Walkabout—A Retargetable Dynamic Binary Translation Framework

Cristina Cifuentes, Brian Lewis, and David Ung

Walkabout—A Retargetable Dynamic Binary Translation Framework

Cristina Cifuentes, Brian Lewis, and David Ung

SMLI TR-2002-106 January 2002

Abstract:

Dynamic compilation techniques have found a renaissance in recent years due to their use in high-performance implementations of the Java™ language. Techniques originally developed for use in virtual machines for such object-oriented languages as Smalltalk are now commonly used in Java virtual machines (JVM™) and Java just-in-time compilers. These techniques have also been applied to binary translation in recent years, most commonly appearing in binary optimizers for a given platform that improve the performance of binary programs while they execute.

The Walkabout project investigates and develops dynamic binary translation techniques that are based on properties of retargetability, ease of experimentation, separation of machine-dependent from machine-independent concerns, and good debugging support. Walkabout is a framework for experimenting with dynamic binary translation ideas, as well as techniques in related areas such as interpreters, instrumentation tools, and optimization.

In this report, we present the design of the Walkabout framework and its initial implementation. Tools generated from this initial framework include disassemblers, machine code interpreters (emulators), and binary rewriting tools for the SPARC® and x86 architectures.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
cristina.cifuentes@sun.com
brian.t.lewis@intel.com
david.ung@itee.uq.edu.au

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.All technical reports are available online on our Website, http://research.sun.com/techrep/.

Walkabout – A Retargetable Dynamic Binary Translation Framework

Cristina Cifuentes
Sun Microsystems Laboratories
Palo Alto, CA 94303, USA
cristina.cifuentes@sun.com

Brian Lewis*
Sun Microsystems Laboratories
Palo Alto, CA 94303, USA
brian.lewis@sun.com

David Ung†
Sun Microsystems Laboratories
Palo Alto, CA 94303, USA
david.ung@sun.com

1 Introduction

Binary translation, the process of translating binary executables[1], makes it possible to run code compiled for a source (input) machine M$_s$ on a target (output) machine M$_t$. Unlike an interpreter or emulator, a binary translator makes it possible to approach the speed of native code on machine M$_t$. Translated code may run more slowly than native code because low-level properties of machine M$_s$ must often be modeled on machine M$_t$. For example, the Digital Freeport Express translator [Dig95] simulates the byte order of the SPARC(R) architecture, and the FX!32 translator [Tho96, HH97] simulates the calling sequence of the source x86 machine, even though neither of these is native to the target Alpha architecture.

The Walkabout framework is a retargetable binary translation framework for experimenting with dynamic translation of binary code. The framework had its inspiration in the University of Queensland Binary Translator (UQBT) framework [CE00, CERL02, CERL01]. We took what we had learned in the areas of retargetability and separation of machine-dependent from machine-independent concerns, and applied these techniques to the new dynamic framework. Of course, the code transformations we could support were different due to differences between dynamic and

[*] Now at Intel Microprocessor Research Labs. Email: brian.t.lewis@intel.com

[†] The author was an intern at Sun Microsystems Laboratories when this work was conducted. Current address: The University of Queensland, david.ung@itee.uq.edu.au

[1] In this document, the terms *binary executable*, *executable*, and *binary* files are used as synonyms to refer to the binary image file generated by a compiler or assembler to run on a particular computer.

static translation. For example, a static translator operates before the target program executes, and so can afford to use expensive optimizations. A dynamic translator, on the other hand, operates while the target program is executing, and the time it requires is unavailable to the target program.

1.1 Goals and Objectives

Binary translation requires machine-level analyses to transform source binary code onto target binary code, either by emulating features of the source machine or by identifying such features and transforming them into equivalent target machine features. In the Walkabout system, we make use of both types of transformations, and determine when it is safe to make use of native target features.

One question that must be answered early in the development of a binary translation system is what intermediate representation to use. In the UQBT system, we make use of RTLs and HRTLs. The former is a register transfer language that makes explicit every transfer of control, while the latter is a high-level register transfer language that resembles simple imperative languages, with explicit control transfers.

Many other binary translation systems have used machine code itself as the intermediate representation, mainly because these systems were binary code reoptimizers and were generating code for the same machine. Such systems include Dynamo [BDB00], Wiggins/Redstone [DGR99] and Mojo [CLCG00].

For Walkabout, we have initially used machine code as the intermediate representation. We plan to use RTL as the next step. However, we want to experiment with how well RTLs support translation into a target representation. RTLs are still machine-dependent as they expose such features as delayed branches and register windows.

The goals of the project are:

- to derive components of binary translators as much as possible from machine descriptions,
- to understand how to instrument interpreters in a retargetable way,
- to determine whether an RTL representation is best suited for dynamic machine translation, and how to best map M_s-RTLs to M_t-RTLs[2],
- to understand how debugging support needs to be integrated into a dynamic binary translation system, and
- to develop a framework for quick experimentation with dynamic binary-manipulation techniques.

We limit binary translation to user-level code and to multiplatform operating systems such as Solaris(TM) and Linux.

[2] RTLs for a given machine M are denoted M-RTLs.

2 Previous Work

Dynamic binary translation techniques evolved from emulation and simulation techniques as a more efficient way of running programs on other machines by generating target native code on the fly. Many of the techniques used in dynamic binary translation systems were originally developed in language interpreters and virtual machines for languages such as APL, Forth, Smalltalk, Self, and now Java(TM). We review the literature in three main areas: existing examples of dynamic binary translators, binary rewriting tools, and virtual machines for object-oriented languages.

2.1 Other Dynamic Binary Translators

DAISY (Dynamically Architected Instruction Set from Yorktown), is a VLIW virtual machine that emulates existing machines in order to run applications, including low-level system programs, on the DAISY machine [EA96]. The machines it supports are the PowerPC, x86, and S/390. DAISY was developed at IBM T.J.Watson to aid in determining what features a new VLIW architecture should have in order to successfully run a variety of existing programs. The DAISY system provides simulations of how well a new architecture might perform if it were built.

HP's Aries is a PA-RISC to IA-64 dynamic translator for the HP-UX operating system, which is planned to be shipped with all IA-64 HP-UX systems [ZT00]. Aries combines a fast interpreter with a dynamic optimizing compiler (i.e., a just-in-time, or JIT, compiler). It operates by compiling frequently interpreted code sequences into native IA-64 code. Aries allows users to migrate their PA-RISC applications transparently, without user intervention. Runtime verification support was also built as part of this tool.

Transmeta's Crusoe architecture makes use of dynamic binary translation—which they term "code morphing"—to support running x86 applications on the Crusoe processor, a VLIW machine with instruction level parallelism [Kla00]. The code morphing software resides in a ROM and is effectively a layer that sits in between the BIOS and the VLIW processor. Crusoe has support at the hardware level for features that are hard or expensive to support at the software level. Precise exception handling is done by shadowing all registers holding x86 state information, and providing a commit and rollback operation to copy registers between the working and the shadow copies. Alias detection hardware guards against illegal moves of load instructions ahead of store instructions, when the store instruction overwrites the previously loaded data. Self-modifying code is supported by detecting when it happens, so that previously-translated code can be flagged as invalid. This is done by protecting pages with a translated bit.

Two academic systems have been described in the literature: UQDBT [UC00] and bintrans [Pro01]. Both systems show that retargetable dynamic binary translation is feasible. The former is based on the UQBT system and makes use of one intermediate representation for both code generation and machine-independent translation purposes. The latter makes no use of an intermediate representation and compiles all the code that is decoded on-the-fly, achieving better performance for the limited class of architectures on which it was tested.

2.2 Other Binary Rewriting Tools

Areas related to dynamic binary translation include emulation and simulation, where work has been done since the 1960s. Recent emulators and simulators include the Sun Microsystems tools Shade [CK93] and Wabi [HMR96, FN96], as well as the Stanford research simulators SimOS [RHWG95] and Embra [WR96].

In the last few years, several dynamic reoptimizers have been developed as research projects. Their goal is to transparently reoptimize binaries at runtime to improve their performance. None of these systems has been used commercially since there are still a number of open research questions in how to implement these systems. For example, it is hard to achieve consistently good performance while correctly executing all programs. The systems include: HP Labs' Dynamo [BDB00], a reoptimizer of PA-RISC code; Compaq's Wiggins/Redstone [DGR99], a reoptimizer for Alpha code; and Microsoft Research's Mojo [CLCG00], a reoptimizer of (x86,Windows) binaries.

Both Dynamo and Mojo count branching instructions while interpreting code to detect frequently executed code and generate native code for it, applying a variety of optimizations. Wiggins/Redstone samples CPU events for frequently executed instructions in the target program using hardware event counters and the Digital Continuous Profiling Infrastructure (DCPI [ABD$^+$97]). This allows the tool to determine a seed instruction out of which a trace of frequently executed code is determined.

All systems show mixed results: some SPEC benchmark programs perform faster, while others perform slower. For programs where performance is degradated, no study has been reported in the literature that aids in understanding what features affected these programs; some of the programs that worked well under Dynamo did not work well under Wiggins/Redstone, for example. In addition, common problems involving changes in a program's phase behaviour have not been studied.

