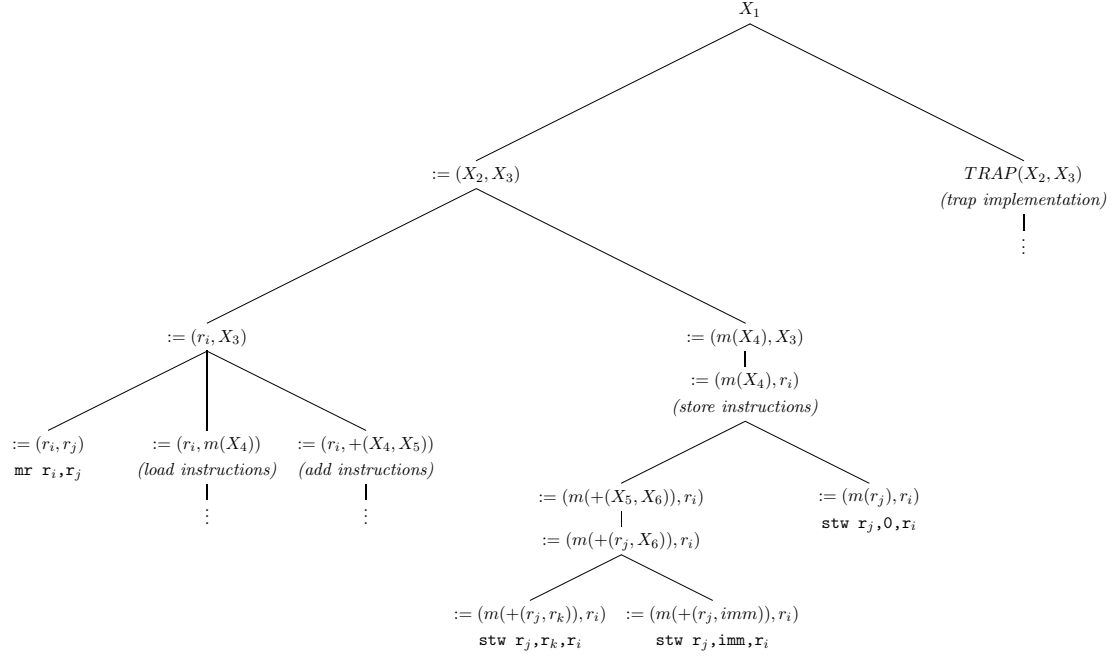
- if $t_i \in X$ and $t_{i-1} \notin X$ then $t \Rightarrow t'$ with $t' = \delta^{(n)}(t_1, \dots, t'_i, \dots, t_n)$ and $t'_i \in T_{\Delta, V_X}$ (i.e. the left-most variable $t_i = X_n$ (if any) rewrites to an extended tile t'_i),
- if $t_i \in (T_{\Delta, V_X} - T_{\Delta, V_s})$ and $t_{i-1} \in T_{\Delta, V_s}$ then $t \Rightarrow t'$ with $t' = \delta^{(n)}(t_1, \dots, t'_i, \dots, t_n)$ and $t_i \Rightarrow t'_i$ (i.e. recursively rewrite the left-most extended tile t_i that is no actual tile into another extended tile t'_i).

Rewriting an extended tile thus replaces variables X_n of the extended tile with other extended tiles from left-to-right and top-down, until all variables of the original extended tile have been replaced by actual tiles. The recursive rewriting of a given extended tile $t \in T_{\Delta, V_X}$ results in a sequence of extended tiles $t'_1, \dots, t'_n \in T_{\Delta, V_X}$ that eventually terminates with an actual tile $t' \in T_{\Delta, V_s}$. For example,

$$X_1 \Rightarrow :=^{(2)}(X_2, X_3) \Rightarrow :=^{(2)}(r_n, X_4) \Rightarrow :=^{(2)}(r_n, r_m)$$

is a possible sequence of rewrite relations of the most general extended tile X_1 into an actual tile $:=^{(2)}(r_n, r_m) \in T_{\Delta, V_s}$.

The rewrite relation \Rightarrow is non-deterministic because more than one extended tiles may rewrite a selected variable X_n at any time. All possible rewrites of an extended tile t are therefore represented as a (infinite) rewrite tree. The actual search tree, which the


Figure 5.6:

A possible rewrite tree for extended tiles, pruned to a valid search tree. The non-deterministic rewrite relation \Rightarrow between two extended tiles (nodes or leaves of the rewrite tree) is represented by a downward edge. The left-most and top-most variable X_n of any extended tile is rewritten into another extended tile. The leaves of the rewrite tree are actual tiles, representing primary effects of machine instructions from the SSL specification file.

maximal muncher is generated from, is in fact a pruned version of the rewrite tree for X_n , such that the search tree is finite and contains the shortest paths from its root to the leaves. The nodes of the rewrite tree are extended tiles in T_{Δ, V_X} , leaves are actual tiles T_{Δ, V_s} , and edges between two nodes or a node and a leaf represent the rewrite relation \Rightarrow .

Figure 5.6 shows a possible rewrite tree for the variable X_1 . Note that this rewrite tree also represents a valid search tree. Two rewrite relations of X_1 result in $:= (X_2, X_3)$ and $TRAP(X_2, X_3)$ which are then the roots of two other rewrite trees. With every rewrite step, a single variable X_n is selected from left-to-right and rewritten into an extended tile in T_{Δ, V_X} . An extended tile $t \in T_{\Delta, V_X}$ thus may eventually rewrite into an actual tile $t' \in T_{\Delta, V_s}$ which, in turn, is the primary effect of an instruction in a given SSL specification. All primary effects of the SSL specification define the leaves of the rewrite tree.

5.11.3 Construction of the Search Tree

A rewrite tree with the most general extended tile X_n as a root node and leaf nodes that all are actual tiles from a given SSL specification file is the search tree that our instruction selector is generated from. This search tree is constructed bottom-up by reducing a given set of actual tiles into common extended tiles, and then recursively reducing the extended tiles and merging reduction sequences if possible. More precisely, the primary effect of an instruction in a SSL specification file, the actual tile in T_{Δ, V_s} , is reduced to extended tiles T_{Δ, V_X} until a reduction step produces an extended tile that is equal to an extended tile from another reduction sequence. In that case, the two reduction sequences are merged into a single one, thus forming a common node in the search tree. Finally, the most general tile X_n forms the root of the search tree and the end of all reduction sequences.

The search tree is constructed from an SSL specification by *reducing* the primary effects of all specified instructions to the most general extended tile, the variable X_n . The deterministic reduction relation $\Rightarrow^R \subseteq T_{\Delta, V_X} \times T_{\Delta, V_X}$ of an extended tile $t = \delta^{(n)}(t_1, \dots, t_i, \dots, t_n)$ with $1 \leq i \leq n$ is defined as follows:

- if $t = \delta^{(n)}(X_1, \dots, X_n)$ then $t \Rightarrow^R X_{n+1}$ (i.e. if all sub-trees of the extended tile are variables in X , the extended tile is reduced to a single variable, the root of the search tree),
- if $t_i \in (T_{\Delta, V_X} - X)$ and $t_i \neq \delta^{(m)}(X_1, \dots, X_m)$ and $t_{i+1} \in X$ then $t \Rightarrow^R t'$ with $t' = \delta^{(n)}(t_1, \dots, t'_i, \dots, t_n)$ and $t_i \Rightarrow^R t'_i$ (i.e. recursively reduce the right-most extended tile that has yet reducible sub-trees),
- if $(t_i \in V_s$ or $t_i = \delta^{(n)}(X_1, \dots, X_n))$ and $t_{i+1} \in X$ then $t \Rightarrow^R t'$ with $t' = \delta^{(n)}(t_1, \dots, t'_i, \dots, t_n)$ and $t'_i = X_{n+1}$ (i.e. reduce the right-most formal operand symbol or sub-tree with variables only to a single variable).

Similar to the rewrite sequence of extended tiles, the recursive reduction of a given actual tile $t \in T_{\Delta, V_s}$ produces a sequence of extended tiles $t'_1, \dots, t'_n \in T_{\Delta, V_X}$ that eventually terminates with the most general tile $t' \in X$:

$$:=^{(2)}(r_i, r_j) \Rightarrow^{R:=^{(2)}}(r_i, X_1) \Rightarrow^{R:=^{(2)}}(X_2, X_1) \Rightarrow^R X_3.$$

The different reduction sequences of all given actual tiles in a SSL specification file are paths from a leaf to the root node of the search tree (cf. figure 5.6). The search tree with leaves that are all actual tiles of a given SSL specification represents the search tree that the instruction selector is generated from.

5.11.4 Generation of the Instruction Selector

A given search tree for a set of actual tiles allows for the generation of an instruction selector that implements a maximal munch algorithm: the maximal muncher consumes a given IR effect top-down by matching nodes of the IR effect with variables X_n in the extended tiles at the nodes of the search tree. For example, the IR effect $:=^{(2)}(r_{16}, m^{(1)}(+^{(2)}(r_{19}, 0x30032000)))$ from listing 5.11 matches both extended tiles of the rewrite relation $X_1 \Rightarrow :=^{(2)}(X_2, X_3)$, with $X_1 = :=^{(2)}(r_{16}, m^{(1)}(+^{(2)}(r_{19}, 0x30032000)))$, $X_2 = r_{16}$ and $X_3 = m^{(1)}(+^{(2)}(r_{19}, 0x30032000))$. Rewriting the left-most variable X_2 produces the next extended tile: $:=^{(2)}(X_2, X_3) \Rightarrow :=^{(2)}(r_i, X_4)$ with the binding of the operand symbol $r_i = r_{16}$. The next rewrite step $:=^{(2)}(r_i, X_4) \Rightarrow :=^{(2)}(r_i, m^{(1)}(X_5))$ still matches the IR effect with $X_5 = +^{(2)}(r_{19}, 0x30032000)$, and so forth until an actual tile at the leaf of the search tree partially or completely covers the IR effect, and the tile's operand symbols are bound to actual operands of the IR effect.

The descent into the IR effect is thus guided by a descent into the search tree: the extended tile of a rewrite step must at any time match the IR effect. The search tree and the descent into the same is implemented by nested `switch` statements. The structure of the nesting is inherent in the search tree, and the instruction selector generator produces the `switch` statements by descending into the search tree. Listing 5.16 shows the generated maximal muncher for the search tree in figure 5.6.

Starting with the root of the search tree, a `switch` is generated for the root operator of the given IR effect (line 3). This operator, according to the two possible rewrites of the variable X_1 , is either an assignment operator `:=` or a trap operator `TRAP`, thus two `case` branches are produced (lines 4 and 51). For the assignment operator, the first operand (i.e. the left-most variable X_2) is either a register or a memory reference, thus a nested `switch` statement (line 5) with two `case` branches is produced (lines 6 and 28). If a variable X_n rewrites into a register $r_n \in V$, the related `case` branch falls through to the `default` case (line 29). The default case handles a partial cover of the tile by producing and consuming a temporary effect (line 30-33), that has a temporary register as its left-hand side location. By descending into the search tree, the instruction selector generator produces a set of nested `switch` statements that implement the search over the set of all available actual tiles in a given SSL specification file.

