Walkabout
A Framework for Experimental Dynamic Binary Translation

Investigators
Cristina Cifuentes
Brian Lewis

Interns
May - Aug 2001: David Ung
May - Aug 2001: Bernard Wong
Jan - Jun 2001: Nathan Keynes

January 2002

Abstract

Dynamic compilation techniques have found a renaissance in recent years due to their applicability to performance improvements of running Java-based code. Techniques originally developed for object-oriented language virtual machines are now commonly used in Java virtual machines and Java just-in-time compilers. Application of such techniques to the process of binary translation has also been done in recent years, mainly in the form of binary optimizers rather than binary translators.

The Walkabout project proposes dynamic binary translation techniques based on properties of re-targetability, 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 ideas in related areas, such as interpreters, instrumentation tools, optimization and more.

Contents

List of Figures

Preface

The Walkabout project was an initial investigation conducted in 2001 at Sun Labs in relation to dynamic binary translation. The project ran for 9 months and was led by Cristina Cifuentes, with input from 3 different interns; Nathan Keynes, David Ung and Bernard Wong, and one other researcher, Brian Lewis.

Walkabout was built based on ideas from the UQBT project (see `http://www.itee.uq.edu.au/csm/uqbt.html`, and as such aimed at retargetable experimentation through specification of features of machines. In fact, the New Jersey Machine Code SLED specifications and the UQBT SSL specifications were reused to automatically generate disassemblers and interpreters for the SPARC and x86 architectures.

We hope that others will make use of the framework; many types of experimentation can be done with the current design and its partial implementation.

Cristina Cifuentes
Mountain View, California
14 Jan 2002

Chapter 1

Introduction

Design: Cristina [2001]; Documentation: Cristina, Brian [Jan 2002]

Binary translation, the process of translating binary executables[1] makes it possible to run code compiled for source platform M_s on destination platform M_d. Unlike an interpreter or emulator, a binary translator makes it possible to approach the speed of native code on machine M_d. Translated code may run more slowly than native code because low-level properties of machine M_s must often be modeled on machine M_d. For example, the Digital Freeport Express translator [8] simulates the byte order of SPARC architecture, and the FX!32 translator [24, 15] 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, dynamic binary translation framework for experimentation with dynamic translations of binary code. The framework grew out of the UQBT framework [4, 3, 5], by taking what we had learned in the areas of retargetability of binary code and separation of machine-dependent from machine-independent concerns, and applying such techniques to the new dynamic framework. Clearly, the choice of transformations on the code would need to be different due to the differences between dynamic and static translations.

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 plan to

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

make use of both types of transformations, determining when it is safe to make use of native target features.

One question that is hard to answer before experimenting in a system is that of the choice of intermediate representation. In the UQBT system we made use of RTLs and HRTLs; the former being a register transfer language that made explicit every transfer of control, and the latter being a high-level register transfer language that resembled simple imperative languages, where control transfers are made explicit. It is unusual for a binary translation system to make use of two different representations for instructions.

Other binary translation systems have made use of the assembly language as the intermediate representation, mainly due to the fact that such systems were generating code for the same machine (i.e. they were optimizers of binary code rather than binary translators per se). Such systems include Dynamo [1], Wiggins/Redstone [19], and Mojo [2].

For Walkabout, we initially use assembly language and we plan to use RTL as the next step, though we would like to experiment with its suitability and ease of translation into a target representation, after all, RTLs are still machine-dependent.

The goals of the project are:

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

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

1.2 Architecture

The architecture of the Walkabout framework borrows from the architecture of most existing dynamic compilation systems such as those for the object-oriented languages Smalltalk [11, 6], SELF [26, 14] and Java [13, 12, 18]. The idea is simple. Based on the premise that most programs

spend 90% of the time in 10% of the code, the dynamic compilation system should only consider compiling that 10% of the code and interpret the rest of the code base, as it is not executed too often.