2.3 Virtual Machines for Object-Oriented Languages

In recent years, there has been a renaissance of dynamic compilation techniques, mainly due to the popularity of the Java programming language and the drive to build high-performance Java virtual machines (JVM(TM)[3]) that run Java code faster.

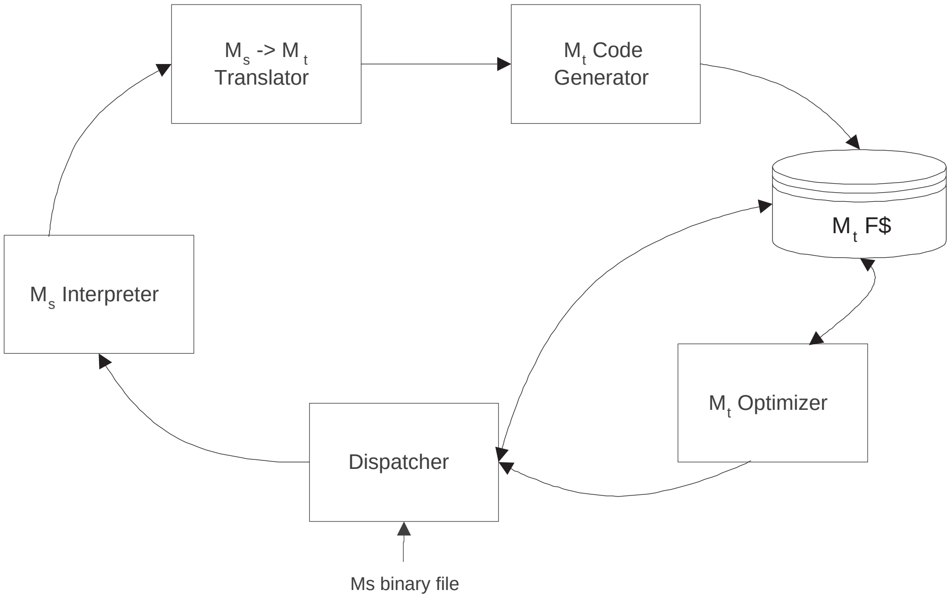
JVMs such as Sun Microsystems Java HotSpot(TM) virtual machine [HBG$^+$97, GM00, PVC01] use dynamic compilation techniques to produce good quality native code. These techniques have been derived over time from virtual machine technology used to implement object-oriented languages such as Smalltalk [GR83, DS84] and Self [US87, Hol94]. Self was the inspiration for the original Java HotSpot virtual machine.

The architecture of optimizing virtual machines is based on the premise that most programs spend 90% of the time in 10% of the code—therefore, the VM only optimizes that 10% of the code and interprets or generates naive code for the rest, where little time is spent.

[3] The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java(TM) platform.

3 Architecture of Walkabout

The Walkabout framework was designed with retargetability in mind. We were interested in supporting binaries for different input and output machines, so we designed the framework to be retargetable for both input and output machines. In our notation, we refer to the input machine as the *source machine* (M_s), and the output machine as the *target machine* (M_t). The framework was designed so that users could instantiate new translators out of the framework for their source and target machines of choice to run on the target machine, which we refer to as the *host machine*.



M_s -> M_t Translator
M_t Code Generator
M_t F$
M_s Interpreter
M_t Optimizer
Dispatcher
Ms binary file

Figure 1: The Architecture of the Walkabout Framework

The architecture of the Walkabout framework borrows from the architecture of most existing dynamic compilation systems. Figure 1 illustrates the architecture of the Walkabout framework. The source binary program is loaded into virtual memory and initially interpreted until a hot path[4] is found. Code is generated for that hot path and placed into a translated instruction cache (called the fragment cache or F$). During code generation, simple optimizations are applied to obtain better code locality. Once the generated code is executed, control transfers to the dispatcher, which decides whether to interpret more code or transfer control to code in the fragment cache. If interpreted, the process repeats. Reoptimization of translated code occurs when a piece of translated code in the fragment cache is executed too often.

Retargetability is supported in the Walkabout framework through the use of specifications: machine descriptions and specifications of the hot path selection methods. The machine descriptions specify the syntax and semantics of machine instructions, and allow the automatic generation of machine-code interpreters (emulators) and instruction encoders.

[4] A hot path is a frequently executed path in a program.

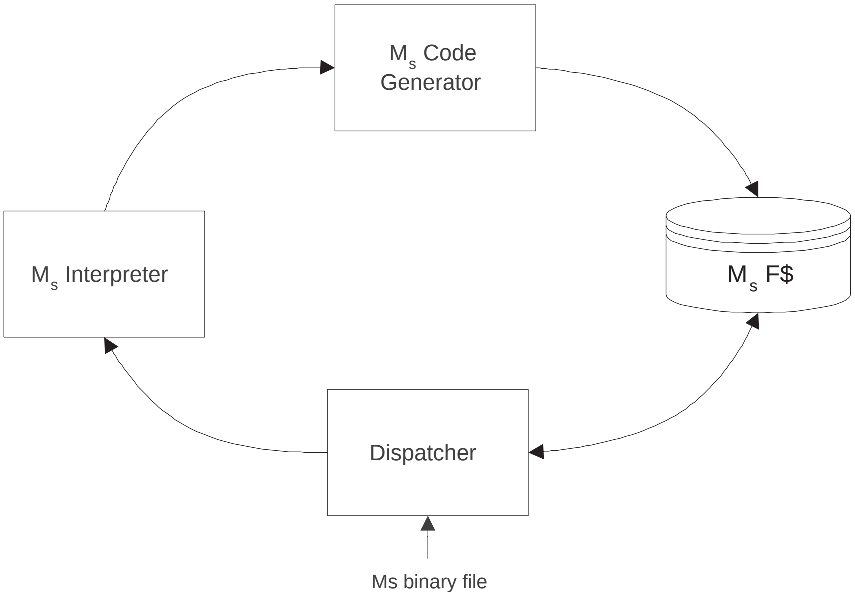


Figure 2: The 2001 Walkabout Framework

The 2001 implementation of the Walkabout framework does not implement the complete framework. The system was built in stages, starting from the interpreters, over a period of 9 months with the help of 3 interns. Figure 2 illustrates the implementation components of the 2001 Walkabout implementation. Next we describe different tools that can be built from this initial implementation.

4 The Interpreter

Interpreters in the Walkabout framework are automatically generated from specifications of syntax and semantics of machine instruction sets. We reused the specifications we had used in the UQBT project [CE00], namely, the New Jersey Machine Code (NJMC) toolkit's SLED descriptions [RF97] and the UQBT's SSL descriptions [CS98].

SLED specifications allow users to specify the mapping between the binary and the assembly representation of a machine instruction set, as well as the machine's registers and names for those registers. SSL specifications allow users to specify the mapping between assembly instructions and their equivalent register transfers, to name new registers and declare overlaps, to define superoperators in the from of macros for commonly set condition codes, and to specify the fetch-execute cycle for the machine.

The combined SLED and SSL specifications provide complete information to generate a user-level interpreter. The user-level restriction is imposed by the SSL descriptions, which describe semantics for user-level instructions only, as per the goals of the UQBT project [CERL01].

The conceptual view of the interpreter generator is illustrated in Figure 3. The interpreter generator takes as input a SLED and an SSL description for a given machine and generates source code for

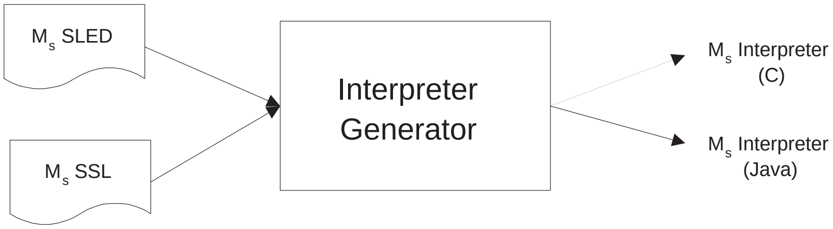


Figure 3: The Interpreter Generator Genemu

an interpreter for the specified machine in either the C or Java language. The generated code can then be compiled into an executable representation, resulting in an interpreter.

The interpreter generator creates interpreters that use either the C or Java language because we were interested in comparing the performance of interpreters implemented in these languages on a relatively fair basis. However, the Java-based interpreters are not as object-oriented as desired, due to implementation issues, as explained in the following section.

4.1 Implementation

The `genemu` tool is the implementation of the interpreter generator in the Walkabout framework. `genemu` relies on the New Jersey Machine Code (NJMC) toolkit, which provides the *matching* statement to decode machine instructions. The syntax of the *matching* statement resembles that of the C language's `switch` statement. It matches the series of bits that identifies an instruction and also extracts the operands and field values for that instruction.

The process used by `genemu` to generate the C or Java language interpreters is illustrated in Figure 4. SLED and SSL files for a machine are parsed by `genemu` and checked for consistency. A matching file (i.e., a `.m` file) is generated, consisting of the core decoder for machine instructions, as well as the associated C language code to implement the semantics of each instruction in predefined interpreter data structures that capture the state of the machine being interpreted. The NJMC toolkit transforms the matching file into C language code. Optionally, a postprocessing phase of the `genemu` tool can transform the C language file into a Java language file, by making transformations on the syntax more than anything else.