If a tile covers a given IR effect completely, the leaf symbols of the tile are bound to actual registers or immediate values, and the host machine instruction associated with the tile is emitted (e.g. line 24 for a store instruction or line 47 for a register-copy instruction). The instruction emission functions are generated from the SLED specification for the host machine using the NJMC Toolkit.

```

1  munch(t:  $T_{\Delta, V}$ ): void // t is an IR effect
2  {
3      switch (t.operator) { // operand is the root of an IR effect
4          case :=: // assignment effects: loads, stores, ...
5              switch (t.child1) { //  $X_2$ 
6                  case m: // store instructions
7                      switch (t.child2) { //  $X_3$ 
8                          case  $r_i$ :
9                              default:
10                             if (t.child2 !=  $r_i$ ) { // partial cover
11                                  $t_{tmp} = tmp\_ir(r_{tmp}, t.child_2)$ ;
12                                 munch( $t_{tmp}$ );
13                             }
14                             switch (t.child1.child1) {
15                                 case +:
16                                     ...
17                                     break;
18                                 case  $r_j$ :
19                                 default:
20                                 if (t.child2 !=  $r_i$ ) { // partial cover
21                                      $t_{tmp} = tmp\_ir(r_{tmp}, t.child_2)$ ;
22                                     munch( $t_{tmp}$ );
23                                 }
24                                 stw( $r_j, 0, r_i$ ); // emit stw instruction
25                             }
26                         }
27                         break;
28                     case  $r_i$ : // assignment to registers
29                     default:
30                     if (t.child1 !=  $r_i$ ) { // partial cover
31                          $t_{tmp} = tmp\_ir(r_{tmp}, t.child_1)$ ;
32                         munch( $t_{tmp}$ );
33                     }
34                     switch (t.child2) { //  $X_3$ 
35                         case m: // load instructions
36                             ...
37                             break;
38                         case +: // add instructions
39                             ...
40                             break;
41                         case  $r_j$ :
42                         default:
43                         if (t.child1 !=  $r_i$ ) { // partial cover
44                              $t_{tmp} = tmp\_ir(r_{tmp}, t.child_1)$ ;
45                             munch( $t_{tmp}$ );
46                         }
47                         mr( $r_i, r_j$ ); // emit mr instruction
48                     }
49                 }
50                 break;
51             case TRAP: // trap implementation
52                 ...
53                 break;
54         }
55     }

```

Listing 5.16:

Generated maximal muncher for the search tree in figure 5.6. The nesting of switch statements is inherent in the generated search tree, and the maximal muncher can be produced by traversing that tree.

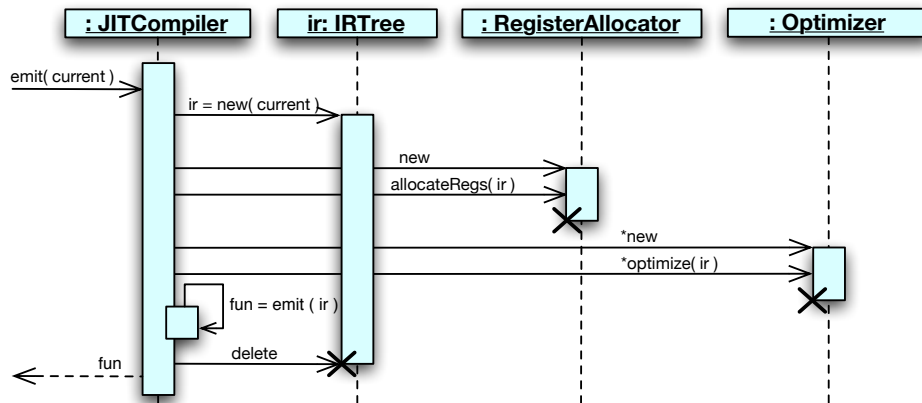


Figure 5.7:

Sequence diagram of a successful translation of a IR trace. The IR trace is generated from the collected semantic information of the interpreted trace `current`, and is then optionally mapped to the host machine state and further optimized. Host machine instructions are then selected for the IR trace, and emitted in form of a dynamic function. The generated dynamic function is then returned to Yirr-Ma's Collector.

5.12 Implementation

Our specification-driven dynamic binary translator implements Yirr-Ma's Emitter interface, as illustrated in the class diagram 5.2 on page 86. It takes the collected semantic information of an interpreted source machine trace as actual parameter, and produces a dynamic function. The sequence diagram in figure 5.7 illustrates the successful generation of an IR trace, its optimization and translation into host machine instructions.

In the following we discuss the implementation of the different components of our specification-driven dynamic binary translator: the interface for the machine dictionaries, the implementations of both source and host machine dictionary, the internal representation of the IR, and finally we talk about the implementation of the different optimizers, the instruction selector and encoder.

5.12.1 The Dynamic Machine Dictionaries

Our specification-driven dynamic binary translator uses two machine dependent dynamic dictionaries: the source machine dictionary contains the dynamic operational semantics of all interpreted source machine instructions, and the host machine dictionary contains the state definition of the host machine. The actual implementation of the machine dictionaries is generic, i.e. independent from the machine they represent, and it is parameterized by a SSL machine specification file. As can be seen in the class diagram in figure 5.8, both machine dictionaries inherit from the class `Dictionary` which is a sub-class of UQBT's `RTLInstDict` dictionary class.

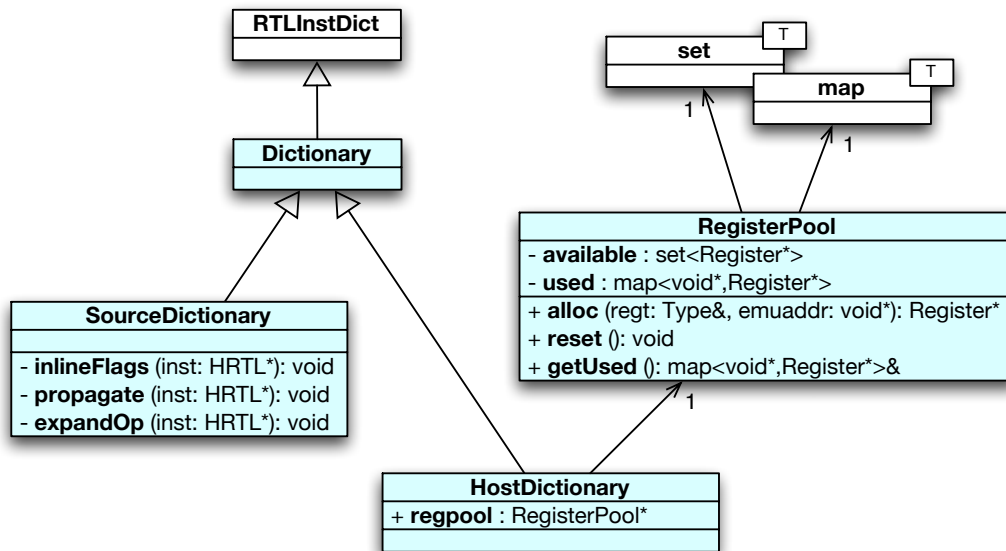


Figure 5.8:

Class diagram of Yirr-Ma's dynamic machine dictionaries. For both source and host machines Yirr-Ma uses a machine dictionary that contains the required information in a suitable form. Machine dictionaries inherit from UQBT's `RTLInstDict` class, a representation of a static SSL machine specification file.

An UQBT machine dictionary is an abstract data type initialized from the SSL specification file for a machine. Similar to the specification file, so contains the machine dictionary the abstract and symbolic state definition of the specified machine and the static operational semantics of machine instructions. Given a `Dictionary` object, the machine dependent dictionaries then adapt the static information for the source and host machines, respectively.

The Source Machine Dictionary

As mentioned previously, the source machine dictionary contains the dynamic operational semantics for every interpreted source machine instruction. Upon object creation, the constructor of the source machine dictionary class `SourceDictionary` initializes the static UQBT dictionary. It then rewrites the static operational semantics of every specified machine instruction for the actual runtime environment that the Walkabout emulator implements (cf. section 5.6.1). Three methods are used to rewrite the static operational semantics of one instruction for the dynamic runtime environment. Each method takes the static operational semantics of a source machine instruction, a list of register transfers expressions, as parameter and modifies that semantics:

- `SourceDictionary::inlineFlags()` inlines the body of a flag macro (if any) into the specification of the instruction

- `SourceDictionary::propagate()` propagates the addresses of the emulated machine registers into the instruction's effects, therewith replacing the occurrences of symbolic register names with the actual host machine addresses of their respective implementation, and also adds the base address of the emulated memory segment to the specification of interpreted memory access instructions
- `SourceDictionary::expandOp()` expands complex SSL operators into a sequence of basic operators.

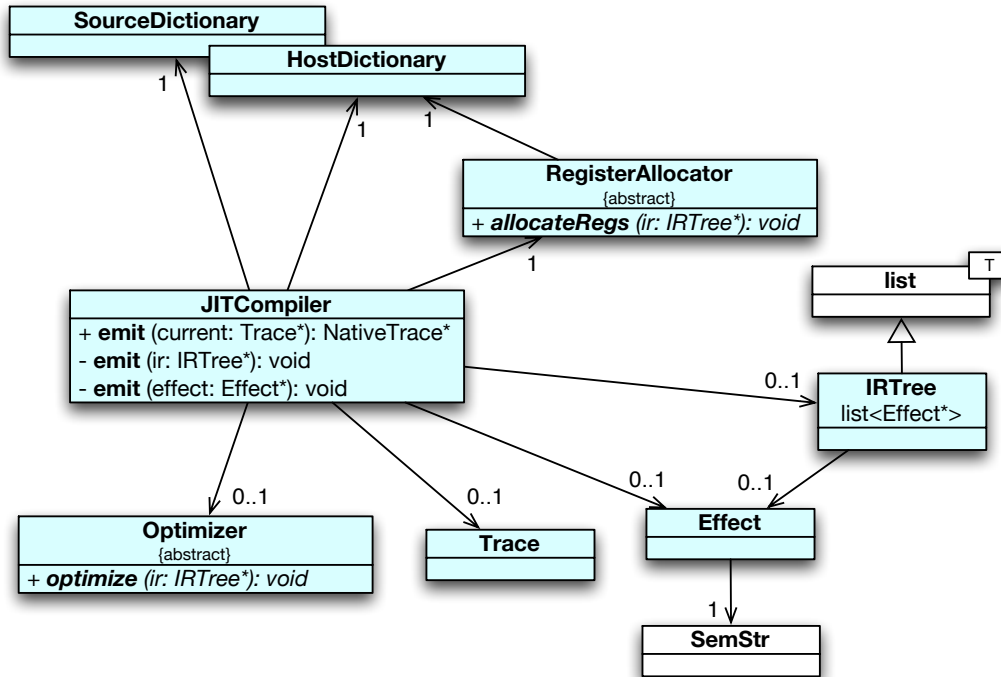
With one iteration over all machine instructions in the dictionary, the three methods are applied to each instruction individually, thus rewriting an instruction's static operational semantics into its dynamic form. After the instantiation and initialization of the source machine dictionary, Yirr-Ma's translator creates the host machine dictionary.

The Host Machine Dictionary and the Register Pool

The host machine dictionary is implemented by the `HostDictionary` class. It contains a definition of the host machine state that is available and relevant to user programs. Reserved registers like the `%pc` register, for example, or other registers dedicated to the use by the host operating system are specified in SSL, but they are not part of the available state that the host machine dictionary represents. The translator's state mapper requires this state information in order to map the emulated machine registers to actual host machine registers. The `HostDictionary` object contains a state definition retrieved from a static SSL specification file, and the constructor of the `HostDictionary` class builds this pool of available registers from that static SSL state definition.

As shown in section 5.5, we annotated a SSL state specification and marked the available host machine registers. Upon construction of the dictionary object, the host machine dictionary scans the state definition in the dictionary and collects the annotated registers into a register pool. Figure 5.8 also shows the class diagram of the register pool.

All available and relevant host machine registers are stored in the register pool. The register pool is implemented by the `RegisterPool` class which has a private set of the available host machine registers. During the state mapping of an IR trace, or during the instruction selection for complex IR effects, a host machine register is allocated from the register pool using its `RegisterPool::alloc()` method. The method takes the required register type and the host address of the emulated source machine register as parameters, and returns either an actual host machine register or `null`. An allocated register is then moved from the set `available` of available registers to the map `used`, thus performing the trace-global state mapping from emulated source machine registers to actual host machine registers. A call to the `RegisterPool::reset()` method resets an existing state mapping by moving all used registers back into the set of available registers.

**Figure 5.9:**

Class diagram of Yirr-Ma's specification-driven dynamic binary translator. It contains persistent references to the two machine dictionaries and the state mapper RegisterAllocator. The different optimizers and all intermediate data structures are constructed and destructed on demand.

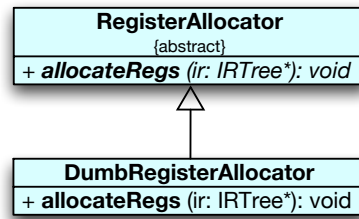
5.12.2 Intermediate Representation and Dynamic Optimizers

The dynamic binary translator itself is implemented by one single class JITCompiler that implements Yirr-Ma's Emitter interface. Figure 5.9 shows the class diagram of the JITCompiler class and its associated classes. The public method `Emitter::emit(Trace*)` is invoked by Yirr-Ma's Collector. It takes a trace of collected semantic information as its actual parameter value and returns a dynamic function which implements the given trace. In the following we introduce the implementation of the IR, its generation and optimization, and the instruction selector.