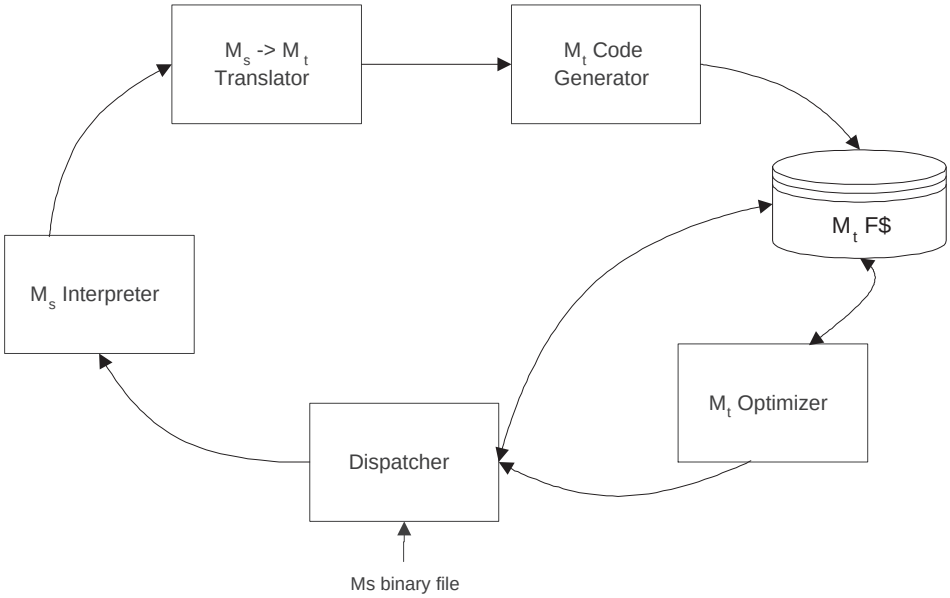

Figure 1.1: The Architecture of the Walkabout Framework

Figure 1.1 illustrates the architecture of the system. The source binary program is initially interpreted until a hot path is found. Code is generated for that hot path and placed into a (translated) instruction cache (called fragment cache or F$ in our notation). Once the generated code is executed, control transfers to the interpreter to interpret more code, and the process repeats for pieces of code that have not been interpreted or translated as yet. For pieces of code that have been translated, the translated version in the instruction cache is executed instead. Further, if a particular piece of translated code is executed too often, the code can be reoptimized and new, efficient code can be generated.

The 2001 Walkabout implementation does not implement the complete framework. This document and the present open source release are the results of a 9-month experiment conducted with interns, hence, only parts of the system are in place. Figure 1.2 illustrates the 2001 Walkabout implementation. As can be seen, code generation was for the same family of machines, the SPARC architecture in this case, where SPARC V9 code was generated for SPARC V8 source binaries.

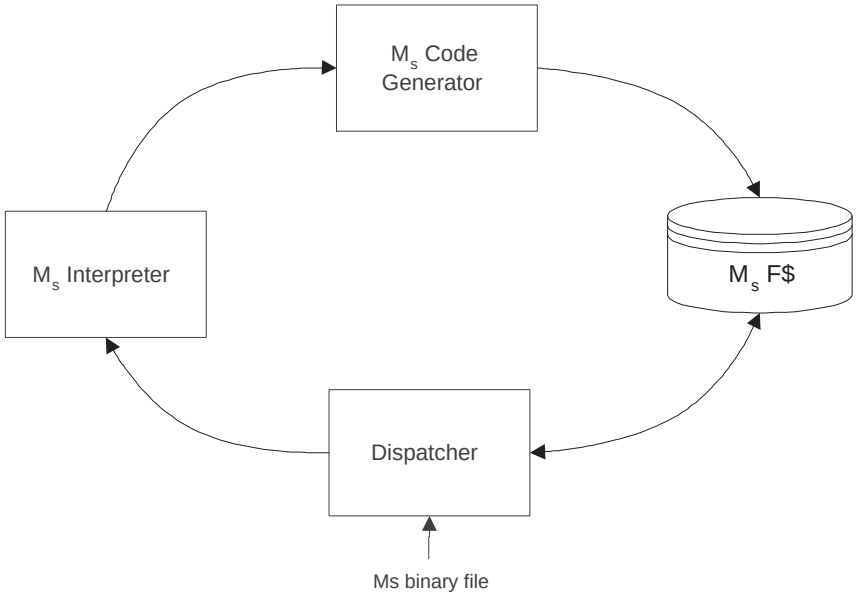

Figure 1.2: The 2001 Walkabout Framework

Applications of the Walkabout Framework

The Walkabout framework can be used to build dynamic translators having a wide range of uses. For example, it supports the construction of analysis and instrumentation tools that insert code during translation in order to understand the behavior 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. Systems using dynamic translation for instrumentation include ATOM [23] and Vulcan [22].

Walkabout can also be used to build optimizers: dynamic translators that improve the performance of programs. Several examples of program optimizers were given above. Also, Schnarr and Larus [20] describe how rescheduling legacy code for newer processors with different pipelines can significantly improve performance. Other applications of Walkabout include machine emulators and program checkers. Machine emulators give executing programs the illusion that they are running on a different machine. They can be used to run legacy programs on newer hardware and to simulate new machines on existing hardware. An example of the latter is the Daisy [9] system. Program checkers execute programs while continuously checking that they operate correctly or safely: for example, that they reference only allowed memory locations, or execute only allowed system calls.