The generated C or Java language decoders are then compiled with three pieces of information: file loading and system calls for the operating system of interest (`SysV_Personality`), and machine stubs (`Ms stub`). `genemu`'s implementation of OS support is for ELF binaries for two SystemV operating systems: Solaris(TM) and Linux. System call support for both these OSs is part of `genemu`. Machine stubs are written to initialize the state of the machine being interpreted, that is, the interpreter's internal data structures, at program startup time.

The `genemu` tool can also automatically generate disassemblers in the C and Java language for a given machine; the SLED description is all that is needed in that case.

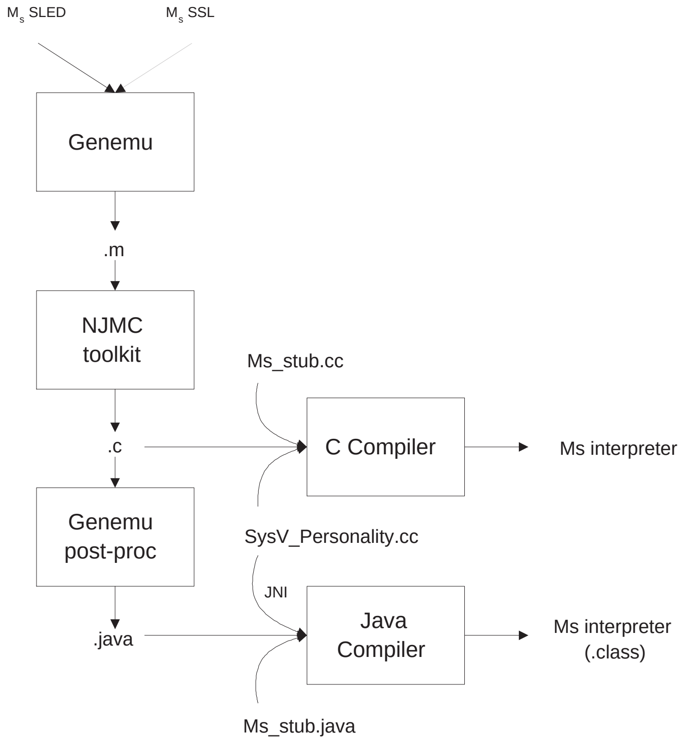


Figure 4: The `genemu` Interpreter Generation Process

4.2 Performance Results

Using `genemu` and the SPARC and x86 architecture's SLED and SSL descriptions, we generated interpreters for the SPARC and x86 architectures running on the Solaris and Linux operating environments respectively. In this section, we report on our experimental results with the C language interpreter for the SPARC architecture.

Figure 5 shows results for the running of the C-language interpreter for the SPARC architecture on a lightly loaded 4 CPU UltraSPARC(R) machine running at 450 MHz per CPU and 4GB of memory. Results are for runs of SPECInt 95 programs, which were compiled for a SPARC V8 machine. For each program, the size of the program in bytes is listed, as well as the interpretation time and the native time to run the programs, in seconds. The native time reported is the average of three runs. The interpretation time reported is the result of one run. The slowdown of the interpreter compared to native code is also shown: this is, on average, approximately 200x.

The Java language interpreters are considerably slower than their C language counterpart. This is partly due to the difficulty of dynamically compiling a program that implements an interpreter. Much of the execution time is spent in a dispatching loop that switches on each instruction, and no significant amount of time is spent in any one path of the `switch` statement. In practice, we observed a 5x slowdown of the Java language version of the interpreter for the SPARC architecture when compared against the unoptimized C language version of the same interpreter, and a 15.5x slowdown when compared against an optimized (O4) version of the C language interpreter, on

Program	Size	Interpreted	Native	Slowdown
go	364,412	57,388	412	139
m88ksim	198,264	47,068	180	261
li	83,168	32,210	181	178
ijpeg	175,268	38,294	187	205
perl	298,296	22,972	137	168
vortex	665,088	40,180	203	198
sieve (3000)	24,452	3,287	15	219
fibo (35)	24,668	184	1.3	141
Mean	229,202	30,198	375.74	189

Figure 5: Performance Results for an Automatically-Generated, C-language Interpreter for the SPARC Architecture

small benchmarks. The results point at a 1000x-3000x slowdown for programs executed by the Java-based interpreter when compared against native code. These results were observed using the Java SDK 1.3 on a SPARC machine.

5 Instrumenting an Interpreter for Hot Path Detection

Some optimizing virtual machines rely on an instrumented interpreter that determines hot paths in the program being interpreted. Native code is then generated for such hot paths.

In the Walkabout framework, we were interested in experimenting with different ways to determine hot paths within an interpreter. This is more challenging in a dynamic translator because binary programs do not include such high-level information as procedure boundaries or loop information. In addition, binary programs are typically less structured and regular than the byte code programs executed by virtual machines, so it can be harder to find natural instrumentation points.

We designed a simple language called INSTR, for instrumentation language, to allow us to instrument the interpreters automatically generated by the `genemu` tool. INSTR allows us to easily create different instrumented interpreters that make use of different instrumentation rules. Figure 6 illustrates how INSTR is used.

The instrumented interpreter generator, `genemu'`, parses SLED and SSL machine descriptions, as well as the INSTR instrumentation description, and generates an interpreter for that machine that can instrument instructions in the way specified in the INSTR specification file. The instrumented interpreter is generated in the C language.

In the Walkabout framework, the INSTR language was used to write a specification for Dynamo's Next Executing Tail tracing method [DB00], as well as other variations of the method. In a similar way, one could specify the method used by the Java HotSpot [GM00] virtual machine, for example,

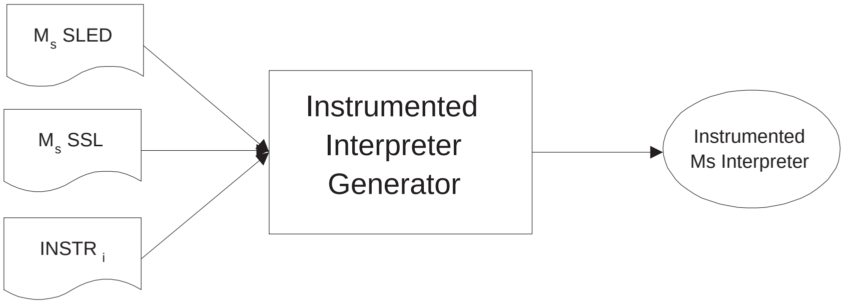

M_s SLED
M_s SSL
INSTR $_i$
Instrumented Interpreter Generator
Instrumented Ms Interpreter

Figure 6: The Instrumented Interpreter Generator

or any other method that was being considered to determine hot paths in an interpreter, so long as it can be specified based on one instruction-at-a-time information.

In the following sections, we describe the INSTR language and give some examples of its use for other applications.

5.1 The INSTR Language

The INSTR language was designed to work in conjunction with an interpreter, therefore, it relied on simple abstractions available in interpreters. An interpreter interprets code one instruction at a time. It also knows about the fetch-execute cycle of the machine being interpreted. Consequently, INSTR's abstractions are centered around individual instructions and the fact that the interpreter has a fetch-execute cycle to decode instructions for the given machine. Further, the interpreters of interest were those automatically generated by the Walkabout framework using the `genemu` tool; therefore, the language required a way to relate to the instruction names that are used in the SLED and SSL specifications from which the interpreters are generated.

Other instrumentation languages used in tools like ATOM [ES95] and Vulcan [SEV01] make available high-level abstractions of the program to developers; the aims of these languages are different than one that needs to be integrated with an automatically generated interpreter, and therefore could not be reused for our purposes.

INSTR allows developers to instrument an instruction at three different points in time:

- before the instruction is fetched,
- before the instruction's semantics are executed, or
- after the instruction's semantics are executed.

Instrumenting in the fetch-execute cycle allows for instrumentation of repetitive actions on all instructions that get fetched, for example, to observe and record each instruction's opcode or to count the number of instructions executed. Instrumenting before or after an instruction's semantics means that such actions are only executed on that particular instruction instead of on each instruction being fetched.

An instrumentation file consists of three main sections:

1. Definition,
2. Fetch-execute cycle, and
3. Support code.

The definition section specifies which instructions are to be instrumented and their corresponding instrumentation code. The fetch-execute cycle section specifies what, if any, commands need to be executed at each iteration of the cycle. The support code section contains support functions used in the instrumentation code; this code is expressed in the C/C++ language. Figure 7 provides the EBNF for the language.

```
specification:      parts+
parts:              definition | support_code
definition:         "DEFINITION" instrm+
instrm:             table | semantics
table:              %STRING "[" SLED_names "]"
semantics:          (%STRING parameter_list instrument_code)+
                    ["FETCHEXECUTE" instrument_code]
parameter_list:     %STRING ("," %STRING)*
instrument_code:    "{" (%action)* "}"
support_code:       "IMPLEMENTATION_ROUTINES" %c_code
```

Figure 7: EBNF for the INSTR Language

In the INSTR language, both `%action` and `%c_code` denote valid C/C++ language code; the difference between them is that `%action` can make use of predefined keywords to refer to fields of an instruction. The keywords and their meaning are:

1. `"SSL_INST_SEMANTICS"`: this keyword denotes the standard semantics of an instruction, as described in the SSL specification file.
2. `"PARAM(" %STRING ")"`: the `"PARAM"` keyword stands for parameter and it is the way to refer to the *value* of a named parameter (i.e., operand) of an instruction. For example, when instrumenting the `BA label` instruction, `PARAM(label)` refers to the value of the branch's label.
3. `"SSL(%" %STRING ")"`: the `"SSL"` keyword denotes one of the SSL-named register locations. For example, `SSL(%pc)` is the location holding the value of the emulated PC register.