The Intermediate Representation

The generated IR is implemented by the IRTree class which instantiates the STL `list` template class with the `Effect*` data type. The `Effect` class, in turn, is a wrapper class for semantic strings provided by the UQBT framework.

Semantic strings are used by the UQBT static binary translation framework [CEe99] to represent weakly typed register transfers: expression trees over SSL operators, symbolic machine registers and immediate values representing the static operational semantics of

**Figure 5.10:**

Class diagram of Yirr-Ma's state mapper. The naive state mapper `DumbRegisterAllocator` implements the `RegisterAllocator` interface. It takes the IR of a source machine trace and allocates host machine registers for emulated source machine registers that are at the left-hand side of assignment effects.

machine instructions. Semantic strings are implemented by a list of integers, where each integer is an index into a semantic table, another of UQBT's data structures that holds information about all SSL operators and, to some extent, their operands. In addition to the data type implementation, UQBT provides a large number of methods to manipulate semantic strings. Owing to the list implementation of what really is a tree data type, locating sub-trees and other operations involving tree traversal over semantic strings have linear complexity plus a constant overhead for lookups of the semantic table. The poor performance of UQBT's semantic strings is therefore transferred into the dynamic Yirr-Ma framework.

The State Mapper

Like most of Yirr-Ma's components, so is the state mapper implemented by an abstract class, and thus defines a software interface for different experimental implementations, as can be seen in the class diagram in figure 5.10.

We implemented a naive state mapper by the `DumbRegisterAllocator` class. Given a list of IR effects, the method `DumbRegisterAllocator::allocateRegs()` generates a trace-global register mapping in a first scan over the IR trace. For every typed memory location $m[a]_{type}$ on the left-hand side of an IR effect, the method requests a register of equivalent type from the host machine dictionary. In a second scan over the IR trace, the method searches all IR effects for occurrences of the mapped source machine registers and replaces them with actual host machine registers, producing an IR trace with an optimized state mapping. The state mapper also constructs and adds the prologue and epilogue to the given IR trace.

Dynamic Optimizers

Mapping the emulated source machine registers of the IR trace to actual host machine registers is a first state optimization. Given this mapped IR, further machine independent and, to some extent, source machine dependent optimizations are applied to the

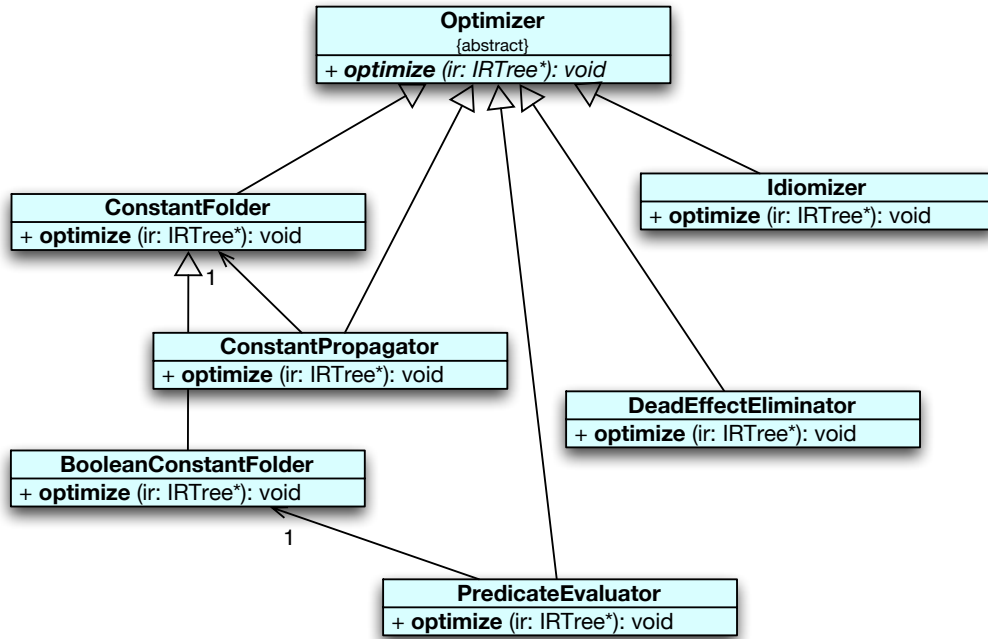


Figure 5.11:

The abstract class `Optimizer` defines the interface for dynamic optimizers that improve a given intermediate representation. Sub-classes then implement different dynamic optimization techniques.

given IR trace.

As shown in the class diagram in figure 5.11, dynamic optimizers for Yirr-Ma are defined by the `Optimizer` interface. An implementation of a dynamic optimizer thus takes the IR trace, i.e. a pointer to an `IRTree*` object, and directly rewrites that IR trace. Different dynamic optimizations may be applied individually, or in combination with one another. Currently, constant propagation and constant folding are combined and applied in a single iteration over the IR trace, whereas dead effect elimination and the customization of an IR with host machine dependent idioms are applied in separate steps.

Each of the dynamic optimization techniques is implemented by a single class. Because the traversal methods are mostly recursive and operate on the IR effects, they directly manipulate UQBT’s semantic strings and therefore receive a drastic performance penalty.

Instruction Selector

The actual instruction selection is implemented by the `JITCompiler` class itself. After an IR trace is generated, mapped and optimized, the host machine instructions are selected to implement the individual effects of the IR. The method `JITCompiler::emit(IRTree*)`

traverses the list of effects and invokes the `JITCompiler::emit(Effect*)` method for each individual IR effect. As shown in previous sections, IR effects are expression trees in $T_{\Delta, V'}$ and the host machine instructions are selected using a maximal munch instruction selector introduced in section 5.10.1. Unfortunately, because of the time constraints towards the end of our research, we did not implement the instruction selector generator. Instead we implemented the maximal muncher by hand, following our specified generation algorithm. Selected host machine instructions are copied into the body of the dynamic function which is then returned to Yirr-Ma's Collector.

5.13 Experimental Results

Our experimental results are so far sparse. In addition to the lack of sufficient time and funding resources to complete our research, this has the following reasons:

- as mentioned before our instruction selector is hand-written: its current implementation supports only instructions with integer operands, and it bails out when it encounters floating point operations or other unsupported SSL operators in the IR trace
- strongly typed IR expressions are essential for the selection of machine instructions, but they are poorly supported by UQBT's semantic strings and therefore in Yirr-Ma's IR; insufficient type information causes conflicts between (falsely or because of an incorrect assumption) selected host machine instructions for IR effects and the generated instruction encoding functions, therewith forcing some benchmark programs to abort prematurely
- internal limitations of the Walkabout machine emulator's implementation causes many larger emulated programs to run out of memory: the emulator interprets the dynamic linker to bind and start a source machine program which, together with the two machine dictionaries, their own and other expensive data structures, poses a serious memory constraint on the runtime environment.

We partially resolved the last problem by interpreting only toy benchmark programs that are statically linked. The tested programs are `banner`, the two recursive programs `fac` to compute the factorial number of 10 and `fib` to compute the Fibonacci number of 30, and `qsort`, an implementation of the recursive quicksort over an array of strings.

Table 5.2 shows the runtime performance results for these four toy benchmark programs. The first two rows compare plain interpretation using a Walkabout machine emulator with Yirr-Ma's dynamic binary translator. It is obvious that the introduced overhead, at least for these toy benchmarks, is not at all amortized and causes an average slowdown by a factor of 2 to 35. The two bottom rows show the time spent on

	banner	fac(10)	fib(30)	qsort
Walkabout	0m 0.2s	0m 0.2s	0m 8.6s	0m 0.1s
Walkabout/Yirr-Ma-DBT	0m 6.9s	0m 3.4s	0m 16.2s	0m 6.1s
Dictionary initialization	1.77s	1.78s	1.77s	1.76s
Optimization overhead	2.73s	0.29s	0.28s	2.23s

Table 5.2:

Experimental performance results for Yirr-Ma's specification-driven dynamic binary translator for some toy benchmark programs. The slowdown, compared to plain interpretation, is depressingly humongous.

	banner	fac(10)	fib(30)	qsort
Number of dynamic functions	103	20	17	74
Bailed out	13	1	2	12
Shortest	1/47	1/52	1/52	1/52
Average	4/129	2/105	2/101	4/131
Longest	20/282	8/175	5/171	20/282
Cache size in kBytes	53.428	8.444	6.896	39.024

Table 5.3:

Statistical information about the size and number of host machine instructions for the generated dynamic functions. The quality of translated instructions has much improved, compared to Yirr-Ma's dynamic partial Inliner (cf. page 74).

the initialization of the dynamic machine dictionaries and the time consumed by the actual dynamic binary translation, including the application of all available dynamic optimization techniques.

Table 5.3 gives information about the generated dynamic functions for the source programs. As can be seen in the first two rows, Yirr-Ma's translator bails out at least once for each of the interpreted programs: in every case the state mapper exhausts the register pool and then fails to compile complex IR effects. It turned out that each of the failed traces contained interpreted `save` or `restore` instructions⁶. The next three rows give the ratio of interpreted source machine instructions to generated host machine instructions. For example, the longest translated trace for the `qsort` program consists of 20 SPARC instructions that are translated into 282 PowerPC instructions, an average ratio of 1:14. Note that further optimizations would improve the quality of the translated host machine instructions. However, compared to Yirr-Ma's dynamic partial Inliner, the specification-driven dynamic binary translator produces a much better code quality. The last row of the table then gives the size of the dynamic code caches.

⁶ The SPARC `save/restore` instructions cause SPARC's register window to shift. In SSL, this semantics is emulated by storing/loading the necessary registers on the emulated stack and then copying registers. The generated IR trace thus contains a sequence of 30 register store/load effects for every instance of these instructions, plus the added overhead for the emulated SPARC register file. The state mapper thus exhausts the host machine's register pool quickly, and is then unable to allocate temporary registers for the compilation of complex IR effects.

Although the few results for the toy benchmark programs are rather disappointing, we expect them to improve (perhaps drastically) for larger source programs. That is because the time spent on the initialization of the dynamic data structures and the dynamic optimization is much shorter in relation to the overall runtime of the program which, ideally, would then spend much more time in translated host machine code.

5.14 Summary and Contribution

In this chapter we discussed at first the quality of dynamically translated machine instructions, and we defined the upper bound for their quality. Using the formal framework for machine emulation and dynamic binary translation that we defined and derived in chapters 1 and 2, respectively, we showed that the quality of translated machine instructions is not only limited by the applied optimization techniques, but also, and more profoundly, by the implementation of the emulated machine state. This implementation of the source machine state on a host machine introduces indirections through emulated memory accesses and loads/stores of emulated machine registers that the generated host machine instructions must account for. We illustrated the upper bound with a comprehensive example. The quality of translated machine instructions can be improved by fine-tuning the implementation of an emulated source machine for a particular host machine, thus increasing the performance of the interpretation of emulated machine instructions and the quality of the dynamically translated machine instructions.

The major part of this chapter was then dedicated to the specification-driven dynamic binary translation, the main contribution of this thesis. Using both an instrumented Walkabout emulator and the generic dynamic optimization framework Yirr-Ma as a vehicle, our dynamic binary translator implements Yirr-Ma's Emitter interface: given the collected semantic information of a trace of interpreted source machine instructions, the translator produces a dynamic function from that trace. All machine dependent components, and therewith most steps of the binary translation process, are either parameterized by, or generated from, machine specification files. Using the source machine dictionary which contains the dynamic operational semantics of every source machine instruction, a sequence of IR effects is generated for a given source machine trace. The derived IR trace may optionally be optimized using various dynamic optimization techniques, and is then compiled into host machine code.

As a first optimization, emulated source machine registers are mapped to actual host machine registers. The register mapping has a generic implementation that is parameterized with the host machine specification file containing a definition of the available set of host machine registers. In subsequent steps, machine independent and some host machine dependent optimizations are applied to the IR trace, thus producing a compact

representation of the selected source machine trace. At last, host machine instructions are selected for every IR effect individually, thus finalizing the translation of source machine instructions to host machine instructions. The selected instructions are then encoded and emitted into a temporary buffer, which is eventually relocated into the body of the dynamic function. The instruction selector itself implements a maximal munch algorithm for given IR effects. We defined the maximal munch algorithm and its generation from a SSL specification file formally.