As an example of the latter, the STRATA [21] dynamic translation system has been used to enforce a number of different software security policies.

1.2.1 The Interpreter Generator

We always thought that the UQBT machine descriptions for syntax and semantics of machine instructions were complete enough to support the generation of interpreters for user-level code. The user-level code restriction is imposed by the SSL descriptions, which only describe user-level instruction semantics. This decision was inline with the goals of the UQBT project.

We took the syntactic (SLED) and semantic (SSL) descriptions for the SPARC and x86 architectures and experimented with the automatic generation of interpreters for these two machines. Figure 1.3 illustrates the process.

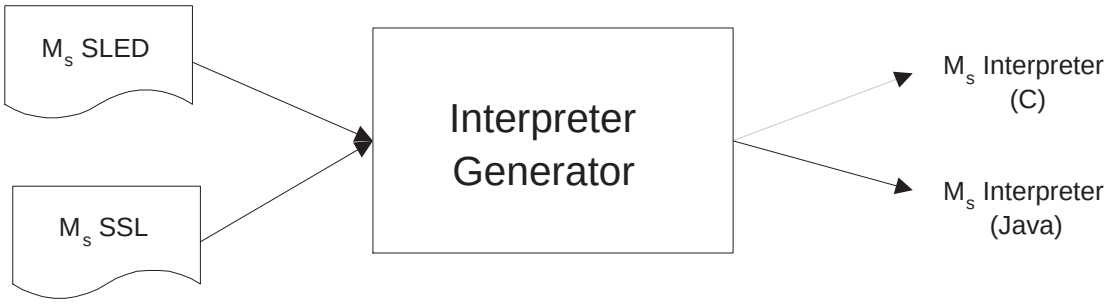

Figure 1.3: The Interpreter Generator Genemu

The interpreter generator, `genemu`, parses SLED and SSL specifications for a machine, knows how to decode ELF binary files, and generates an interpreter for that machine in the C or Java language.

As reported in Chapter 2, the C-based interpreters were tested against the SPEC95 integer benchmarks, whereas the Java-based interpreters were tested against smaller benchmarks as they took so long to run.

More explanation about this subsystem is given in Chapter 2. Note that the documentation makes use of the term “emulator” to refer to the “interpreter”.

1.2.2 The Instrumented Interpreter Generator

We were interested in experimenting with different ways in which we could determine hot paths within an interpreter, hence we designed an instrumentation language, INSTR, which was used in conjunction with the emulator generator in order to generate interpreters that instrumented the code

in the way specified in our INSTR spec. In this way, we could quickly specify different ways of instrumenting code and automatically generate interpreters instrumenting in the scheme of choice. Figure 1.4 illustrates the process.

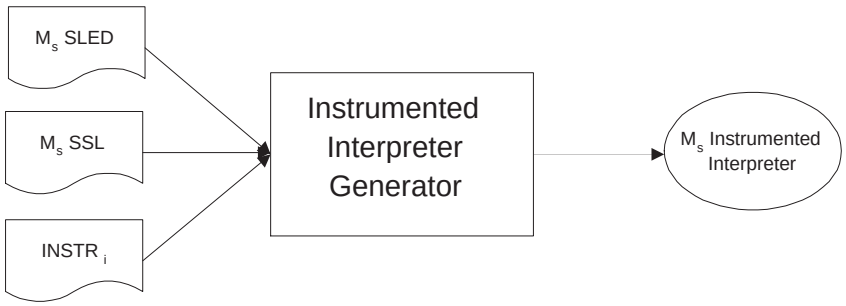

Figure 1.4: The Instrumented Interpreter Generator

The instrumented interpreter generator, `genemu_i`, is an extension of `genemu`, which parses SLED and SSL machine descriptions, as well as the INSTR instrumentation description, and generates an interpreter for that machine which would instrument instructions in the way specified in the INSTR spec. The instrumented interpreter was generated in the C language.

More explanation about this subsystem is given in Chapter 3.

1.2.3 The PathFinder

The 2001 Walkabout implementation is what is referred to as the `pathfinder`. The PathFinder implements Figure 1.5, which interprets SPARC V8 (and a few V9?) instructions, uses one of four different instrumentation schemes to determine hot paths, and generates SPARC V9 code for those hot paths into a fragment cache.

The PathFinder was tested against some SPEC95 and SPEC2000 benchmarks. More explanation about this subsystem is given in Chapter 4.

1.2.4 Debugging Support

One of the goals of the Walkabout project was to provide for 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.