5.2 Examples

The INSTR language allows the Walkabout framework to be used to construct instrumentation tools that insert code during interpretation in order to understand the behaviour of running programs. These tools can do basic block counting and profiling. They can also record dynamic memory accesses, branches taken or not, and instruction traces. The data they collect can be used to drive related tools such as pipeline and memory system simulators.

Basic Block Counting

If we want to count the number of basic blocks executed in a program, we need to increment a counter each time an instruction that causes an end-of-basic block condition is reached.

Using INSTR, we can group all the SPARC architecture branching instructions in a table called `branch` and then give semantics to that group of instructions as the instrumentation for such instructions. In the example code, all branching instructions of the kind `branch` will now increment a counter called `BB_count` in the instrumented interpreter and will also maintain their original semantics (as specified in the SSL representation of those instructions).

```
DEFINITION

branch [ "BA", "BN", "BNE", "BE", "BG", "BLE", "BGE", "BL",
         "BGU", "BLEU", "BCC", "BCS", "BPOS", "BNEG", "BVC", "BVS",
         "BNEA", "BEA", "BGA", "BLEA", "BGEA", "BLA", "BGUA",
         "BLEUA", "BCCA", "BCSA", "BPOSA", "BNEGA", "BVCA", "BVSA",
         "RET", "RETL", "CALL", "JMPL" ]

branch label {
   BB_count++;
   SSL_INST_SEMANTICS
}
```

Load Monitor

A load monitor could be specified in the following way. All load instructions of interest are grouped in a `load` table. All load instructions take two operands, the effective address of the load (`eaddr`) and the register where the value is to be loaded to (`reg`). Whenever a load instruction is decoded, if the `monitor_mode` flag is set, the C function `monitor_eaddr` will be called before the semantics of the load instruction is executed by the interpreter. The `monitor_eaddr` function monitors the runtime value of the effective address of the load instruction (referred to as `PARAM(eaddr)`) and records the memory reference of the load instruction (`SSL(%pc)`; i.e., the PC value). The `monitor_eaddr` function would be defined in the support section of the specification file.

```
DEFINITION

load [ "LD", "LDA", "LDD", "LDUH", "LDUHA", "LDUB", "LDUBA",
       "LDSH", "LDSHA", "LDSB", "LDSBA", "LDF", "LDDF", "LDSTUB" ]

load eaddr, reg {
   if (monitor_mode) {
      monitor_eaddr (SSL(%pc), PARAM(eaddr));
   }
   SSL_INST_SEMANTICS
}
```

Edge Counting

In order to instrument all branches of an x86 architecture, we can define a table `jump32s` with the names of all such branches. If we want to count the number of occurrences of edges taken in the program, we can extend the behaviour of the branches by incrementing a counter before the instruction's semantics is executed by the interpreter, as follows

```
DEFINITION

jump32s [ "JVA", "JVNBE", "JVAE", "JVNB", "JVB", "JVNAE", "JVBE",
          "JVNA", "JVC", "JVCXZ", "JVE", "JVZ", "JVG", "JVNLE", "JVGE",
          "JVNL", "JVL", "JVNGE", "JVLE", "JVNG", "JVNC", "JVNE", "JVNZ",
          "JVNO", "JVNP", "JVPO", "JVNS", "JVO", "JVP", "JVPE", "JVS",
          "JMPJVOD" ]

jump32s label
{
    increment_counter(SSL(%pc), PARAM(label));
    SSL_INST_SEMANTICS
}
```

where the function `increment_counter` is defined in the support code section of the specification file. In the above example, all branches take one operand, the target address of the branch instruction, referred to as `label` in the example.

For illustration purposes, we show the support code section for this specification, in which the function `increment_counter` is implemented.

```
IMPLEMENTATION_ROUTINES

#include <map>
#include <iostream>

map <pair<unsigned, unsigned>, int> edge_cnt;

void increment_counter (int addr1, int addr2) {
    pair<unsigned, unsigned> edge =
                             pair<unsigned, unsigned>(addr1, addr2);
    map <pair<unsigned, unsigned>, int>::iterator i;
    if ((i = edge_cnt.find(edge)) == edge_cnt.end())
        edge_cnt[edge] = 1;
    else
        i->second++;
}
```

The `increment_counter` routine makes use of the `edge_cnt` map of edges to execution counts, in order to record occurrences of branch taken edges during execution time. When a count on a taken edge is to be incremented, the routine gets an iterator to traverse the map of pairs (i.e., the edges) and increment the counter for that edge.

At runtime, the instrumented emulator will increase the count on each edge taken during the execution of the input program. A support print routine can then display the number of occurrences of each edge.

6 The PathFinder

The PathFinder is a simple semi-optimizing virtual machine for SPARC machine code that generates SPARC V9 code for hot paths, performs simple optimizations during code generation, and places the generated code in a fragment cache. The PathFinder is a dynamic binary rewriting tool that can be used to experiment with dynamic code optimization. The PathFinder is the core of the 2001 implementation of the Walkabout framework. Figure 8 illustrates the PathFinder's architecture.

The instrumented SPARC V8 interpreter is automatically generated by the `genemu'` tool based on the SLED and SSL specifications for the SPARC instruction set, and several different INSTR specification files. In its present form, the PathFinder tool for the SPARC architecture resembles the dynamic optimizing systems Dynamo [BDB00], Wiggins/Redstone [DGR99] and Mojo [CLCG00], which were written for the PA-RISC, Alpha and x86 platforms, respectively. Part of the objective for the PathFinder was to be able to contrast techniques used in other systems in the context of a retargetable framework. However, the PathFinder does not currently implement a reoptimizer, and so cannot be fully compared experimentally with those systems. Conceptually, all these translators work in similar ways; i.e., they generate target code for hot pieces of source machine code based on some criteria for determining what paths are frequently executed, and perform varying levels of optimization.

The PathFinder's code generator performs code layout optimizations as well as several simple optimizations, which we explain in turn. No intermediate representation is built in the PathFinder; the code generator relies on transforming SPARC V8 assembly instructions directly onto V9 instructions.

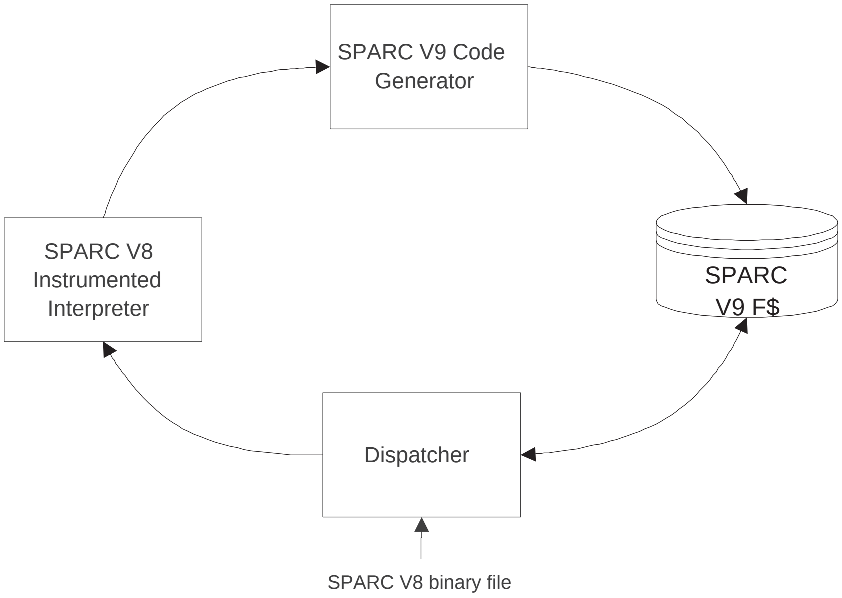


Figure 8: PathFinder: The Implementation of the 2001 Walkabout Framework

The PathFinder's memory system is trivial. We reserve a fixed-sized fragment cache: once that cache is full, it is flushed; no attempt is made to figure out what fragments to keep.

6.1 Code Layout

Code layout is a simple optimization that is achieved by placing basic blocks that execute frequently in sequential fashion, contiguous in memory, to achieve better code locality as well as reduce the number of branches needed in the code.

Figure 9 shows an example of code layout. To the left of the diagram is a control flow graph for a program. The program executes a loop quite a large number of times by following the path ACFGDEA. Note that the loop includes the call to a routine and its return from that routine. The right side of the diagram illustrates how the code is placed in the fragment cache, therefore improving code locality. In the diagram, ACFGDE are placed sequentially in memory. Nodes P_B and P_H are the *exit portals* of this fragment. An exit portal is an exit basic block for a fragment of code. It contains code that allows the program to go back to the interpreter or to another fragment of code (when fragment linking has been implemented, see §6.5).