Initial and experimental results show the applicability of our work, but are far from useful for a production environment. We talk about the conclusions drawn from our experiences and address future work in this and related areas of research in the following chapter.

6 Experiences and Future Work

Ach, nur wer weiß, wohin er fährt, weiß auch welcher
Wind gut und sein Fahrtwind ist.

(Friedrich Nietzsche)

6.1 Experiences With Walkabout and Yirr-Ma

Over the almost two years of our research and using instrumented Walkabout emulators that are linked with the generic dynamic optimizer Yirr-Ma, we gathered much experience with the framework's constraints and limitations, its advantages and disadvantages. We talk about these experiences and then motivate future work in this and related areas in the following sections.

Walkabout

Walkabout is an experimental framework with an as yet fragile implementation. All of its components, the SSL specifications and processing tool, the emulator generator and the generated emulator itself are in places incomplete. Despite of its shortcomings, however, Walkabout and its instrumentation language are conceptually a great experimentation platform that is easy to use and to extend.

The emulator generator reliably processes only SSL machine specification files for SPARC architectures. During our work we extended the emulator generator and its SSL parser such that it now processes Pentium and ARM specification files, only to encounter deeper nested implementation issues. These prevent the emulation and binary translation of little endian source machines on big endian host machines because of the current implementation of the generated instruction interpretation functions. This limited our research to SPARC source machine emulators only. During our work with SSL specifications, we found many bugs in the actual specification files, which consequently resulted in the generation of faulty machine emulators. Curiously, these bugs do not affect the static binary translator UQBT. Locating these bugs is an especially difficult task due to the complexity of the framework and the amount of data produced at runtime. Furthermore, Walkabout's emulator generator, like our dynamic binary translator, faces the problem of weakly typed SSL specifications. Even using a thorough static three-pass type inference over every SSL register transfer, Walkabout's emulator generator warns about insufficient type information and resolves these conflicts using C/C++ type casts.

The generated SPARC emulator is unstable and was not as easy to port to a new host machine as we expected. Because different flavors of Linux on various host machines organize their per-process address space individually, we had to modify the emulator's binary file loader for the different host machines. On PowerPC/Linux host machines, for example, the interpreted dynamic linker `ld.so` of the Solaris source machine must be loaded to a different address in the emulator's address space than on Pentium/Linux. The interpretation of the Solaris linker itself failed for versions other than the SunOS 5.8/patch 3-2002 version. We suspect that this will lead to incompatibilities with other dynamic linker libraries as well.

Yirr-Ma

Because of its consequent use of software interfaces, Yirr-Ma proved to be a very flexible and extensible framework. It allowed us to experiment with different implementations of its various components, and it helped us to study and understand the inner workings of dynamic optimizers in general. However, Yirr-Ma is an add-on to a Walkabout machine emulator linked with the emulator through a custom instrumentation, not a built-in component. Writing an instrumentation definition for the SPARC ISA was both easy and tedious. Expectedly, for more complex and less clean ISAs like Pentium or ARM (both of which are CISC architectures) writing the instrumentation was a more challenging task because of the amount of instructions and their more complex semantics.

Like any other software project, Yirr-Ma evolved with our experience and need. Thus, the implementation of its current architecture contains relics of older versions and quick fixes that can be safely removed or merged with other parts of Yirr-Ma. The redesign of some of Yirr-Ma's interfaces, e.g. that of dynamic functions and machine code wrappers, would be of great value.

Frameworks like Walkabout and Yirr-Ma are complex software products. The location and elimination of bugs often turned into a tedious and time consuming task, and the fact (and poor implementation decision) that none of its components is programmed to be particularly verbose with its debug information was also of little help. Nevertheless, the sheer amount of information that was produced during the interpretation of hundreds of thousands machine instructions was difficult to manage. Furthermore, debugging dynamic functions, i.e. generated and optimized host machine code, was only possible by intercepting their invocation at runtime, and then single-stepping through their disassembled code.

On the other hand, porting Yirr-Ma to new host architectures was always exciting and went smoothly. It required to learn about a new ISA and demonstrated Yirr-Ma's flexibility by short migration times, for example three days to port Yirr-Ma's Inliner to the PowerPC host machine. During the last weeks of our research we started to

implement a Lightning (cf. section 2.5) backend for our specification-driven dynamic binary translator, and we expect this step to increase Yirr-Ma's portability (though using a different approach than the generation of an instruction selector from SSL). Unfortunately, we did not finish the work on this backend.

6.2 Future Work

With Yirr-Ma as an extension to Walkabout emulators, we demonstrated the feasibility and applicability of a purely specification-driven dynamic binary translator. We formalized the general process of dynamic binary translation as a combination of two out of many dynamic optimization techniques, and we discussed the upper bound for the quality of translated machine instructions. Our experimental implementation of Yirr-Ma illustrated how components of a specification-driven dynamic binary translator are composed from, and parameterized by, machine specification files and how they interact with one another. In the end, and most importantly, we learned how one should not implement such a framework.

Debugging and Validation

Neither Walkabout nor Yirr-Ma provide particularly verbose debug interfaces. With every interpreted instruction, Walkabout prints the instruction's mnemonic and the content of the register files automatically, whereas Yirr-Ma logs and prints every activity and method call of each of its components. The extension of Yirr-Ma with a disassembler for dynamic functions and a mapping from interpreted source machine instructions to generated host machine instructions would be of great support in order to locate a faulty translation. (This mapping will be necessary to implement precise exception location for optimized dynamic functions.)

The validation of a correct translation by comparison of emulated machine states with actual machine states, or by verification of the semantics of source and translated host machine instructions, would be of invaluable help. In general, the entire area of *proving the correctness* of machine specification files, and of generated machine emulators and their optimizers is left completely unconsidered by this thesis. It is, however, with respect to the growing importance of virtual machines, the logical consequence of our work, and a required step stone for the development of secure and reliable software.

Specification Files

The SSL specification language lacks some important features, at least for our purposes. Thus, future versions of SSL either require appropriate extensions with explicit type information and enriched state definitions, or SSL itself should be replaced by a specification language that is more suited for our requirements. Consequently, the IR that we borrowed from the UQBT static binary translator needs to be reimplemented with

more performant abstract data types. In addition to the aforementioned problems, other shortcoming of the SSL language that we have encountered for our work are:

- operands (e.g. common addressing modes) can be specified in SSL in a separate section, thus simplifying the specification of instructions that share the same operand types; unfortunately, such operands must not contain more than one register transfer, which is e.g. for the ARM ISA insufficient
- no support of predicated instructions, although guarded SSL register transfers can be used to emulate predicates at the cost of generated instruction interpretation functions with poor performance
- the absence of the possibility to specify an exception model and the invocation of (typed) exceptions by machine instructions
- the missing support of loop and conditional statements (of some sort) over register transfer lists.

Note that some of the mentioned “problems”, particularly the lack of a loop construct, were conscious design decisions of the UQBT team.

More of a side note are some known limitations of the SLED specification language and its toolkit. The major shortcoming is SLED’s incapability to express context sensitive properties of bit patterns: according to a global value/bit or a bit in the instruction’s opcode, a given bit pattern is sometimes required to be decoded differently: Pentium’s 16-bits and 32-bits mode, different available addressing modes for various 68k instructions, or the different “meaning” of the same operand bit pattern of some ARM instructions controlled by a bit in the instruction’s opcode. Furthermore, the tools of the NJMC Toolkit are experimental and different implementations have difficulties to process certain language constructs or more complex machine specification files. They also do not check for the completeness of a specification, i.e. there is no consistency check whether all bit patterns of every specified instruction are covered by the definition, or if a given specification is incomplete.

Alternatively, the investigation of declarative specification languages like SLED or λ -RTL is certainly intriguing future research. The combination of both syntax and semantic specification of machine instructions and their respective code generation tools is a promising research direction. Particularly the implementation of different interpretation algorithms (e.g. partial decoding and evaluation of instructions, or the premature computation of operators) may be catered for by such combined specification languages. After all, decoding an instruction’s bit pattern *is already* a semantic interpretation of the instruction, and thus should be handled as such. Note, in this context, that the interpretation of patterns also includes type information suggesting a typed SLED specification language.

Like conventional source code, so is the writing of machine specification files a creative and also error prone process. The *correctness of machine specification files* themselves is therefore essential for the correct emulation of a machine. A more theoretically oriented research is the validation and verification of such specification files. Preliminary work which validates the correctness of specification files for the syntax of machine instructions is introduced in [RF94].

Our framework provides a good foundation for the previously mentioned validation of optimizing machine emulators because we already use machine specification files to generate both the emulator frontend and the dynamic binary translator backend. The specification files, if appropriately designed, may then be used to validate the correctness of a machine emulator and of the dynamic binary translator, e.g. using a continuous state consistency checking model.

Precision of Machine Emulation

The quality of both the specification files and the emulator generator defines the precision and granularity of the generated optimizing machine emulator. SSL, for example, does not support the specification of processor exceptions, or that of privileged instructions and user models of the specified machine. Therefore, Walkabout’s machine emulator does not support the interpretation of exceptions raised by machine instructions. Also, due to our applied optimizations, dynamic functions obliterate the boundaries of interpreted instructions, and they guarantee the state consistency only at the function’s boundaries. The specification and support of precise exceptions, even in optimized dynamic functions, is a challenging task (cf. Transmeta [DGB⁺03]).

Yirr-Ma

Currently, Yirr-Ma is but an extension to a generated Walkabout emulator. The integration of both and the generation of a seamless optimizing machine emulator are immediate future steps. Using this integrated framework, *layered optimizations* of hot paths will further improve the emulator’s performance: layered optimizations are the combined application of dynamic optimization functions on different levels of the instruction interpretation. Sequences of source machine instructions are optimized by the optimization function for source machine traces $\omega_{I_S}^*$. Code motion and branch inversion, as demonstrated in [UC00b], then generate more potential for the application of the optimization function $\omega_{I_H}^*$ for host traces: optimized and rearranged source machine traces are translated into host machine code by applying the parameterized translation function $\tau^{\omega_{S_H}}$.

Yirr-Ma’s separation of the hot-spot detector and code cache can be combined into a single implementation of both interfaces. The *temporal working set* of an interpreted source machine program, i.e. in temporal proximity frequently interpreted source machine traces, is represented by the hot-spot detector’s set of most recently used counters.

An optimized version of the current temporal working set is always stored in the code cache. The implementation of both interfaces by one single class will simplify the implementation of Yirr-Ma and also remove more time and memory overhead. Furthermore, maintenance procedures for the hot-spot detector (such as the removal of dead and expired counters, or counters of value one) and for the code cache further improve Yirr-Ma's performance, in particular for interpreted source machine programs with an ever-changing working set.

Generation of a Dynamic Binary Translator

As an alternate and more performant technique for dynamic binary translation, Walkabout and Yirr-Ma can easily implement a fixed mapping of source machine instructions to sequences of host machine instructions. Instead of hand-writing an instrumentation that collects the semantic information about interpreted source machine instructions, an emulator generator can

- compute the fixed mapping of source machine instructions to host machine instructions
- extend the generated interpretation functions such that they optimize and emit host machine instructions on the fly.

The dynamic operational semantics of single interpreted machine instructions can be optimized with good results on the level of the IR effects, as previously shown in section 5.9.

In general, the investigation of the generation and validation of *optimizing* machine emulators is an entire new field for future research. Due to the time and memory constraints faced by such a dynamic framework, the techniques and their implementation are different from those of generated static optimizers.

As we have mentioned before, machine specification files may be extended by idiom descriptions: idioms define the equivalence of the operational semantics between different machine instructions, or sequences thereof. The *derivation of the semantic equivalence* of machine instructions from specifications is a challenging future research task which eventually will improve the generation of optimizing machine emulators. Note that the derivation of a semantic equivalence between machine instructions of different ISAs *is* the process of dynamic binary translation.

One of the most striking problems we have encountered was the amount of generated code repetition for the instruction interpretation functions. SSL allows for a very compact specification of the semantics of machine instructions using a lexical concatenation of effects for the specified instructions. However, the generated interpretation functions consist of the unfolded and complete semantics for every single machine instruction. Whereas a SSL specification file contains almost no repeated code, the generated

instruction interpreter contains almost a maximum thereof. Walkabout's emulator generator, in its current implementation, does not harvest the potential for code compaction provided in a SSL specification file.