The Walkabout debugger is a Java language GUI 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.

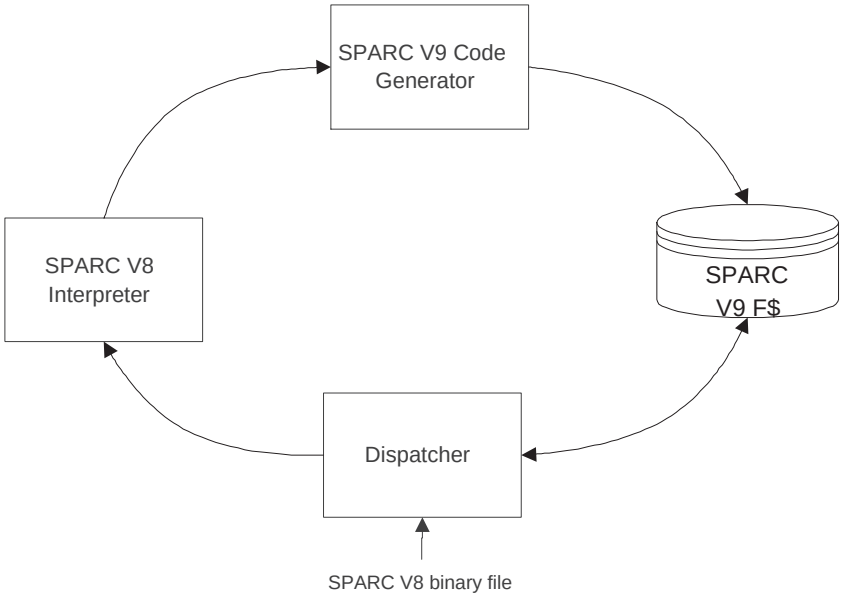

Figure 1.5: PathFinder: The Implementation of the 2001 Walkabout Framework

More information about debugging support is given in Chapter 5.

1.3 Status

The 2001 Walkabout implementation consisted of 16 man-months over a period of 9 months. The project was led by Cristina Cifuentes, and several interns worked on the implementation of different components; namely, Nathan Keynes worked on the emulator generator, Bernard Wong worked on the debugger and disassemblers, and David Ung worked on the hot path instrumentation and code generator. Brian Lewis investigated its debugging and testing support, framework applications, and helped design portions of the emulator generator.

Chapter 2

Emulator Generator

Design: Nathan, Cristina; Implementation: Nathan; Documentation: Nathan [Apr 01]

This chapter documents the inner-workings of the emulator generator for the Walkabout framework. It describes design issues and concentrates on interfaces and usage of the emulator generator tool. This chapter documents the initial version of the emulator generator rather than the release version (i.e. it has not been fully updated) [Cristina, Jan 2002].

2.1 Design

The main design goals of building an emulator for Walkabout, were retargetability, reuse of existing specifications, and efficiency. This has been achieved by making use of an emulator generator, which accepts the SLED and SSL specifications, and outputs source code for an emulator core, in either the C or Java language. With the addition of some simple scaffolding, the result is an easily retargetable emulation system.

Another goal for the emulator generator was to allow users to create emulators without needing to write them in assembler language or even know the assembler of the machine where the emulator is to be run.

2.2 Using the generator

This section explains the way an emulator can be generated, what its interface is and what to know in order to make changes to the code generation module.

2.2.1 Invocation

The emulator generator is called `genemu` and it is invoked in the following way:

```
genemu [options] <SLED spec file> <SSL spec file> [outputfile]
Recognized options:
   -c  output C code [default]
   -d  disassembler only (do not generate emulator core)
   -j  output Java code
```

2.2.2 Interface

The generated code exports one fundamental function, `void executeOneInstruction()`, which completely executes the instruction pointed to by the current PC. The emulator expects a pointer `mem` to have been set up to refer to the start of the emulated program's address space.

The CPU registers are exported as `regs`, which is a `RegisterFile` structure (defined in the generated header file). Note that right now it's not always simple to know a-priori how to access a given register from outside the core, although `%pc` should always be `regs.r_pc`.

If the SSL specification makes use of traps, the calling code must also define a callback, of the form

```
   void doTrap( bool cond, int trapNumber );
```

where `cond` indicates whether the trap should execute, and `trapNumber` gives the trap id.

A minimal use might look something like

```
char *mem;

int main()
{
    mem = loadBinary();
    regs.r_pc = getStartAddress();
    run(); /* Supplied by generated emulator */
}
```

2.2.3 Making the specifications work for you

In order to do what it's supposed to do, the emulator generator relies on certain correlations between the SLED and SSL files that it's using. This section documents those requirements.