6.2 Branch Inversion

As part of code layout, inversion of certain branch instructions is necessary to keep basic blocks contiguous in memory. Branch targets that are not part of the trace jump to exit portals in that fragment.

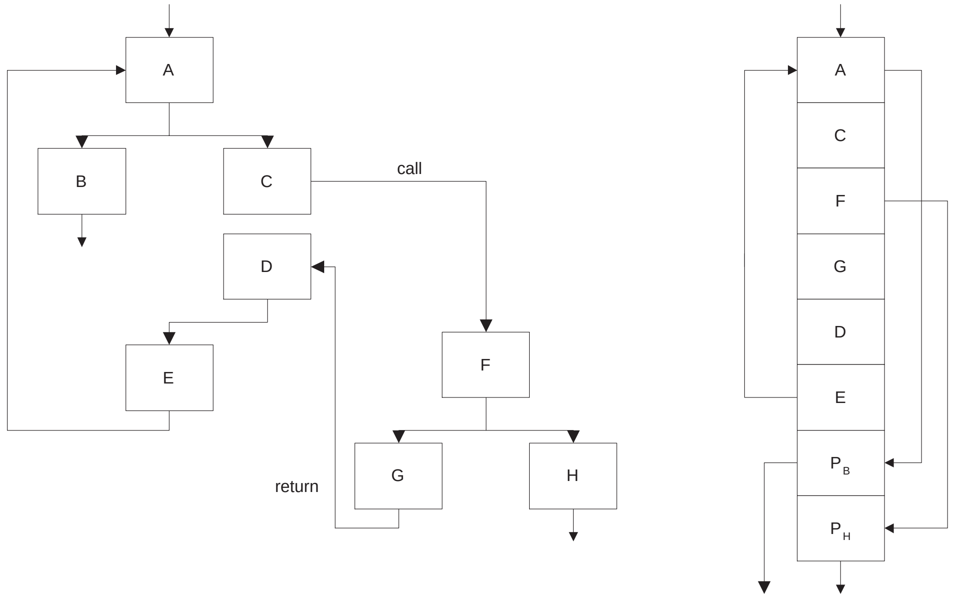


Figure 9: Code Layout Example

The inversion of some delayed control transfer instructions in the SPARC and other architectures is not as straightforward as replacing the branch by its inverse branch, as these instructions may not be antonyms based on the delay slot semantics. In the following code, the annulling branch instruction bne,a at address 0xfe08d10 is to be inverted as the most common behaviour in that program is to branch on not equals. This annulling branch executes the delay slot instruction only if the branch is taken, otherwise it annuls it.

```
Trace:
0xfe08d0c:      cmp  %o0, 0
0xfe08d10:      bne,a   0xfe08d2c
0xfe08d14:      ld  [ %i3 + 0xc ], %i3
...
```

Inverting the branch requires a non-annulling branch on equal, i.e., be, and a nop in the delay slot of that branch, so that the fall through case executes the code that was previously located at addresses 0xfe08d14 and 0xfe08d2c; now located starting at <code_cache+628>. The code fragment looks as follows

```
...
<code_cache+616>:    cmp  %o0, 0
<code_cache+620>:    be  <code_cache+696>
<code_cache+624>:    nop
<code_cache+628>:    ld  [ %i3 + 0xc ], %i3
...
```

6.3 Branch Linking

Branch linking relates to the removal of unconditional branches when the target of the branch is moved to be located immediately following the branch instruction in the fragment cache. These branches typically include the branch always (ba) and branch never (bn) instructions, as well as their annulled counterparts. The following code serves as example, where the branch to address 0x50034 is removed in the fragment cache, and the code at source address 0x74ae0 is placed after the delay slot instruction.

```
0x4fe68:        ba  0x50034
0x4fe6c:        add  %i5, 4, %i5
...
0x50034:        ba  0x74ae0
0x50038:        add  %l0, 4, %l0
0x5003c:        ba  0x7e510
0x50040:        ld  [ %l0 + %i0 ], %g2
```

The SPARC call instruction is a special case of an unconditional branch, one that affects the state of register %o7 when the program counter is written to that register (in order to preserve the return address). In this case, the semantics of the assignment to %o7 is preserved, while the instruction is modified, as per the following example, which includes one call instruction.

```
0x49f98:        ld  [ %i0 + 0x11c ], %o1        ! 0x9211c
0x49f9c:        call  0x60f24
0x49fa0:        mov  %l4, %o0
...
```

The fragment generated for the above trace computes the return address of the source program (0x49f9c), and stores it in register %o7, as per standard call semantics, leading to the following code

```
<code_cache+3432>:   ld  [ %i0 + 0x11c ], %o1
<code_cache+3436>:   sethi  %hi(0x49c00), %o7
<code_cache+3440>:   add  %o7, 0x39c, %o7    ! 0x49f9c
<code_cache+3444>:   mov  %l4, %o0
...
```

Figure 10 shows a sample SPARC assembly code for a program on the left-hand side and its corresponding code fragment. The left-hand side shows in bold face the instructions that belong to a trace in this program. The branch at 0xfe1c970 transfers control to 0xfe1c9cc, the branch at 0xfe1c9e8 transfers control to 0xfe1cad4, and the branch at 0xfe1cad8 jumps back to the start of the trace at 0xfe1c960. In the example, code layout, branch inversion and branch linking has been applied to the code. The right-hand side shows in bold face the branch at 0x200b3ca0, which has been inverted, and the last branch at 0x200b3ccc which jumps to the start of the fragment. The pieces of code at 0x200b3cd4, 0x200b3d00, etc, are all exit portals.

```
...
0xfe1c960:  sll  %i2, 4, %i2
0xfe1c964:  bl  0xfe1ca20
0xfe1c968:  srl  %i5, 1, %i5
0xfe1c96c:  subcc  %i3, %i5, %i3      0x200b3c90:  sll   %i2, 4, %i2
0xfe1c970:  bl  0xfe1c9cc             0x200b3c94:  bl     0x200b3d84
0xfe1c974:  srl  %i5, 1, %i5          0x200b3c98:  srl   %i5, 1, %i5
0xfe1c978:  subcc  %i3, %i5, %i3      0x200b3c9c:  subcc %i3, %i5, %i3
0xfe1c97c:  bl  0xfe1c9a8             0x200b3ca0:  bge   0x200b3d58
0xfe1c980:  srl  %i5, 1, %i5          0x200b3ca4:  srl   %i5, 1, %i5
...                                   0x200b3ca8:  addcc %i3, %i5, %i3
0xfe1c9cc:  addcc  %i3, %i5, %i3      0x200b3cac:  bl     0x200b3d2c
0xfe1c9d0:  bl  0xfe1c9fc             0x200b3cb0:  srl   %i5, 1, %i5
0xfe1c9d4:  srl  %i5, 1, %i5          0x200b3cb4:  subcc %i3, %i5, %i3
0xfe1c9d8:  subcc  %i3, %i5, %i3      0x200b3cb8:  bl     0x200b3d00
0xfe1c9dc:  bl  0xfe1c9f0             0x200b3cbc:  srl   %i5, 1, %i5
0xfe1c9e0:  srl  %i5, 1, %i5          0x200b3cc0:  subcc %i3, %i5, %i3
0xfe1c9e4:  subcc  %i3, %i5, %i3      0x200b3cc4:  add   %i2, 7, %i2
0xfe1c9e8:  ba  0xfe1cad4             0x200b3cc8:  deccc %i4
0xfe1c9ec:  add  %i2, 7, %i2          0x200b3ccc:  bge   0x200b3c90
0xfe1c9f0:  addcc  %i3, %i5, %i3      0x200b3cd0:  tst   %i3
0xfe1c9f4:  ba  0xfe1cad4             0x200b3cd4:  !exit to 0xfe1cae0
0xfe1c9f8:  add  %i2, 5, %i2          ...
...                                   0x200b3d00:  !exit to 0xfe1c9f0
0xfe1cad4:  deccc  %i4                ...
0xfe1cad8:  bge  0xfe1c96a            0x200b3d2c:  !exit to 0xfe1c9fc
0xfe1cadc:  tst  %i3                  ...
0xfe1cae0:  bl,a   0xfe1cae8          0x200b3d58:  !exit to 0xfe1c978
0xfe1cae4:  add  %i3, %i1, %i3        ...
0xfe1cae8:  tst  %l1                  0x200b3d84:  !exit to 0xfe1ca20
...                                   ...
```

Figure 10: Code Layout Example: Sample Trace (in bold face) and Generated Code for the Trace

6.4 Inline Caching

Inline caching is a technique originally developed for Smalltalk virtual machines to cache "in line" a lookup result for a message send call, hence removing the overhead of the system's lookup routine [DS84]. The inlined routine adds, in its prologue, guard code to determine that the receiver type is the expected one.

Inline caching is used in the PathFinder to predict the target address of indirect transfers of control. The technique is simple: given a trace of the targets for such transfers of control at a given point in a program, the most frequent target becomes the predicted one.