Dynamic Binary Translator

The quality of a generated dynamic optimizer and, in the context of this thesis, that of a dynamic binary translator is only as good as the specification files that the optimizer was generated from, and as the processing tools. Another future improvement is, therefore, the extension of dynamic functions with local stack frames (to support e.g. register spills or emulated temporary registers), and the generation and implementation of a more sophisticated state mapper or register allocator that improves the quality of the generated host machine trace. The required information must, obviously, also be defined in the machine specification files.

Furthermore, the removal of the machine dictionaries is essential in order to reduce the current runtime and memory overhead. Their generic and parameterized implementation can easily be replaced with a tailored implementation that is generated from the respective machine specification files.

We have not yet addressed the problem of translating an ISA with three instruction's operands to an ISA with two operands, and vice versa. However, this rather minor problem can be solved by dynamically generating, or rewriting, the IR effects appropriately using a fixed and machine independent scheme. The information whether or not to rewrite IR effects must either be provided by, or derived from, the machine specification files.

In order to minimize the number of byte swaps that are performed with every emulated load and store instruction across differing source/host machine pairs, a data cache can be associated with every generated dynamic function. If data is loaded from the emulated memory segment, its byte swapped representation can be stored in the cache for subsequent loads and computations, thus saving additional loads and byte swaps. The epilogue of the dynamic function then synchronizes the cache content with the emulated memory segment by byte swapping all data representations and storing them at their designated host addresses.

Another central problem, especially for cross-platform binary translation (static and dynamic), are the flag registers. In SSL, flag register values are computed by secondary yet explicit effects. Thus, a generated Walkabout emulator, and therefore our dynamic binary translator, handles flag effects explicitly: the emulator computes flag values into dedicated memory locations, and the dynamic binary translator emits host machine code accordingly. The investigation of *implicit* flag emulation and translation is a challenging task which would drastically improve both the emulation of machine flags and the generated host machine code.

Instruction Selector

Clearly, implementing the instruction selector generator for Yirr-Ma as proposed in section 5.11 is the first step to complete our current work. This includes the extension of the algorithms to floating point and control transfer instructions for the host machine. Furthermore, the primary effect of some instructions, e.g. those specified by the SSL TRAP operator or floating point arithmetic across different machines, can not be translated from source to host machine, but must be emulated either by a sequence of host machine instructions or by the invocation of a function on the host machine. The derivation of a host implementation from machine specifications, or its explicit specification, is another problem that future work will have to address.

The next step then is the investigation of more complex instructions, where the primary semantics is defined by more than one effect. Instruction selection through pattern matching would then extend to a conjunctive pattern matcher, where the match of more than one actual tile with a given IR effect (or even a sequence thereof) selects a single host machine instruction.

Alternatively, given declarative machine specifications such as λ -RTL, investigating the derivation of a rewrite grammar from such a specification is an intriguing research project. However, this would affect and cover the areas of both static and dynamic compilation.

Currently, Yirr-Ma interprets source machine system calls by implementing trap instructions with a call to a custom core function: Walkabout's system call mapper. The translation of trap instructions to host machine trap instructions requires a mapping and the adjustment its parameters. More research is thus required to investigate machine specifications, either by extending SSL or by introducing a more abstract system service description. Using this new specification, a generator can generate an instruction selector which performs the mapping and rewriting of parameters when it selects host machine trap instructions. Considering the different endianness and data alignment of different ISAs, however, a generic system call mapper seems to be the simpler and more effective solution to that particular problem.

At last, in order to increase the performance of instruction encoding and emission for the dynamic binary translator, a next version of Yirr-Ma's instruction selector will have to replace SLED's default emission functions by custom functions, thus removing much of the dynamic overhead.

7 Conclusions

Learning what I am, feeling like a Bluebird, flying
away – I love the drops of rain smiling on my
feathers, guiding my way.

(Linda Perry)

In this thesis we formalized the concepts of machine emulation and dynamic optimization of selected traces of interpreted source machine instructions on demand. We also implemented these formal concepts with the introduced Yirr-Ma framework. Yirr-Ma is a generic dynamic optimizer that extends generated Walkabout machine emulators using a custom instrumentation. The added instrumentation allows Yirr-Ma to observe, intercept, and divert the interpretation of user-level programs. In order to do so, it selects traces of interpreted source machine instructions for dynamic optimization, then generates optimized versions of these traces in form of dynamic functions, stores the dynamic functions in a code cache, and finally invokes them in preference to the original trace interpretation. Yirr-Ma is composed of different components, each of whose abstract behavior is defined by a software interface. We have implemented different code caches and hot-spot detectors, and in particular two different dynamic optimizers: a dynamic partial Inliner and an experimental specification-driven dynamic binary translator.

Extending our formal framework for dynamic optimization, we derived a dynamic binary translation function τ that is parameterized with a state optimization function ω_{SH} . We then investigated the upper bound for the quality of dynamically translated machine instructions, and we showed that the source/host machine instruction ratio can not be used in practice to measure the quality of a dynamic binary translation. In fact, the upper bound for the quality of dynamically translated machine instructions is not defined only by the applied dynamic optimizations, but also and more importantly by the implementation of the emulated machine state on the host machine. Using our formal framework as a foundation, we implemented our specification-driven dynamic binary translator. The translator implements an IR generator and a state mapper using dynamic machine dictionaries, both of which are parameterized by source and host machine specification files, respectively. The generated IR trace is a representation of the dynamic operational semantics of interpreted source machine instructions. Dynamic binary translation is performed in several steps. After retrieving the IR trace of the subject source machine trace from the source machine dictionary, the state of the IR

trace is first mapped to the host machine state, thus improving the implementation of the emulated machine state on the host machine. In a next step, the IR trace may optionally be optimized by applying various dynamic optimization techniques, thus improving the interpretation of the source machine instructions by the host machine. Finally, host machine instructions are selected for the mapped and optimized IR trace, completing the translation of a trace of source machine instructions to a trace of host machine instructions. The instruction selector itself can be generated from the same machine specification files that we used to build the Walkabout emulator and to parameterize the generic components of the Yirr-Ma framework.

We presented experimental work in this thesis. Because of severe time constraints we were unable to complete a few aspects of our intended work successfully. However, we demonstrated that an optimizing machine emulator, using dynamic binary translation as an optimization technique, can indeed be built from machine specification files. We pointed out the properties that such specification languages must implement, and provided initial results with the translation of toy benchmark programs. However, the performance results so far indicate a prohibitive large overhead for small programs that we expect to amortize for more complex benchmark programs. A detailed discussion of our experiences and future work addressed how this overhead can be largely removed, and we gave numerous future research directions.

Our current research and the implementation of the Walkabout/Yirr-Ma framework are first steps towards a complete and performant optimizing machine emulator and, for that matter, virtual machine that are generated from machine specification files only. We have encountered numerous problems and implementation issues, provided solutions for most of them and pointed out future work for others. The potential of our approach is impressive, and by demonstrating its feasibility we have as yet only scratched the surface of this area.

A Additional Listings

A.1 Yirr-Ma's FBD Code Cache on Pentium and PowerPC Host Machines

We discussed the quality of dynamically translated machine instructions in section 5.2. The quasi-dynamic translation of the SPARC example trace from listing 3.1 to a SPARC host machine illustrated our reasoning in figure 5.1. The following listings show the same SPARC example trace translated to a Pentium and a PowerPC host machine, respectively. For the Pentium host machine, the compiled trace looks as follows:

```
movl    $0, (%edi)           ; ptr to emulated register file in %edi
movl    12(%ebp), %ecx       ; membase to %ecx
movl    64(%edi), %esi      ; %l0S ↦ %esi
movl    (%esi,%ecx), %eax   ; %l1S ↦ %eax
movl    304(%edi), %ecx     ; %npcS ↦ %ecx
rorw    $8, %ax            ; byteswap 32 bits in %eax
rorl    16, %eax
rorw    8, %ax
movl    %eax, 68(%edi)      ; store %l1S
cld
andb    $1, %dl            ; %NFS ↦ %dl, %dl := %l1S@[31:31]
movb    %dl, 280(%edi)     ; store %NFS
testl   %eax, %eax
sete    %al                ; %ZFS ↦ %al, %al := %l1S=0 ? 1 : 0
movb    %al, 300(%edi)     ; store %ZFS
xorb    $1, %al           ; %CONDS ↦ %al, %al := ~ %ZFS
testb   %al, %al
movb    $0, 281(%edi)      ; store (clear) %OFS
movl    $266349092, %edx   ; %npcS ↦ %edx, %edx set to 0xfe02a24
movb    $0, 257(%edi)     ; store (clear) %CFS
movb    %al, 258(%edi)    ; store %CONDS
jne     .L1034
leal    12(%ecx), %edx     ; %npcS ↦ %edx, %edx set to fallthrough
.L1034:
leal    4(%esi), %eax      ; %l0S ↦ %eax, increment by 4
leal    4(%edx), %esi     ; %pcS ↦ %edx, %npcS ↦ %esi
movb    $1, 259(%edi)     ; store %CTIS
movb    $0, 264(%edi)     ; store (clear) %DLYAS
movl    %eax, 64(%edi)    ; store %l0S
movl    %edx, 308(%edi)   ; store %pcS
movl    %esi, 304(%edi)   ; store %npcS
```

Similarly, the following listing shows the SPARC example trace translated to a PowerPC host machine:

```

; ptr to emulated register file in r31
; emulated membase in r24
li      r29,0          ; clear r29
lwz     r6,64(r31)    ; %l0S ↦ r6
stw     r29,0(r31)    ; store (clear) %g0S
lwz     r7,r6,r24     ; %l1S ↦ r7, r7 := [membase + %l0S]
lwz     r8,304(r31)   ; %npcS ↦ r8, r8 := 0xfe02a28
subfic  r0,r7,0
adde    r3,r0,r7      ; %ZFS ↦ r3, r3 := %l1S=0 ? 1 : 0
srwi    r28,r7,31     ; %NFS ↦ r28, r28 := %l1S@[31:31]
stb     r3,300(r31)   ; store %ZFS
addi    r5,r8,8       ; %npcS ↦ r5, r5 := %npcS + 8
lbz     r0,300(r31)   ; %ZFS ↦ r0
addi    r8,r5,4       ; r8 set to 0xfe02a34 (fallthrough)
stb     r28,280(r31) ; store %NFS
subfic  r3,r0,0
adde    r27,r3,r0     ; %CONDS ↦ r27, r27 := ~ %ZFS
stb     r29,257(r31) ; store (clear) %CFS
stb     r27,258(r31) ; store %CONDS
lbz     r9,258(r31)   ; %CONDS ↦ r9
stw     r7,68(r31)    ; store %l1S
cmpwi   r1,r9,0       ; test r9 (%CONDS) register
stb     r29,281(r31) ; store (clear) %OFS
beq-    1,L1032       ; if %CONDS = 0 then L1032
lis     r4,0xfe0
ori     r8,r4,10788   ; %npcS ↦ r8, r8 set to 0xfe02a24 (taken)
L1032:
addi    r26,r6,4      ; %l0S ↦ r26, increment by 4
addi    r6,r8,4       ; %npcS ↦ r6, and %pcS ↦ r8
li      r9,1
li      r12,0
stw     r9,259(r31)   ; store %CTIS
stw     r12,264(r31) ; store (clear) %DLYAS
stw     r26,64(r31)  ; store %l0S
stw     r6,304(r31)  ; store %npcS
stw     r8,308(r31)  ; store %pcS

```

Note the structural similarities of the generated code for the three examples: all three were translated using the gcc compiler on the respective host machines.