The primary requirement is that in general all names must be the same (modulo case). This means instruction names are the same, and register names are the same. Additionally operand names which appear in the SSL file must be defined as fields in the SLED file, but need not necessarily appear in the actual instruction constructor. Also, an operand which is used as a register index (i.e. appears inside an r[] expression), must be defined in the SLED as a field with register names.

Limitations: Assigning to an operand is not currently supported in Java.

2.3 Inside the generator - Maintainer's notes

2.3.1 Roadmap

The directory structure for the emulator generator code is as follows.

include/

codegen.h	Abstract class declarations for CodeGenApp and CodeGenLanguage
codegenemu.h	Main header for emulator generation
codegen_c.h	Header file for C language support
codegen_java.h	Ditto for Java (inherits from C)
sledtree.h	Declaration of the SLED AST classes

tools/

codegen.cc	Generic methods for CodeGenApp and CodeGenLanguage
codegen_c.cc	Methods for C language support
codegen_java.cc	Ditto for Java
gendasm.cc	Methods for generating disassembly functions
gendecode.cc	Methods for generating instruction decoders (via NJMC)
genemu.cc	Main function, and initialization
genregs.cc	Register structure computations
genss.cc	Semantic String / instruction handling
sledscanner.l	Lexical specification for SLED files (lex)
sledparser.y	Syntax specification and ast construction for SLED files (yacc)
sledtree.cc	Methods for SLED AST, and some tree construction support
sledtest.cc	Main function for testing SLED routines
match.cc	Simple code to generate NJMC-style matching statements

emu/

emumain.cc	The emulator's main
sparcmain.java	The emulator's main for Java-based version
personality.h	Base class for the OS Personality
personality.cc	Implementation of the base Personality class
linux.cc	Linux personality implementation
solaris.cc	Solaris personality implementation
sparcemu.h	Emulator interface file for the SPARC architecture
sparcemu.m	Emulator implementation file for the SPARC
sparcemu.cc	Generated file from sparcemu.m
instrsparcstub_c.cc	C language stub methods for the SPARC architecture
sparcstub_c.cc	C language SPARC stub methods
sparcstub_java.cc	Java language SPARC stub methods
x86stub_c.cc	C language x86 stub methods
sysv.cc	SysV loader and process initializer

tools/runtime/

emuskel.c	Emulator implementation file skeleton for C
emuskel.h	Emulator interface file skeleton for C
emuskel.java	Emulator implementation file skeleton for Java

2.3.2 Generator code structure

Conceptually, the generator is separated out into the following pieces:

- Language neutral, abstract application base class (`CodeGenApp`)
- Main emulator implementation (`CodeGenEmu`)
- Application neutral language support classes (`CodeGenC`, `CodeGenJava`)
- Skeleton files (`emuskel.c`, `emuskel.h`, `emuskel.java`, etc)

After parsing it's input, the generator reads in the specification files given to it, and computes the register structures. Then it reads and processes each skeleton file in turn, writing to the given output file and substituting in the actual code as it goes.

All actual code is generated by calls through `lang`, which is an instantiation of `CodeGenLanguage`[1] The skeleton logic is all generic, and so is implemented in `CodeGenApp`.

2.3.3 Skeleton files

The skeleton files are processed by reading them line by line, substituting for any variables found, and writing out again. Variables are implemented similarly to the Unix shell, and can be specified as either $VARNAME or ${VARNAME}. Variable names may only contain alphanumeric or underscore characters.

There is also a simple built-in conditional generation mechanism -

```
@SECTION
 ...
@SECTION
```

will be generated if and only if the SECTION section is active, whereas

```
@!SECTION
 ...
@!SECTION
```

will be generated if and only if the SECTION section is inactive. In either case the conditional directives themselves will not be copied to the output.

2.3.4 Generated code structure

The emulator generator normally creates 2 files - interface (ie .h) and implementation (.c). (In the case of the Java language, it obviously only generates a single output file). The interface file contains some basic typedefs and function prototypes, along with the declaration of the main register structure (`RegisterFile`).

The guts are of course in the implementation file - This is roughly divided into prologue (general macros), disassembler, parameter decoding (mapping register parameters to registers, and breaking up complex operands), instruction routines (one execute routine per instruction), the main execute() function (essentially instruction decode and dispatch), and finally the exported executeOneInstruction() routine, which handles the main fetch-execute cycle (1 cycle's worth).

[1] At least, all code *should* be generated in this way. Currently there are still quite a few places where operators are inserted directly, which would need to be fixed to support less C-like languages.