For example, an indirect jump on the contents of register `%g1` at offset 0xf4

```
           jmp %g1 + 0xf4
```

can be transformed into an unconditional branch to the predicted location (labelled `predicted` in the below code). Lets say that the target address for this jump is predicted to be that stored in `predicted_val`, then, the code at the predicted location ensures that jumps reaching this code are the right ones, i.e., it compares the predicted value against the expected value. This code transformation avoids the indirect transfer of control for the most common destination address of the branch

```
predicted: add %g1, 0xf4, scratch1
           set predicted_val scratch2
           cmp scratch1, scratch2
           bne exit_scratch1
           ...
```

In a similar way, indirect calls and returns with no corresponding call instructions in the fragment, are treated as indirect jumps. In the SPARC architecture, return instructions are indirect transfers of control on the contents of register `%i7`, which holds the address of the instruction that invoked the procedure. The following code shows a trace that has a return instruction but no corresponding call instruction in the trace itself. Once the return is executed, execution continues at address `0x4fb50`.

```
0x4fc40:        sll  %i0, 2, %g2
0x4fc44:        mov  1, %o3
0x4fc48:        ld  [ %g2 + %o5 ], %g2
0x4fc4c:        ld  [ %o1 + %o5 ], %g3
0x4fc50:        cmp  %g3, %g2
0x4fc54:        be  0x4fc64
0x4fc58:        cmp  %o3, 0
0x4fc5c:        ret                   ! jmpl %i7+8, %g0
0x4fc60:        restore  %g0, 0, %o0

0x4fb50:        cmp  %o0, 0
0x4fb54:        bne,a   0x4fb78
0x4fb58:        clr  %o0
...
```

The following code fragment shows how the one return instruction at address `0x4fc5c` in the trace, is replaced by 5 instructions in the fragment cache, starting at address `<code_cache+3932>`. This is because a guard has been placed to check that the correct jump targets reach the address `<code_cache+3956>`, as this address is now contiguous to the rest of the trace code.

```
<code_cache+3904>:   sll  %i0, 2, %g2
<code_cache+3908>:   mov  1, %o3
<code_cache+3912>:   ld   [ %g2 + %o5 ], %g2
<code_cache+3916>:   ld   [ %o1 + %o5 ], %g3
<code_cache+3920>:   cmp  %g3, %g2
<code_cache+3924>:   be   <code_cache+4192>
<code_cache+3928>:   cmp  %o3, 0
<code_cache+3932>:   add  %i7, 8, %g5
<code_cache+3936>:   sethi  %hi(0x4f800), %g6
<code_cache+3940>:   add  %g6, 0x350, %g6      ! 0x4fb50
<code_cache+3944>:   sub  %g5, %g6, %g6
<code_cache+3948>:   brnz,pn   %g6, <code_cache+4140>
<code_cache+3952>:   restore  %g0, 0, %o0

<code_cache+3956>:   cmp  %o0, 0
<code_cache+3960>:   bne,a  <code_cache+4088>
<code_cache+3964>:   clr  %o0
...
```

6.5 Fragment Linking

Fragment linking is the process of joining one fragment to another, so as to avoid going through the dispatcher; i.e., it reduces context switching. Fragment linking can be done during code generation of a fragment or as a reoptimization which patches an existing portal to a new fragment.

During code generation, when generating a fragment where one of its exits leads to an address that is the start of another fragment, instead of creating an exit portal, a direct jump to the second fragment can be generated instead.

When performing reoptimization of code, fragment code that shows a frequently-executed behaviour can generate new hot traces (note that these are traces of code in the fragment cache). In these traces, exit portals can be patched so that the flow of control does not need to go through the dispatcher any longer.

6.6 Peephole Optimizations of V7 Code

The SPARC V7 architecture did not have integer multiply or division instructions at the hardware level, hence, library routines were made available to implement these instructions in software by making use of several additions and/or subtractions. Post V7 machines implement the multiply and division instructions in hardware.

To this day, there are natively compiled SPARC binaries that make use of the V7 library routines to perform multiplication and division. ISVs tend to compile for the lowest-common denominator machine in many cases.

A simple peephole optimization for such binaries is to transform the V7 library calls and associated instructions that set up its parameters and return value, to the equivalent multiply or divide instruction for the host platform.

For example, if multiplying the values of registers `%i0` and `%i1`, and expecting the result into register `%i0`, the following V7 code

```
    mov     %i0, %o0
    mov     %i1, %o1
    call    .umul
    nop
    mov     %o0, %i0
```

can be replaced by the more simple and efficient V8/V9 instruction

```
    umul    %i0, %i1, %i0
```

6.7 Experimental Results

The PathFinder was tested against SPEC95 benchmarks. Note that no reoptimization of code is performed by the PathFinder, so these results can be improved upon. We report on experimental results using Dynamo's next executing tail (NET)[5] trace selection method [DB00], a 1 MB cache size and the following optimizations: code layout, branch inversion, branch linking and fragment linking. Figure 11 shows the results. For each program, its static size in bytes is given, as well as its user execution time running on the interpreter, the PathFinder, and on the native machine. Results were obtained on a lightly loaded 4 CPU UltraSPARC II machine at 450 MHz per CPU and 4 GB of main memory.

Except for `compress` and `sieve`, which have a tight loop that performs the bulk of the program's work, the other benchmarks show slowdowns when compared against native execution runs. These results were collected in a system that is not yet tuned for performance; simple optimizations such as constant propagation, inline caching, and better fragment linking are missing. Of interest, benchmarks `go`, `ijpeg` and `vortex` behave badly; these benchmarks are the ones in which the Dynamo system bails out as the system cannot accurately predict which paths to compile. The

[5] The NET method maintains counts only for executed targets of backward taken branches. Once the counter of any such target exceeds a threshold, the next executing path is predicted by collecting that information on the next iteration of the loop in a trace buffer, where its first element is named the trace-head. The end-of-trace conditions used by NET are:

1. the target of the backwards branch is to the trace-head,
2. the current instruction is a backward taken branch (i.e., a new start-of-trace), or
3. the history buffer has reached a size limit.

Program	Static Size	Interpreter Time	PathFinder Time	Native Time
compress	85,572	n/a	74	76
go	364,412	57,388	3,717	412
m88ksim	198,264	47,068	709	180
li	83,168	32,210	1,821	181
ijpeg	175,268	38,294	1,816	187
perl	298,296	22,972	338	137
vortex	665,088	40,180	35,905	203
sieve(3000)	24,452	3,287	8.3	14.6

Figure 11: Experimental Results Using the Interpreter and the PathFinder, Without Performing Optimization of the Code

PathFinder does not implement a bail out option. In the case of `vortex`, the execution time in the PathFinder approaches that of running the same program in interpretation mode.

Two classes of programs showed bad behaviour using this trace selection method: recursive programs and large `switch` statement-based programs.

Recursive programs do not have a back branch that can be easily detectable as an end-of-trace condition, hence, no trace is found to be hot in a method like NET. Some existing virtual machines, such as the Self VM, used inlining of recursive methods to optimize for recursive programs. Inlining was done once the count on a method entry became hot; in fact, the Self version of highly recursive programs such as Fibonacci and Takeuchi runs much faster than the native C version of the same program [Hol94]. Present day Java virtual machines do not seem to perform inlining of recursive methods. Note that inlining of procedures in machine code programs is not as straightforward as inlining of methods in VM-based languages such as Self and Java. This is because the boundaries of a procedure in a binary program are not well defined, unlike their VM-based languages.

Figure 12 shows results of running a highly recursive program, Fibonacci, using two different trace selection methods in the PathFinder: NET and a variation which we call “recursive”. The recursive method keeps track of the stack levels formed in the trace, and adds a new end-of-trace condition to the NET method: stop building a trace after 5 levels of stack are in the trace (i.e., 5 levels of call or returns). In this way, calls to the procedure Fibonacci are replaced by the setting up of the `%o7` register. The stack frames themselves cannot be removed as there may be exits from the interpreter.

Program	Static Size	Scheme	PathFinder Time	Native Time
fibo (35)	24,668	NET	189s	1.3s
fibo (35)	24,668	Recursive	20s	1.3s

Figure 12: Experimental Results for Problematic Programs Using the PathFinder configured with NET Trace Selection Method

The results in Figure 12 show that the version of the PathFinder using the NET approach behaves just as slowly as the interpreter on its own; i.e., 189s vs 184s (see Figure 5). The results also show considerable improvement in the execution time when using the procedure inlining method.

Programs where the core of the execution time is spent jumping between different `switch` arms of the statement do not normally exhibit good behaviour in VMs unless particular arms of the `switch` statement could be compiled. Using the NET method, the PathFinder detects a hot instructions; however, as the hot trace is determined by executing the next iteration of the loop, the trace that is collected for this type of programs tends to be incorrect most of the times.