A.2 Walkabout/Yirr-Ma and the Pentium Frontend

Generating an instrumented Pentium frontend from SLED and SSL specifications required us to fix and modify some components of the Walkabout framework. The incorporation of Yirr-Ma, however, went without problems, such that we could experiment with a Pentium frontend, and generate an optimized IR for selected Pentium traces. For example, given is the following trace of Pentium source machine instructions:

```

trace starting at 0x8048120 (0)
out edges: taken (nil) fallthrough (nil)

```

```

08048120: 31 ed          XORmrod   %ebp, %ebp
08048122: 5e           POPod    %esi
08048123: 89 e1       MOVmrod  %ecx, %esp
08048125: 83 e4 f0    ANDiodb  %esp, -16
08048128: 50         PUSHod   %eax
08048129: 54         PUSHod   %esp
0804812A: 52         PUSHod   %edx
0804812B: 68 c0 87 04 08 PUSH.Iv 0x80487c0
08048130: 68 6c 87 04 08 PUSH.Iv 0x804876c
08048135: 51         PUSHod   %ecx
08048136: 56         PUSHod   %esi
08048137: 68 34 82 04 08 PUSH.Iv 0x8048234
0804813C: e8 17 01 00 00 CALL.Jv 0x8048258
08048258: 55         PUSHod   %ebp
08048259: b8 00 00 00 00 MOVId    %eax, 0
0804825E: 31 d2       XORmrod  %edx, %edx
08048260: 89 e5       MOVmrod  %ebp, %esp
08048262: 57         PUSHod   %edi
08048263: 56         PUSHod   %esi
08048264: 53         PUSHod   %ebx
08048265: 81 ec ec 01 00 00 SUBId    %esp, 492
0804826B: 85 c0       TEST.Ev.Gv %eax, 0
0804826D: 8b 5d 0c    MOVrmod  %ebx, 12[%ebp]
08048270: 8b 75 10    MOVrmod  %esi, 16[%ebp]
08048273: 8b 7d 1c    MOVrmod  %edi, 28[%ebp]
08048276: 8d 4c 9e 04 LEAod    %ecx, 4[%esi][%ebx * 4]
0804827A: 74 0f      Jb.Z     0x804828b

```

The above trace of Pentium instructions is represented by the following dynamic operational semantics, derived from the Pentium source machine dictionary with a base address of the emulated memory segment of 0xa7e74000:

```

m[0838408c]{32} := 0x8048120 ; pseudo constant %pc value
; XORmrod %ebp, %ebp
m[08384064]{32} := 0
m[08384078]{1} := 0
m[08384085]{1} := 0
m[08384089]{1} := ((m[08384064]{32} = 0) ? 1 : 0)
m[08384084]{1} := ((m[08384064]{32} & -1) >>A 31)
; POPod %esi
m[08384068]{32} := m[m[08384060]{32} + a7e74000]
m[08384060]{32} := (m[08384060]{32} + 4)
; MOVmrod %ecx, %esp
m[08384054]{32} := m[08384060]{32}
; ANDiodb %esp, -16
m[08384060]{32} := (m[08384060]{32} & sgnex(8,32,-16) )
m[08384078]{1} := 0
m[08384085]{1} := 0
m[08384089]{1} := ((m[08384060]{32} = 0) ? 1 : 0)
m[08384084]{1} := ((m[08384060]{32} & -1) >>A 31)
; PUSHod %eax
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[08384050]{32}
; PUSHod %esp
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[08384060]{32}

```

```
    ; PUSHod %edx
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[08384058]{32}
    ; PUSH.Iv od 0x80487c0
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := 80487c0
    ; PUSH.Iv od 0x804876c
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := 804876c
    ; PUSHod %ecx
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[08384054]{32}
    ; PUSHod %esi
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[08384068]{32}
    ; PUSH.Iv od 0x8048234
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := 8048234
    ; CALL.Iv od 0x8048258
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := (m[0838408c]{32} + 5)
m[0838408c]{32} := 8048258
    ; PUSHod %ebp
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[08384064]{32}
    ; MOVid %eax, 0
r[r32] := 0
    ; XORmrod %edx, %edx
m[08384058]{32} := 0
m[08384078]{1} := 0
m[08384085]{1} := 0
m[08384089]{1} := ((m[08384058]{32} = 0) ? 1 : 0)
m[08384084]{1} := ((m[08384058]{32} & -1) >>A 31)
    ; MOVmrod %ebp, %esp
m[08384064]{32} := m[08384060]{32}
    ; PUSHod %edi
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[0838406c]{32}
    ; PUSHod %esi
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[08384068]{32}
    ; PUSHod %ebx
m[08384060]{32} := (m[08384060]{32} - 4)
m[m[08384060]{32} + a7e74000] := m[0838405c]{32}
    ; SUBid %esp, 492
r[tmp1] := m[08384060]{32}
m[08384060]{32} := (m[08384060]{32} - 492)
m[08384078]{1} := ((~ ((r[tmp1] & -1) >>A 31) & -2) |
    (((m[08384060]{32} & -1) >>A 31) &
    (~ ((r[tmp1] & -1) >>A 31) | -2)))
m[08384085]{1} := (((((r[tmp1] & -1) >>A 31) & 1) &
    ~((m[08384060]{32} & -1) >>A 31)) |
    ((~ ((r[tmp1] & -1) >>A 31) & -2) &
    ((m[08384060]{32} & -1) >>A 31)))
m[08384084]{1} := ((m[08384060]{32} & -1) >>A 31)
m[08384089]{1} := ((m[08384060]{32} = 0) ? 1 : 0)
    ; TEST.Ev.Gv od %eax, %eax
```

```

r[tmp1] := (m[08384050]{32} & m[08384050]{32})
m[08384078]{1} := 0
m[08384085]{1} := 0
m[08384089]{1} := ((r[tmp1] = 0) ? 1 : 0)
m[08384084]{1} := ((r[tmp1] & -1) >>A 31)
; MOVrmod %ebx, 12[%ebp]
m[0838405c]{32} :=
    m[(m[08384064]{32} + sgnex(8,32,12) ) + a7e74000]
; MOVrmod %esi, 16[%ebp]
m[08384068]{32} :=
    m[(m[08384064]{32} + sgnex(8,32,16) ) + a7e74000]
; MOVrmod %edi, 28[%ebp]
m[0838406c]{32} :=
    m[(m[08384064]{32} + sgnex(8,32,28) ) + a7e74000]
; LEAod %ecx, 4[%esi][%ebx * 4]
m[08384054]{32} :=
    ((m[08384068]{32} + sgnex(8,32,4) ) + (m[0838405c]{32} * 4))
; Jb.Z 0x804828b
m[0838408c]{32} :=
    ((m[08384089]{1} = 1) ? 804828b : (m[0838408c]{32} + 2))

```

With the PowerPC as host machine, using the state mapping

```

m[0838408c]32i (%pc) ↦ r[14],    m[08384064]32i (%ebp) ↦ r[15],
m[08384068]32i (%esi) ↦ r[16],  m[08384060]32i (%esp) ↦ r[17],
m[08384054]32i (%ecx) ↦ r[18],  m[08384058]32i (%edx) ↦ r[19],
m[0838405c]32i (%ebx) ↦ r[20],  m[0838406c]32i (%edi) ↦ r[21],

```

and through the application of the different dynamic optimizations, e.g. the propagation and folding of constant values, the forward propagation of pseudo constants and dead effect elimination, the following optimized IR is retrieved:

```

r[17] := m[08384060]{32} ; prologue
r[17] := (r[17] + 4)
r[17] := (r[17] & -16)
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := m[08384050]{32}
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := r[17]
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := r[19]
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := 80487c0
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := 804876c
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := r[18]
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := r[16]
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := 8048234
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := 8048125 ; (%pcS = 8048120) + 5
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := 0

```

```
r[r32] := 0
r[19] := 0
r[15] := r[17]
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := r[21]
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := r[16]
r[17] := (r[17] - 4)
m[r[17] + a7e74000] := r[20]
r[tmp1] := r[17]
r[17] := (r[17] - 492)
r[tmp1] := (m[08384050]{32} & m[08384050]{32})
m[08384078]{1} := 0
m[08384085]{1} := 0
m[08384089]{1} := ((r[tmp1] = 0) ? 1 : 0)
m[08384084]{1} := (r[tmp1] >>A 31)
r[20] := m[(r[15] + 12) + a7e74000]
r[16] := m[(r[15] + 16) + a7e74000]
r[21] := m[(r[15] + 28) + a7e74000]
r[18] := ((r[16] + 4) + (r[20] * 4))
r[14] := ((m[08384089]{1} = 1) ? 804828b : 804825a)
m[08384054]{32} := r[18] ; epilogue
m[08384058]{32} := 0
m[0838405c]{32} := r[20]
m[08384060]{32} := r[17]
m[08384064]{32} := r[15]
m[08384068]{32} := r[16]
m[0838406c]{32} := r[21]
m[0838408c]{32} := r[14]
```

For conceptual and implementation reasons, the generated Walkabout emulator for Pentium source machines does not run on big endian host machines. Because we do not have an instruction selector for Pentium instructions, we were unable to generate host machine code from the optimized IR.

Note that the above IR trace is incomplete. First, the pseudo registers `r[tmp1]` and `r[r32]` are not recognized by Yirr-Ma's current implementation, and thus they are neither adapted to the dynamic operational semantics of the IR trace nor mapped to actual host machine registers. Second, the emulated `%pc` register is not computed correctly by the IR trace. That is because the Pentium SSL specification file was not purified, i.e. the operational semantics of the individual instructions is not self contained and thus incomplete: the `%pc` register is computed by the fetch-and-execute cycle of Walkabout's emulator, and not by the instruction itself. Our original paper on Yirr-Ma [TG02] elaborates on the purification of a SSL specification.

A.3 Walkabout/Yirr-Ma and the ARM Frontend

The generation of an ARM frontend for Walkabout was a matter of less than two weeks. However, the frontend interpreted only a subset of the specified ARM instructions. That

is because the NJMC Toolkit failed to process the complete SLED specification file, and we reduced the set of specified instructions to a subset containing only CTIs, arithmetic and logic instructions for integer values. Furthermore, the SSL specification language in its current implementation is not expressive enough to (efficiently) specify the predicated ARM ISA.

```

20000e0: e3a0b000    mov     r11, #0
20000e4: e8bd0002    ldmia  sp!, {r1}
20000e8: e1a0200d    mov     r2, sp
20000ec: e92d0001    stmdb  sp!, {r0}
20000f0: e59f0010    ldr    r0, [pc, #10]
20000f4: e92d0001    stmdb  sp!, {r0}
20000f8: e59f000c    ldr    r0, [pc, #c]
20000fc: e59f300c    ldr    r3, [pc, #c]
2000100: eb00002b    bl     20001b4

```

The above example trace of ARM instructions is represented by the following dynamic operational semantics. Note that the IR trace below omits the predicates for the individual instructions of the source machine trace which, in this particular case, is of no relevance because all instructions have the implicit predicate AL (always true) set.

```

m[08f8b69c]{32} := 0x20000e0 ; pseudo constant %pc value
; mov r11, #0
m[08f87e30]{32} := (zfill(8,32,0) rr 0)
m[08f8b6ae]{1} := (1 ? m[08f8b6a1]{1} :
((m[08f87e30]{32} & 1) >>A 0))
m[08f8b68c]{32} := m[08f87e30]{32}
; ldmia sp!, {r1}
m[08f87e30]{32} := 1
m[08f87e30]{32} := m[08f8b694]{32}
m[08f8b694]{32} := (m[08f8b694]{32} + (m[08f87e30]{32} * 4))
(2@0:0) = 1 =>
m[08f8b660]{32} := m[m[08f87e30]{32} + a7e74000]
(2@0:0) = 1 =>
m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@1:1) = 1 =>
m[08f8b664]{32} := m[m[08f87e30]{32} + a7e74000]
(2@1:1) = 1 =>
m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@2:2) = 1 =>
m[08f8b668]{32} := m[m[08f87e30]{32} + a7e74000]
(2@2:2) = 1 =>
m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@3:3) = 1 =>
m[08f8b66c]{32} := m[m[08f87e30]{32} + a7e74000]
(2@3:3) = 1 =>
m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@4:4) = 1 =>
m[08f8b670]{32} := m[m[08f87e30]{32} + a7e74000]
(2@4:4) = 1 =>
m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@5:5) = 1 =>
m[08f8b674]{32} := m[m[08f87e30]{32} + a7e74000]
(2@5:5) = 1 =>
m[08f87e30]{32} := (m[08f87e30]{32} + 4)