The disassembler and decoder both depend heavily on the New-Jersey Machine Code toolkit (NJMCTK) to generate the real decoders – the generator itself just produces (long) match statements for these parts, in the form of .m (matching) files which NJMCTK translates into .cc files.

2.4 Stand-alone emulation

In order to test the emulator properly (and transitively the specifications), it's useful to be able to run it in isolation from the rest of Walkabout, ie to completely emulate a binary application. The emulator source generated from this tool is obviously not capable of doing this by itself - it needs support to load binary files, handle operating system calls, etc.

Included in the `emu/` directory is a small set of files to provide the needed infrastructure. Currently it contains support for Solaris and Linux platforms (at least partially, more work is needed for completeness), and stubs for SPARC and x86. In order to support a new CPU core with these platforms, it is only necessary to generate an emulator with the toolkit, and write a small stub file. The task of the latter is to supply routines to set the stack pointer, setup the program counter, and most importantly handle parameter passing to and from system calls. (We currently make the unsupported assumption that there is a standard for this on each CPU architecture - when adding more platforms these stub files will undoubtedly need to handle multiple conventions).

In the case of a Java-based CPU core, the stub file is also responsible for thunking certain calls through to the Java runtime environment.

2.4.1 Personality

The Personality class (and subclasses thereof) is responsible for the loading of binary files, initial stack setup, and syscall handling—in other words for imitating the normal behaviour of the kernel on a real system. Personality in itself only supplies a few utility and factory methods; each subclass is responsible for implementing two key methods:

- bool execve(const char *filename, const char **argv, const char **envp)
- int handleSyscall(int callno, int *parms)

`execve` behaves exactly as the POSIX standard `execve` function, except that the caller retains control, and it does not actually start running the process (which can be done by executing run() as described later). `handleSyscall` is called from the relevant stub file whenever the CPU core encounters the architectural equivalent of a SYSCALL instruction, with the number of the syscall,

and an array of up to 6 parameters. The `handleSyscall` function should return the result of the call in the first parameter, possibly setting the carry flag via `setReg_CF()`.[2]

Note that this is indeed somewhat biased towards Unix and Unix-like systems, however at least currently that includes all systems of interest. It also seems likely that most other systems could be mapped to make use of this interface.

In addition to being a base class for OS-specific behaviour, Personality supplies some basic functions for dealing with the process memory image - in particular all memory accesses from subclasses should be routed through the `putUser*` / `getUser*` functions, as they ensure correct byte ordering.

2.4.2 SysVPersonality

As a convenience, due to the large amount of overlap between modern Unix-like systems, the class SysVPersonality (`sysv.cc`) was introduced to contain the common parts. This is primarily an implementation of `execve`, which is a largely standard (and rather non-trivial) process on systems supporting the ELF file format. Note that the BinaryFile API used in UQBT is not used here, as rather lower-level information is needed by the loader. SysVPersonality also creates one new abstract method

```
int handleAuxv( AUXV_T *auxv )
```

which permits subclasses to add additional items to the process image's auxiliary vector (is passed pointer to first free vector, returns number of items added).

Note that there are some small machine dependencies which have to do with the exact stack layout. Adding a new architecture to the emulator may require adding an entry to the switch here (unfortunately this seems unavoidable).

2.4.3 Stubs

As previously mentioned, a stub file needs to be written for each CPU architecture (and for each language, for that matter). A list of these functions is at the top of `personality.h`, but a quick description may be useful

- `setReg_pc(int)` Set the program counter to the given value, so that execution resumes from that point.

[2] On architectures which expect the parameters to be returned unchanged, the stub file is responsible for making a copy of them.

- `setReg_sp(int)` Set the stack pointer to the given value
- `setReg_CF(int)` Set the carry flag to true/false if the value is non-zero/zero
- `initCore()` Initialize the processor core (normally a no-op for C cores)
- `setMem(char *)` Set the memory base for the emulator core
- `run()` Begin execution - run until told to stop
- `stop(int)` Terminate execute with the given exit value
- `getArchitecture()` Return an ID code corresponding to the CPU architecture being emulated
- `getDefaultPersona()` Return a personality ID representing the "default" platform for a given architecture (ie Solaris for SPARC, Linux for x86)
- `dumpMainRegisters(FILE *)` Dump the main CPU registers to the given stream.

2.5 Performance Analysis

Implementation: Bernard; Documentation: Bernard [May 2001]

This section documents the performance analysis that was done on the emulators generated by the emulator generator, with emphasis on the SPARC emulator. Descriptions of performance analysis tools and results using different tools and techniques are given. Note that these experiments were run in May 2001, prior to completion of the emulator generator's final form.