7 Debugging Support

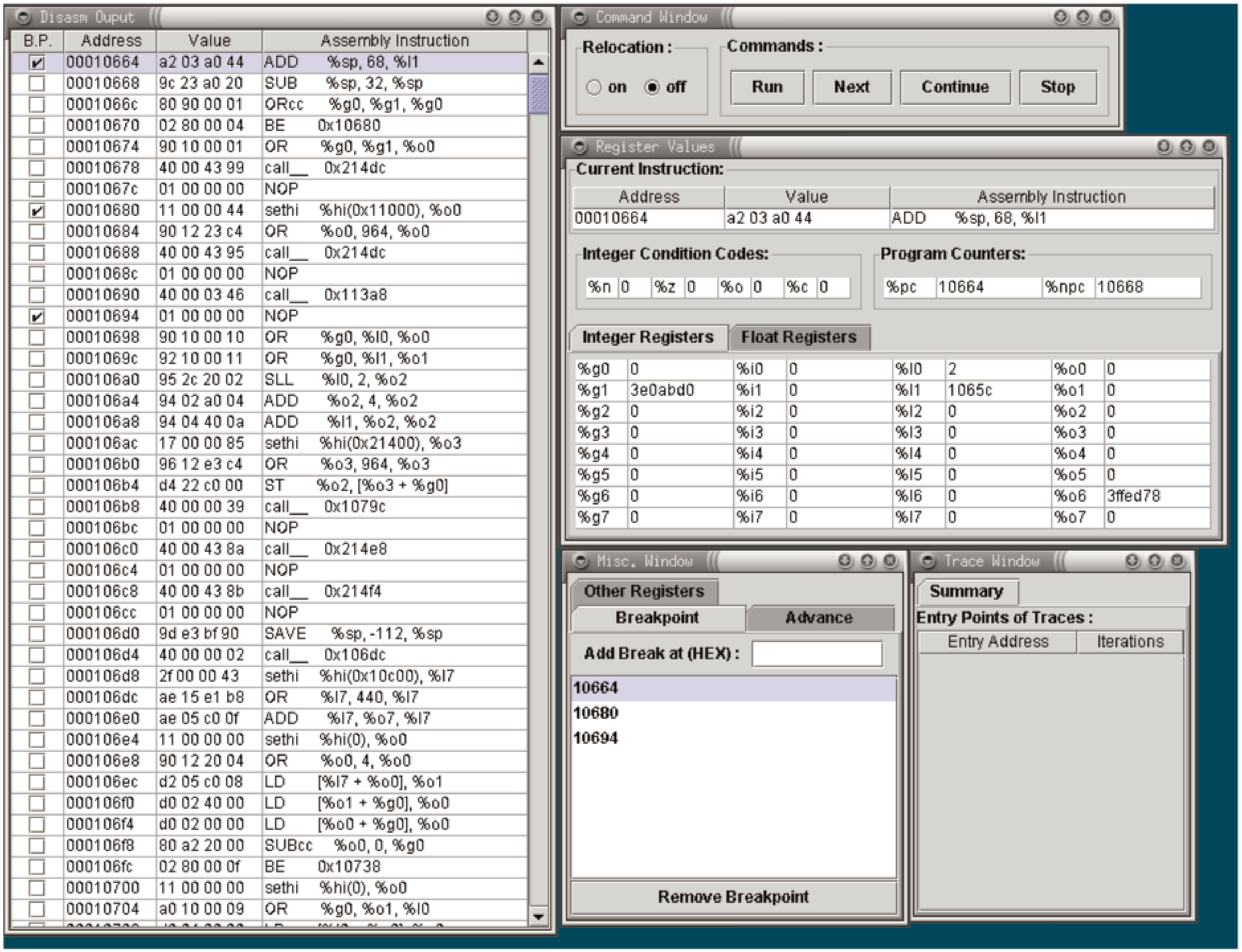
One of the goals of the Walkabout project was to provide better debugging support than its UQBT counterpart. A debugger was built to integrate with the other components of the Walkabout system, relying on the automatic generation of the disassembler and the interpreter as its core components.

The Walkabout debugger is a graphical Java language tool that provides several windows to display the assembly instructions of the program, as well as its state (i.e., register contents). Users can set breakpoints and run the program to a given state.

Programs are run on the C language version of the interpreter. Both the interpreter and the debugger run in separate processes and communicate through a socket, allowing the debugger to remain active even when the interpreter has crashed. The disassembler that is used is the Java language version of the Walkabout disassembler.

Figure 13 shows a view of the Walkabout debugger. Five windows display different information about the state of the program:

- The disassembly window (Disasm Output) shows the disassembly of the program. The first column allows users to set breakpoints, the second column shows the virtual address where the instruction is loaded, the third column displays the instruction in hexadecimal format, and the last in assembly format.
- The command window allows simple debugging control, basically, to (re)start the program run, to step one instruction at a time, to continue program execution until a breakpoint or the end of program is met, or to stop program execution. Relocation in memory of the program can also be set.
- The register values window displays the state of the registers, condition codes and program counters of the machine after execution of each instruction, as well as the details of the current instruction.
- The miscellaneous window displays breakpoint information, and
- The trace window displays a hot trace obtained by running the PathFinder tool.



B.P.	Address	Value	Assembly Instruction	
☑	00010664	a2 03 a0 44	ADD	%sp, 68, %l1
□	00010668	9c 23 a0 20	SUB	%sp, 32, %sp
□	0001066c	80 90 00 01	ORcc	%g0, %g1, %g0
□	00010670	02 80 00 04	BE	0x10680
□	00010674	90 10 00 01	OR	%g0, %g1, %o0
□	00010678	40 00 43 99	call__	0x214dc
□	0001067c	01 00 00 00	NOP	
☑	00010680	11 00 00 44	sethi	%hi(0x11000), %o0
□	00010684	90 12 23 c4	OR	%o0, 964, %o0
□	00010688	40 00 43 95	call__	0x214dc
□	0001068c	01 00 00 00	NOP	
□	00010690	40 00 03 46	call__	0x113a8
☑	00010694	01 00 00 00	NOP	
□	00010698	90 10 00 10	OR	%g0, %l0, %o0
□	0001069c	92 10 00 11	OR	%g0, %l1, %o1
□	000106a0	95 2c 20 02	SLL	%l0, 2, %o2
□	000106a4	94 02 a0 04	ADD	%o2, 4, %o2
□	000106a8	94 04 40 0a	ADD	%l1, %o2, %o2
□	000106ac	17 00 00 85	sethi	%hi(0x21400), %o3
□	000106b0	96 12 e3 c4	OR	%o3, 964, %o3
□	000106b4	d4 22 c0 00	ST	%o2, [%o3 + %g0]
□	000106b8	40 00 00 39	call__	0x1079c
□	000106bc	01 00 00 00	NOP	
□	000106c0	40 00 43 8a	call__	0x214e8
□	000106c4	01 00 00 00	NOP	
□	000106c8	40 00 43 8b	call__	0x214f4
□	000106cc	01 00 00 00	NOP	
□	000106d0	9d e3 bf 90	SAVE	%sp, -112, %sp
□	000106d4	40 00 00 02	call__	0x106dc
□	000106d8	2f 00 00 43	sethi	%hi(0x10c00), %l7
□	000106dc	ae 15 e1 b8	OR	%l7, 440, %l7
□	000106e0	ae 05 c0 0f	ADD	%l7, %o7, %l7
□	000106e4	11 00 00 00	sethi	%hi(0), %o0
□	000106e8	90 12 20 04	OR	%o0, 4, %o0
□	000106ec	d2 05 c0 08	LD	[%l7 + %o0], %o1
□	000106f0	d0 02 40 00	LD	[%o1 + %g0], %o0
□	000106f4	d0 02 00 00	LD	[%o0 + %g0], %o0
□	000106f8	80 a2 20 00	SUBcc	%o0, 0, %g0
□	000106fc	02 80 00 0f	BE	0x10738
□	00010700	11 00 00 00	sethi	%hi(0), %o0
□	00010704	a0 10 00 09	OR	%g0, %o1, %l0

Relocation :

○ on ● off

Commands :

Current Instruction:

Address	Value	Assembly Instruction	
00010664	a2 03 a0 44	ADD	%sp, 68, %l1

Integer Condition Codes:

%n	0	%z	0	%o	0	%c	0

Program Counters:

%pc	10664	%npc	10668

%g0	0	%i0	0	%l0	2	%o0	0
%g1	3e0abd0	%i1	0	%l1	1065c	%o1	0
%g2	0	%i2	0	%l2	0	%o2	0
%g3	0	%i3	0	%l3	0	%o3	0
%g4	0	%i4	0	%l4	0	%o4	0
%g5	0	%i5	0	%l5	0	%o5	0
%g6	0	%i6	0	%l6	0	%o6	3ffed78
%g7	0	%i7	0	%l7	0	%o7	0

Add Break at (HEX) :

10664
10680
10694

Entry Points of Traces :

Entry Address	Iterations

Figure 13: A View of the Walkabout Debugger

8 Experience

The Walkabout framework was designed and partially implemented over a period of 9 months by 2 researchers and 3 interns. This work was done at the same time as the final experiments in the static binary translation framework UQBT [CERL02]. The total effort was 3.5 researcher-months and 11 intern-months.

Our experiences with the implementation and results of the Walkabout framework are positive. These results were achieved thanks to a design that supported retargetability as well as a separation of machine dependence concerns. Retargetability was achieved through the use of specifications for both machine instruction sets and instrumentation.

The implementation of the Walkabout framework was staged. First, disassemblers were generated, then machine code interpreters (emulators), followed by instrumented emulators that determine hot paths in a program, followed by the creation of a binary-code rewriting tool called PathFinder. Many of the components of these tools were generated automatically.