```

```
(2@6:6) = 1 =>
  m[08f8b678]{32} := m[m[08f87e30]{32} + a7e74000]
(2@6:6) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@7:7) = 1 =>
  m[08f8b67c]{32} := m[m[08f87e30]{32} + a7e74000]
(2@7:7) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@8:8) = 1 =>
  m[08f8b680]{32} := m[m[08f87e30]{32} + a7e74000]
(2@8:8) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@9:9) = 1 =>
  m[08f8b684]{32} := m[m[08f87e30]{32} + a7e74000]
(2@9:9) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@10:10) = 1 =>
  m[08f8b688]{32} := m[m[08f87e30]{32} + a7e74000]
(2@10:10) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@11:11) = 1 =>
  m[08f8b68c]{32} := m[m[08f87e30]{32} + a7e74000]
(2@11:11) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@12:12) = 1 =>
  m[08f8b690]{32} := m[m[08f87e30]{32} + a7e74000]
(2@12:12) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@13:13) = 1 =>
  m[08f8b694]{32} := m[m[08f87e30]{32} + a7e74000]
(2@13:13) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@14:14) = 1 =>
  m[08f8b698]{32} := m[m[08f87e30]{32} + a7e74000]
(2@14:14) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(2@15:15) = 1 =>
  m[08f8b69c]{32} := m[m[08f87e30]{32} + a7e74000]
(2@15:15) = 1 =>
  m[08f8b6af]{1} := ((m[m[08f87e30]{32} + a7e74000] & 1)
    >>A 0)
(2@15:15) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
  ; mov r2, sp
m[08f87e30]{32} := m[08f8b694]{32}
m[08f8b6ae]{1} := m[08f8b6a1]{1}
m[08f8b668]{32} := m[08f87e30]{32}
  ; stmdb sp!, {r0}
m[08f87e30]{32} := 1
m[08f87e30]{32} := (m[08f8b694]{32} - (m[08f87e30]{32} * 4))
m[08f8b694]{32} := (m[08f8b694]{32} - (m[08f87e30]{32} * 4))
(1@0:0) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b660]{32}
(1@0:0) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@1:1) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b664]{32}
```

```

(1@1:1) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@2:2) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b668]{32}
(1@2:2) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@3:3) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b66c]{32}
(1@3:3) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@4:4) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b670]{32}
(1@4:4) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@5:5) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b674]{32}
(1@5:5) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@6:6) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b678]{32}
(1@6:6) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@7:7) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b67c]{32}
(1@7:7) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@8:8) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b680]{32}
(1@8:8) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@9:9) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b684]{32}
(1@9:9) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@11:11) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b68c]{32}
(1@11:11) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@12:12) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b690]{32}
(1@12:12) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@13:13) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b694]{32}
(1@13:13) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@14:14) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b698]{32}
(1@14:14) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@15:15) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b69c]{32}
(1@15:15) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
  ; ldr r0, [pc, #10]
m[08f87e30]{32} := (1 ? (m[08f8b69c]{32} + zfill(12,32,16) ) :
                    (m[08f8b69c]{32} - zfill(12,32,16) ))

```

```
m[08f8b660]{32} := (m[m[08f87e30]{32} + a7e74000] rr
                    ((m[08f87e30]{32} & 3) * 8))
m[08f8b660]{32} := ((Rd = 15) ? m[08f8b660]{32} :
                    m[08f8b660]{32})
; stmdb sp!, {r0}
m[08f87e30]{32} := 1
m[08f87e30]{32} := (m[08f8b694]{32} - (m[08f87e30]{32} * 4))
m[08f8b694]{32} := (m[08f8b694]{32} - (m[08f87e30]{32} * 4))
(1@0:0) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b660]{32}
(1@0:0) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@1:1) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b664]{32}
(1@1:1) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@2:2) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b668]{32}
(1@2:2) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@3:3) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b66c]{32}
(1@3:3) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@4:4) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b670]{32}
(1@4:4) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@5:5) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b674]{32}
(1@5:5) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@6:6) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b678]{32}
(1@6:6) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@7:7) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b67c]{32}
(1@7:7) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@8:8) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b680]{32}
(1@8:8) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@9:9) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b684]{32}
(1@9:9) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@11:11) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b68c]{32}
(1@11:11) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@12:12) = 1 =>
  m[m[08f87e30]{32} + a7e74000] := m[08f8b690]{32}
(1@12:12) = 1 =>
  m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@13:13) = 1 =>
```

```

    m[m[08f87e30]{32} + a7e74000] := m[08f8b694]{32}
(1@13:13) = 1 =>
    m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@14:14) = 1 =>
    m[m[08f87e30]{32} + a7e74000] := m[08f8b698]{32}
(1@14:14) = 1 =>
    m[08f87e30]{32} := (m[08f87e30]{32} + 4)
(1@15:15) = 1 =>
    m[m[08f87e30]{32} + a7e74000] := m[08f8b69c]{32}
(1@15:15) = 1 =>
    m[08f87e30]{32} := (m[08f87e30]{32} + 4)
    ; ldr r0, [pc, #c]
m[08f87e30]{32} := (1 ? (m[08f8b69c]{32} + zfill(12,32,12)) :
                    (m[08f8b69c]{32} - zfill(12,32,12)))
m[08f8b660]{32} := (m[m[08f87e30]{32} + a7e74000] rr
                    ((m[08f87e30]{32} & 3) * 8))
m[08f8b660]{32} := ((Rd = 15) ? m[08f8b660]{32} :
                    m[08f8b660]{32})
    ; ldr r3, [pc, #c]
m[08f87e30]{32} := (1 ? (m[08f8b69c]{32} + zfill(12,32,12)) :
                    (m[08f8b69c]{32} - zfill(12,32,12)))
m[08f8b66c]{32} := (m[m[08f87e30]{32} + a7e74000] rr
                    ((m[08f87e30]{32} & 3) * 8))
m[08f8b66c]{32} := ((Rd = 15) ? m[08f8b66c]{32} :
                    m[08f8b66c]{32})
    ; bl 20001b4
m[08f8b698]{32} := (m[08f8b69c]{32} + 4)
m[08f8b69c]{32} := (m[08f8b69c]{32} + (sgnex(24,32,43) * 4))

```

The premature evaluation of guards for IR expressions (denoted by the SSL operator *guard => expr* in the above IR trace) in particular led to a much improved IR trace. Furthermore, the application of constant value propagation and folding, dead effect elimination and other dynamic optimizations results in the following much more compact IR trace:

```

r[17] := m[08f8b694]{32} ; prologue
r[16] := 0
r[15] := r[17]
r[17] := (r[17] + (r[15] * 4))
r[18] := m[r[15] + a7e74000]
r[15] := r[17]
m[08f8b6ae]{1} := m[08f8b6a1]{1}
r[19] := r[15]
r[15] := (r[17] - 4)
r[17] := (r[17] - (r[15] * 4))
m[r[15] + a7e74000] := r[20]
r[20] := m[20000f0 + a7e74000]
r[15] := (r[17] - 4)
r[17] := (r[17] - (r[15] * 4))
m[r[15] + a7e74000] := r[20]
r[20] := m[20000ec + a7e74000]
r[20] := ((Rd = 15) ? r[20] : r[20])
r[15] := 20000ec
r[21] := m[20000ec + a7e74000]
r[21] := ((Rd = 15) ? r[21] : r[21])

```

```

r[22] := 20000e4
r[14] := 200018c
m[08f87e30]{32} := 20000ec ; epilogue
m[08f8b660]{32} := r[20]
m[08f8b664]{32} := r[18]
m[08f8b668]{32} := r[19]
m[08f8b66c]{32} := r[21]
m[08f8b68c]{32} := 0
m[08f8b694]{32} := r[17]
m[08f8b698]{32} := 20000e4
m[08f8b69c]{32} := 200018c

```

The state mapping from the ARM source machine to the PowerPC host machine is hereby defined as follows:

$$\begin{aligned}
m[08f8b69c]_{32i}(\text{pc}) &\mapsto r[14], & m[08f87e30]_{32i}(r[\text{tmp}]) &\mapsto r[15], \\
m[08f8b68c]_{32i}(r11) &\mapsto r[16], & m[08f8b694]_{32i}(\text{sp}) &\mapsto r[17], \\
m[08f8b664]_{32i}(r1) &\mapsto r[18], & m[08f8b668]_{32i}(r2) &\mapsto r[19], \\
m[08f8b660]_{32i}(r0) &\mapsto r[20], & m[08f8b66c]_{32i}(r3) &\mapsto r[21], \\
m[08f8b698]_{32i}(1r) &\mapsto r[22].
\end{aligned}$$

Similar to the incomplete IR trace for the Pentium source machine in the previous section [A.2](#), and for the same reasons, so is this IR trace incomplete.

A.4 Walkabout/Yirr-Ma and the PowerPC Backend

Figure [5.15](#) in section [5.10](#) shows an excerpt of the PowerPC host machine code that was generated from the SPARC example trace in listing [5.6](#). The following listing gives the complete generated host machine code, annotated with the IR effects that the respective machine instructions were generated for:

```

; r[19] := m[102e39c8]{32}
0x105216d8: lis    r20,4142
0x105216dc: ori    r20,r20,14792
0x105216e0: lwz   r19,0(r20)
; r[16] := m[r[19] + 0x30032000]
0x105216e4: lis    r20,12291
0x105216e8: ori    r20,r20,8192
0x105216ec: add   r20,r19,r20
0x105216f0: lwz   r16,0(r20)
; r[15] := r[16]
0x105216f4: ori    r15,r16,0
; m[102e3aa0]{1} := (r[15] >>A 31)
0x105216f8: srawi r20,r15,31
0x105216fc: lis    r21,4142
0x10521700: ori    r21,r21,15008
0x10521704: andi. r20,r20,1
0x10521708: stb   r20,0(r21)
; m[102e3ab4]{1} := ((r[15] = 0) ? 1 : 0)
0x1052170c: cmpwi cr3,r15,0
0x10521710: beq-  cr3,0x10521720

```

```

0x10521714:  lis    r20,0
0x10521718:  ori    r20,r20,0
0x1052171c:  b      0x10521728
0x10521720:  lis    r20,0
0x10521724:  ori    r20,r20,1
0x10521728:  lis    r21,4142
0x1052172c:  ori    r21,r21,15028
0x10521730:  andi.  r20,r20,1
0x10521734:  stb    r20,0(r21)
; m[102e3aa1]{1} := 0
0x10521738:  lis    r20,0
0x1052173c:  ori    r20,r20,0
0x10521740:  lis    r21,4142
0x10521744:  ori    r21,r21,15009
0x10521748:  andi.  r20,r20,1
0x1052174c:  stb    r20,0(r21)
; m[102e3a89]{1} := 0
0x10521750:  lis    r20,0
0x10521754:  ori    r20,r20,0
0x10521758:  lis    r21,4142
0x1052175c:  ori    r21,r21,14985
0x10521760:  andi.  r20,r20,1
0x10521764:  stb    r20,0(r21)
; m[102e3a8a]{1} := ~ m[102e3ab4]{1}
0x10521768:  lis    r20,4142
0x1052176c:  ori    r20,r20,15028
0x10521770:  lbz    r20,0(r20)
0x10521774:  andi.  r20,r20,1
0x10521778:  not    r20,r20
0x1052177c:  lis    r21,4142
0x10521780:  ori    r21,r21,14986
0x10521784:  andi.  r20,r20,1
0x10521788:  stb    r20,0(r21)
; m[102e3a8a]{1} = 0 ? ...
0x1052178c:  lis    r20,4142
0x10521790:  ori    r20,r20,14986
0x10521794:  lbz    r20,0(r20)
0x10521798:  andi.  r20,r20,1
0x1052179c:  cmpwi  cr3,r20,0
0x105217a0:  beq-   cr3,0x105217b0
; r[14] := 0xfe02a24
0x105217a4:  lis    r14,4064
0x105217a8:  ori    r14,r14,10788
0x105217ac:  b      0x105217b8
; r[14] := 0xfe02a34
0x105217b0:  lis    r14,4064
0x105217b4:  ori    r14,r14,10804
; r[15] := (r[15] - r[15])
0x105217b8:  subf   r15,r15,r15
; r[18] := r[19]
0x105217bc:  ori    r18,r19,0
; r[19] := (r[19] + 4)
0x105217c0:  addi   r19,r19,4
; r[17] := r[14]
0x105217c4:  ori    r17,r14,0
; r[14] := (r[14] + 4)
0x105217c8:  addi   r14,r14,4

```

```
    ; m[102e1d70]{32} := r[18]
0x105217cc:  lis    r20,4142
0x105217d0:  ori    r20,r20,7536
0x105217d4:  stw   r18,0(r20)
    ; m[102e3988]{32} := 0
0x105217d8:  lis    r20,0
0x105217dc:  ori    r20,r20,0
0x105217e0:  lis    r21,4142
0x105217e4:  ori    r21,r21,14728
0x105217e8:  stw   r20,0(r21)
    ; m[102e39c8]{32} := r[19]
0x105217ec:  lis    r20,4142
0x105217f0:  ori    r20,r20,14792
0x105217f4:  stw   r19,0(r20)
    ; m[102e39cc]{32} := r[16]
0x105217f8:  lis    r20,4142
0x105217fc:  ori    r20,r20,14796
0x10521800:  stw   r16,0(r20)
    ; m[102e3ab8]{32} := r[14]
0x10521804:  lis    r20,4142
0x10521808:  ori    r20,r20,15032
0x1052180c:  stw   r14,0(r20)
    ; m[102e3abc]{32} := r[17]
0x10521810:  lis    r20,4142
0x10521814:  ori    r20,r20,15036
0x10521818:  stw   r17,0(r20)
```

As can be seen would host machine specific optimizations, e.g. using a peephole optimizer, improve the quality of the translated trace of host machine instructions. Such optimizations, however, are outside of the scope of this thesis.