2.5.1 Performance Analysis

From previous performance evaluation work that had been done by Nathan Keynes, the approximate performance of the emulator is known (see Figure 2.1). From this work, we know that the current emulator is approximately 77 times slower than a natively executed program.

Profiler Breakdown

To get a better understanding of where the extra time is spent, profiling of the emulator need to be done. First an appropriate profiler need to be chosen for the task.

Emulated Program	Slow down from native
099.go	66.85x
124.m88ksim	96.51x
129.compress95	77.32x
130.li	70.1x
132.ijpeg	112.49x
134.perl	63.67x
147.vortex	68.82x
Mean	77.79x

Figure 2.1: Previous performance evaluation of the C++ version of the emulator taken from Nathan Keynes' presentation slides

Gprof	Quantify	Shade
Pros • Easy to setup and use • Gives information of time spent in each function • Shows function trace of the application *Cons* • Groups time spent in each function with time spent in function's children • Text based function trace very hard to follow	*Pros* • Gives a nice graphical call graph • Very detail breakdown of function usage *Cons* • Proprietary tool where the data gathered can only easily be viewed within the program	*Pros* • Poweful tool for creating custom profilers • Can be used to analyse specific instructions or instruction sets *Cons* • Very large amount of work required to create custom profiler

Figure 2.2: Comparison and evaluation of different profilers for use with the C++ based emulator

Three different profilers were evaluated - `gprof`, `quantify` and `Shade`. A brief summary of each tool can be found in Figure 2.2.

From the limited profiling requirements of the project, a tool such as `Shade` is far more complex and time consuming than necessary. Most of the important information from profiling are given by much easier to use tools such as `gprof` and `quantify`.

Although `gprof` gives enough information to meet most of the profiling needs for the emulator, `quantify` can give the same information as `gprof` but in greater detail and in a graphical environment. The proprietary nature of the `quantify` tool is not a large concern as the data set gathered does not need to be further analysed by other tools. Therefore, `quantify` was used to perform the remainder of the profiling for the C++ version of the emulator.

VM Overhead

The first data set that needs to be gathered is relationship between the time spent in the actual emulation of the code and the overhead in creating the environment necessary for the emulation. Figure 2.3 is a profile of the functions called from `main` in the emulator and the amount of cycles spent in each of these functions and their descendents. The program that the emulator is running is the `sieve` program, generating the first 3000 primes. The `sieve` program is compiled with an optimization of O4 with gcc 2.81.

Functions Called from Main	Cycles (w/descendants)
executeOneInstruction	851,395,870,815
BinaryFile::Load	21,582,646
runDynamicLinker	68,794
initVM	22,423
atexit	398
Total Cycles	*851,417,545,076*

Figure 2.3: Breakdown of cycles spent in functions called from main of the emulator - `sieve`

From this data, it can be seen that more than 99% of the time is spent executing the actual emulation code. However, sieve is a relative small program with a size of only 24,452 bytes and requires a relatively large amount of CPU cycles. A larger program that requires a smaller amount of CPU cycles will not fare as well.

Figure 2.4 shows the cycle breakdown of the `banner` program displaying the word “yo”, an example of a program that requires much fewer CPU cycles. The size of the `banner` executable is 6,084 bytes compared to the size of the `sieve` executable which is 24,548 bytes.

With the analysis of the `banner` program, we see that only 73% of the time is spent executing the actual emulation code. However, much of this time is spent in loading the executable which is unavoidable as even a natively executing program must spend time loading itself into memory. However, the efficiency of the loader compared to the native OS loader is currently unknown and requires further analysis.

Functions Called from Main	Cycles (w/descendants)
executeOneInstruction	6,515,532
BinaryFile::Load	2,350,321
runDynamicLinker	70,200
initVM	22,821
atexit	398
Total Cycles	*8,959,272*

Figure 2.4: Breakdown of cycles spent in functions called from main of the emulator - `banner`

A reasonable conclusion can be made that, although the efficiency of the emulator's binary loader is not known, it cannot account for much of the overall performance slowdown of the emulator.

Child Functions Breakdown

In order to further analyse the performance of the emulator, a list of the most time consuming functions were generated to see where the emulator is spending most of its time. Figure 2.5 is a list of the ten most time consuming functions as seen from running the sieve3000 program.