The machine code emulator generator was a big win. We were able to reuse existing syntactic and semantic descriptions for the SPARC and x86 instruction sets and generate, automatically, emulators for these machines. These emulators were able to run existing programs including the SPEC95 benchmark suite. The emulator generator was able to generate two different versions of each emulator: one in the C language and another in the Java language. This allowed us to compare how well those languages worked for this purpose. The use of the emulator generator allowed us to quickly experiment with multiple machines and to generate correct emulators, as extensive testing in the past had ensured that the specifications were correct.

The development of a new specification language for instrumenting machine code emulators, INSTR, was also a big win. The emulator generator was constructed in such a way that it allowed easy integration of instrumentation support code at different points in a program. The instrumentation language itself was simple and made it easy to specify new profiling schemes. It could also be used for other kinds of instrumentation: e.g., traditional EEL-like instrumentation of binaries. It was not restricted to use in a dynamic interpreter/reoptimizer.

Once we could specify and generate instrumented machine code emulators, the next step was to decide how to determine a program's hot paths. It proved straightforward to specify this criteria. Code generation for these hot paths on the SPARC architecture was done by reusing the syntactic instruction specifications for the SPARC architecture. The hot path analysis code was only concerned with deciding what types of transformations to apply to the source machine code, which simplified development time.

The current implementation of the Walkabout framework is incomplete. The PathFinder tool does not reoptimize code, which is needed to improve the performance of running applications.

Since PathFinder does no optimizations at this time and, being mostly automatically generated, is not tuned, we did not expect it would consistently improve the performance of programs. This is confirmed by the performance results we obtained. The runtime of most programs is somewhere between native code and interpreted code. Some programs require close to native time, others run

10 or more times slower than native code, and others run close to interpretation time (200x slower in the Walkabout generated interpreters). Optimizations and fine tuning the code generated by PathFinder would improve these results.

The graphical Walkabout debugger proved to be a help in debugging the system. It was implemented using the Java language and relied upon several components which were automatically generated from the Walkabout framework. This includes the Java version of the disassembler, the C version of the interpreter, and the C version of the PathFinder's hot trace generator.

9 Conclusions

This report describes the design of the Walkabout framework, a dynamic binary translation framework designed to simplify experiments with binary manipulation ideas and based on the use of specifications to simplify retargetability.

The 2001 implementation of the Walkabout framework provides mechanisms to automatically generate machine code interpreters (emulators) and disassemblers in both the C and Java languages. It also supports the automatic creation of instrumented interpreters in the C language by the use of a new specification language called INSTR. Finally, Walkabout includes a general-purpose binary rewriting tool called PathFinder that can be used to implement, for example, binary code reoptimizers. The Walkabout framework can generate tools for the SPARC and x86 platforms.

Acknowledgments

The authors would like to thank Nathan Keynes for the implementation of the interpreter generator `genemu`, and Bernard Wong for the implementation of the debugger. We would also like to thank Mario Wolczko for suggestions in ways to improve the presentation of this report.

The Walkabout distribution is available online under an open source license. Please refer to the Walkabout home site for more information about this project and for links to the distribution:
`http://research.sun.com/walkabout`

References

[ABD$^+$97] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: Where have all the cycles gone? Technical Report SRC Technical Note 197-016a, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 1997. `http://www.research.digital.com/SRC/`.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, Canada, June 2000. ACM Press.

[CE00] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, March 2000.

[CERL01] C. Cifuentes, M. Van Emmerik, N. Ramsey, and B. Lewis. The University of Queensland Binary Translator (UQBT) framework, December 2001. Documentation from the UQBT open source distribution, available from `http://www.itee.uq.edu.au/csm/uqbt.html`.

[CERL02] C. Cifuentes, M. Van Emmerik, N. Ramsey, and B. Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical Report TR-2002-105, Sun Microsytems Laboratories, Palo Alto, CA 94303, January 2002.

[CK93] R.F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI TR-93-12, Sun Microsystems Laboratories, Inc., July 1993.

[CLCG00] W. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, December 2000.

[CS98] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *Proceedings of the International Workshop on Program Comprehension*, pages 126–133, Ischia, Italy, 24–26 June 1998. IEEE CS Press.

[DB00] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, USA, November 2000. ACM Press.

[DGR99] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Redstone: An on-line program specializer. Slides compendium for Hot Chips 11. Stanford, CA, August 1999.

[Dig95] Alpha migration tools. Freeport Express. `http://www.support.compaq.com/amt/freeport/index.html`, 1995.

[DS84] Peter Deutsch and Alan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM Press, January 1984.

[EA96] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. Technical Report RC 20538, IBM, IBM T.J. Watson Research Center, Yorktown Heights, New York, August 1996.

[ES95] A. Eustace and A. Srivastava. ATOM a flexible interface for building high performance program analysis tools. In *Proceedings USENIX Technical Conference*, pages 303–314, January 1995. Also as Digital Western Research Laboratory Technical Note TN-44, July 1994.

[FN96] S. Fordin and S. Nolin. *Wabi 2: Opening Windows*. Sun Microsystems Press, 1996.

[GM00] Robert Griesemer and Srdjan Mitrovic. A compiler for the Java HotSpot virtual machine. In László Böszörményi, Jurg Gutknecht, and Gustav Pomberger, editors, *The School of Niklaus Wirth: The Art of Simplicity*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2000.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[HBG$^+$97] U. Hölzle, L. Bak, S. Grarup, R. Griesemer, and S. Mitrovic. Java on steroids: Sun's high-performance Java implementation. Proceedings of HotChips IX, August 1997.

[HH97] R.J. Hookway and M.A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[HMR96] P. Hohensee, M. Myszewski, and D. Reese. Wabi CPU emulation. Proceedings Hot Chips VIII, 1996.

[Hol94] Urs Holzle. Adaptive optimization for SELF: Reconciling high performance with exploratory programming. Thesis CS-TR-94-1520, Stanford University, Department of Computer Science, August 1994.

[Kla00] A. Klaiber. *The Technology Behind Crusoe^{TM} Processors*. Transmeta Corporation, 3940 Freedom Circle, Santa Clara, CA 95054, January 2000. White Paper.

[Pro01] M. Probst. Fast machine-adaptable dynamic binary translation. In *Proceedings of the Workshop on Binary Translation*, Barcelona, Spain, September 2001.

[PVC01] Michael Paleczny, Chnstopher Vick, and Cliff Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the $Java^{TM}$ Virtual Machine Research and Technology Symposium (JVM-01)*, pages 1–12, Monterey, USA, April 23–24 2001. USENIX Association.

[RF97] N. Ramsey and M. Fernández. Specifying representations of machine instructions. *ACM Transactions of Programming Languages and Systems*, 19(3):492–524, 1997.

[RHWG95] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: the SimOS approach. *IEEE Parallel and Distributed Technology*, pages 34–43, 1995.

[SEV01] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, One Microsoft Way, Redmond, WA 98052, April 2001.

[Tho96] T. Thompson. An Alpha in PC clothing. *Byte*, pages 195–196, February 1996.

[UC00] D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 30–40, Boston, MA, January 2000. ACM Press.

[US87] D. Ungar and R.B. Smith. SELF: The power of simplicity. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 227–241. ACM Press, October 1987.

[WR96] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems (Sigmetrics)*, Philadelphia, USA, 1996. ACM, ACM Press.

[ZT00] C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer*, 33(3):47–52, March 2000.

About the Authors

Cristina Cifuentes is a Senior Staff Engineer at Sun Microsytems Laboratories in Mountain View, California, where she investigates techniques and applications of binary translation. Cristina has published in the areas of binary translation, program comprehension, software maintenance, compiler construction, reverse engineering, decompilation, copyright and legal aspects of computing. She has co-edited two books, given invited lectures worldwide on various topics, and has served on the program committee of numerous conferences and workshops. Cristina was principal investigator of the Walkabout, UQBT, and dcc projects. Prior to joining Sun Microsystems Laboratories in July 2000, she held academic positions at The University of Queensland and The University of Tasmania, Australia. Cristina obtained a Ph.D. from the Queensland University of Technology, Australia, in 1995.

Brian Lewis is a Senior Staff Software Engineer in the Programming Systems Lab of Intel's Microprocessor Research Labs. His interests include binary translation as well as virtual machine and programming language implementation. Previously at Sun, he implemented automatic checkpointing in a high-performance Java virtual machine. He implemented the Tcl 8.0 and the Clarity C++ virtual machines. He also developed monitoring and debugging tools for the Spring distributed operating system. Prior to joining Sun, Brian worked at Olivetti and Acorn Research, where he implemented a user interface toolkit and runtime software. At Xerox, he led the team that developed the Shared Books distributed, multi-user publication management system. He implemented portions of the Mesa programming system and developed software version management tools. He received the Ph.D. in Computer Science from the University of Washington.

David Ung is a Ph.D. student at The University of Queensland in Brisbane, Australia, where he works in the area of dynamic binary translation and optimization. His areas of interest include binary translation, emulation, compilers, reverse engineering and mobile computing. David graduated from the same university in 1996 with First Class Honours in Computer Science. He was an intern at Sun Microsystems Laboratories in 1997 and 2001, where he worked on object cache simulation and the Walkabout project.