Bibliography

- [AFG⁺00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000. 35
- [AK91] Hassan Ait-Kaci. *Warren’s Abstract Machine, A Tutorial Reconstruction*. The MIT Press, 1991. 8
- [Bar84] Henk Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, 1984. 7
- [BCF⁺99] Micheal Burke, Jong-Deok Choi, Stephen Fink, David Grove, Micheal Hind, Vivek Sarkar, Mauricio Serrano, Vugranam Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, June 1999. 35
- [BDB99a] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent Dynamic Optimization. Technical Report HPL-1999-77, HP Laboratories Cambridge, 1999. 32
- [BDB99b] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent Dynamic Optimization: The Design and Implementation of Dynamo. Technical Report HPL-1999-78, HP Laboratories Cambridge, 1999. 32
- [BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. *ACM SIGPLAN Notices*, 35(5):1–12, 2000. 32
- [Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973. 36
- [Bel04] Fabrice Bellard. The QEMU CPU Emulator, 2004. <http://fabrice.bellard.free.fr/qemu/>. 43
- [BGvN46] Arthur Burks, Herman Goldstine, and John von Neumann. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. Report to the U.S. Army Ordnance Department, 1946. Reprinted in [Sch76, pp. 221–221]. 7
- [Bon03] Paolo Bonzini. GNU Lightning, 2003. <http://www.gnu.org/software/lightning/lightning.html>. 43, 93
- [Cat78] Roderic Cattell. Formalization and Automatic Derivation of Code Generators. Technical Report CMU-CS-78-115, Carnegie-Mellon University, 1978. 108

- [Cat80] Roderic Cattell. Automatic Derivation of Code Generators from Machine Descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, 1980. [119](#)
- [CEe99] Cristina Cifuentes, Mike Van Emmerik, and et.al. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. In *Workshop on Binary Translation, Technical Committee on Computer Architecture Newsletter*, 1999. [6](#), [20](#), [128](#)
- [CH97] Anton Chernoff and Ray Hookway. DIGITAL FX!32 — Running 32-Bit x86 Applications on Alpha NT. In *Proceedings of the USENIX Windows NT Workshop*, pages 9–13, 1997. [11](#)
- [CK94] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994. [10](#)
- [CLCG00] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gilles. Mojo: A Dynamic Optimization System. Technical report, Microsoft Research, 2000. [34](#)
- [CLU02] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout – A Retargetable Dynamic Binary Translation Framework. Technical Report SMLI TR-2002-106, Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, 2002. [22](#), [26](#), [34](#)
- [CM96] Cristina Cifuentes and Vishv Malhorta. Binary Translation: Static, Dynamic, Retargetable? In *International Conference on Software Maintenance – IEEE Computer Society*, 1996. [36](#)
- [Col90] Hélène Collavizza. Functional Semantics of Microprocessors at the Microprogram Level and Correspondence with the Machine Instruction Level. In *Proceedings of the Conference on European Design Automation*, pages 52–56. IEEE Computer Society Press, 1990. [19](#)
- [CS98] Cristina Cifuentes and Shane Sendall. Specifying the Semantics of Machine Instructions. In *6th International Workshop on Program Comprehension*, pages 126–133. IEEE Computer Society, 1998. [19](#)
- [DA02] Evelyn Duesterwald and Saman P. Amarsinghe. On the Run – Building Dynamic Program Modifiers for Optimization, Introspection and Security. In *ACM SIGPLAN on PLDI*, 2002. [32](#)
- [Dav92] Antony Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992. [8](#)
- [DGB⁺03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24, 2003. [39](#), [139](#)

- [EA97] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *ISCA*, pages 26–37, 1997. 41
- [EAGS01] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, and Sumedh W. Sathaye. Dynamic Binary Translation and Optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001. 41
- [Ee02] Mike Van Emmerik and et.al. Boomerang decompiler homepage. BSD licenced software, 2002. <http://boomerang.sourceforge.net>. 20
- [Ert01] M. Anton Ertl. Threaded Code Variations and Optimizations. In *EuroForth 2001 Conference Proceedings*, pages 49–55, 2001. 36
- [FE98] Manel Fernandez and Roger Espasa. Dixie Architecture Reference Manual (Version 1.0). Technical report, Computer Architecture Department, Universitat Politecnica de Catalunya-Barcelona, 1998. 12
- [FE99] Manel Fernandez and Roger Espasa. Dixie: A Retargetable Binary Instrumentation Tool. In *Workshop on Binary Translation, PACT*, 1999. 12
- [FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, 1992. 108
- [FS99] Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley, 1999. 48
- [FSW94] Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. Tree Automata for Code Selection. *Acta Informatica*, 31(8):741–760, 1994. 108
- [GEK01] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an Efficient Java Interpreter. *Lecture Notes in Computer Science*, 2110, 2001. <http://www.complang.tuwien.ac.at/forth/threaded-code.html>. 36
- [GJSB00] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000. 9
- [GL97] John Gough and Jeff Ledermann. Optimal Code-Selection using MBURG. *20th Australasian Computer Science Conference*, 19(1):441–450, 1997. 108
- [Gou01] John Gough. Stacking Them Up: A Comparison of Virtual Machines. In *Australian Computer Systems and Architecture Conference*, 2001. 9
- [HA96] Urs Hölzle and Ole Agesen. Dynamic vs. Static Optimization Techniques for Object-Oriented Languages. Technical report, University of California in Santa Barbara, 1996. 36
- [HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *In the Proceedings of the ECOOP*, 1991. 36

- [HS02] Kim Hazelwood and Michael D. Smith. Code Cache Management Schemes for Dynamic Optimizers. In *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, 2002. 52, 57
- [Jos99] Nicolai Josuttis. *The C++ Standard Library: a Tutorial and Reference*. Addison-Wesley, 1999. 56
- [KF99] Thomas Kistler and Michael Franz. The Case for Dynamic Optimization: Improving Memory-Hierarchy Performance by Continuously Adapting the Internal Storage Layout of Heap Objects at Run-Time. Technical Report 99–21, University of California, Irvine, 1999. 35
- [Kis99] Thomas Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, 1999. 35, 36, 49
- [Kla00] Alexander Klaiber. The Technology behind Crusoe Processors. Technical report, Transmeta Corporation, 2000. <http://www.transmeta.com/developers/techdocs.html>. 39
- [Knu71] Donald E. Knuth. An Empirical Study of FORTRAN Programs. *Software—Practice and Experience*, 1:105–133, 1971. 30
- [Law04] Kevin Lawton. Bochs - a x86 Emulator, 2004. <http://bochs.sourceforge.net/>. 14
- [LD97] Mark Leone and R. Kent Dybvig. Dynamo: A staged Compiler Architecture for Dynamic Program Optimization. Technical report, Indiana University Bloomington, 1997. 8
- [LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading MA, 1997. 9
- [McC85] John McCarthy. *LISP Programming Language*. MIT Press, 2nd Edition, 1985. 8
- [Mel99] Steve Meloan. The Java HotSpot Performance Engine: An In-Depth Look. *Sun Microsystems Technical Article*, 1999. <http://developer.java.sun.com/developer/technicalArticles/>. 9
- [MR03] James Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley, 1st Edition, 2003. 8, 10
- [NL03] Drew Northup and Eric Laberge. Plex, 2003. <http://savannah.nongnu.org/projects/plex86/>. 14
- [ÖG98] Soner Önder and Rajiv Gupta. Automatic Generation of Microarchitecture Simulators. In *IEEE International Conference on Computer Languages*, Chicago, 1998. 18, 19
- [PKS02] Mark Probst, Andreas Krall, and Bernhard Scholz. Register Liveness Analysis for Optimizing Dynamic Binary Translation. In *Proceedings of the Ninth Working Conference on Reverse Engineering*. IEEE Computer Society, 2002. 42

- [PR98] Ian Piumarta and Fabio Riccardi. Optimizing Direct-threaded Code by Selective Inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998. 63
- [Pro95] Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *Conference Record of POPL 1995: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, California, 1995. 63
- [Pro01] Mark Probst. Fast Machine-Adaptable Dynamic Binary Translation. In *Workshop on Binary Translation*, 2001. 42
- [Pro02] Mark Probst. Dynamic Binary Translation. In *UKUUG Linux Developers' Conference*, 2002. 42
- [RD99] Norman Ramsey and Jack W. Davidson. Specifying Instructions' Semantics using λ -RTL. Technical report, Department of Computer Science, University of Virginia, 1999. 19
- [RDF01] Norman Ramsey, Jack W. Davidson, and Mary F. Fernández. Design Principles for Machine-Description Languages, 2001. <http://www.eecs.harvard.edu/~nr/pubs/desprin-abstract.html>. 18, 24
- [RDGY99] Chris Reeve, Dean Deaver, Rick Gorton, and Bharadwaj Yadavalli. Wiggins/Redstone: A Dynamic Optimization and Specialization Tool. Technical report, Compaq Computer Corporation, 1999. 33
- [RF94] Norman Ramsey and Mary F. Fernández. New Jersey Machine-Code Toolkit Reference Manual Version 0.1. Technical Report TR-471-94, Computer Science Department, Princeton University, October 1994. 18, 116, 139
- [RF97] Norman Ramsey and Mary F. Fernández. Specifying Representations of Machine Instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997. 18
- [RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, 1995. 11
- [Sch76] Earl Schwartzlander. *Computer Design Development - Principal Papers*. Hayden Book Company Inc., 1976. 159
- [Sco99] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufman Publishers, 1999. 48, 101, 102
- [SD01] Kevin Scott and Jack W. Davidson. Strata: A Software Dynamic Translation Infrastructure. Technical Report CS-2001-17, University of Virginia, 2001. 13

- [SEV01] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft Corp., 2001. 34
- [SKV⁺03] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. Retargetable and Reconfigurable Software Dynamic Translation. In *International Symposium on Code Generation and Optimization*, pages 36–47, 2003. 13, 26
- [Smi00] Michael D. Smith. Overcoming the Challenges to Feedback-Directed Optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Boston, MA, 2000. 31
- [TC02] Jens Tröger and Cristina Cifuentes. Analysis of Virtual Method Invocation for Binary Translation. In *Proceedings of the Ninth Working Conference on Reverse Engineering*. IEEE Computer Society, 2002. 6
- [Tec00] Transitive Technologies. Dynamite – Dynamic Binary Translation. Technical report, Transitive Technologies Ltd, San Diego, USA, 2000. <http://www.transitives.com/>. 40
- [TG02] Jens Tröger and John Gough. Fast Dynamic Binary Translation – The Yirra-Ma Framework. In *Workshop on Binary Translation, in conjunction with ACM PACT*, 2002. 63, 150
- [Trö01] Jens Tröger. Specification of the HRTL Abstract Machine. Technical Report FIT-TR-2001-03, Queensland University of Technology, Brisbane, Australia, 2001. 6
- [Tur36] Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2, Vol. 42:230–265, 1936. 7, 16
- [UC00a] David Ung and Cristina Cifuentes. Machine-Adaptable Dynamic Binary Translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 41–51, 2000. 38
- [UC00b] David Ung and Cristina Cifuentes. Optimizing Hot Paths in a Dynamic Binary Translation. *Workshop on Binary Translation*, 2000. 38, 139
- [US87] David Ungar and Randall Smith. SELF: The Power of Simplicity. In *Proceedings on OOPSLA*, pages 227–241, 1987. <http://research.sun.com/research/self/>. 8, 36
- [WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996. 11