Functions	% Time	# of times called
execute	40.16	6,010,495,867
executeOneInstruction	16.86	6,010,495,867
executeSUBCC	10.57	737,516,466
decodereg_or_imm	8.62	3,139,748,631
executeADDCC	5.09	342,876,189
decodeeaddr	2.86	660,334,656
executeRESTORE	2.39	94,237,151
executeSAVE	2.32	94,237,151
executeORCC	2.10	383,153,136
executeOR	1.12	565,495,820
executeBL	0.94	401,856,672

Figure 2.5: Ten most time consuming functions of the emulator when running the sieve3000 program

A few noticeable patterns can be seen from the function breakdown in Figure 2.5.

1. Most of the time is spent in the `execute` function. This function is used to match the instruction bit patterns to the assembly equivalent for the source machine. For every instruction that needs to be executed, one iteration of the execute function is required.

   Since the matching of the bit patterns requires significant amount of branches and also because of the frequency of this function call, it is not surprising that this function takes a significant amount of the processing time.

2. The function `executeOneInstruction` also required a significant amount of time. Again, this is because this function is called once for every instruction that must be executed.

3. The amount of function calls of the top ten functions alone is staggering as the same functions are called over and over again. Since each function itself requires very little time to execute, all of these functions are good potential targets for inlining.

4. Functions that require the manipulation of conditions codes such as `executeSUBCC`, `executeADDCC`, and `executeORCC` require a significant amount of execution time even though the frequency of these instructions are relatively low for the SPARC architecture. This indicates that each of these instructions require a significant more time to execute than their counterparts that do not require condition codes.

5. The two decoding functions `decodereg_or_imm` and `decodeaddr` both are called a significant amount of time and take up more than 10% of the total execution time. These decode the compound matchining statments `reg_or_imm` and `eaddr` from the SPARC spec.

6. The two functions `executeRESTORE` and `executeSAVE` are called very infrequently as they are only needed on function calls and returns in the original source program. However, they both take up a significant amount of time which indicates that they both are very time consuming functions and perhaps an area that can be optimized.

2.5.2 Performance Experiments

Inlining Functions

In order to reduce the amount of function calls, all the `execute*` and `decode*` functions were inlined. This will in effect cause the bulk of the emulator to be compiled and executed and one large function. The benefits of this is the elimination of saving and restoring registers and also allows the compiler to perform greater amount of optimizations.

As can be seen from Figure 2.6, the effects of inlining is a noticeable increase in performance in the range of 10%.

Program Executed	Original Emulator	Modified Emulator	% Time Saved
SPEC95 130.li	3h 22m 49s	2h 47m 52s	17%
sieve-3000	19m 29s	16m 45s	14%
fibonacci-32	14.4s	13.2s	8%

Figure 2.6: Performance improvements due to inlining of the execute and decode* functions. Test performed on a 4 CPU Sun Ultra-80, with low load

Bitwise Operation Simplification

To address the issue of time consuming condition code manipulating functions, much of the condition code addressing areas have been analysed and redundancies have been removed. For example, many of the condition code manipulation required accessing a particular bit in a variable. The majority of this bit access is to the same bit. However, in order to access this bit, a very cumbersome but generic operation is used.

```
#define BITSLICE(x,lo,hi) (((x) & ((1LL<<(hi+1))-1))>>lo)
```

where hi and lo are both 31. This was replaced with the following

```
#define BITPICK(x,lo,hi) (((uint32)x) >> 31)
```

However, later analyse of the assembly code shows that the compiler when set at a reasonable level of optimization will already perform this type of conversion.

Register Window Modifications

The `executeSAVE` and `executeRESTORE` operations contributed a significant amount of execution time yet were called only a small amount of times. This can be due to the register window implementation of the emulator where there only exist one window. Therefore, every save and restore operation will require spilling out and reading from the stack, which results in a significant amount of memory operations.

A true sliding register window would require far less memory operations. However, due to the SSL and SLED based nature of the emulator, a true sliding register window implementation can not be easily accomplished. Currently, the global registers are stored in the same array as the window registers. Therefore, attempts to allocate a multiple window array and simply slide the offsets of the array would cause references to the global registers to be incorrect.

Logic can be added to every register operation to determine whether the register in question is a global register. However, the performance hit of this was considered to be too significant and further experimentation in an overlapping sliding register windows was not investigated.

A non-overlapping sliding register windows implementation was implemented as the work required is significantly less. In this implementation, the `in` and `out` registers are not overlapped. The window actually slides by 32 registers and the values of the global and the in/out registers are copied to the new position. Therefore, all registers are in the correct position and no additional logic is required to determine whether a register is global.

However, in performance evaluation, the performance had actually decreased by roughly *10%*. The reason is due to the signal handlers, which save and restore the CPU context at every signal. When this occurs, the whole register window is saved, and since the register window is now signficantly bigger, the context save and restore time is significantly slower and therefore there exists a performance decrease with this implementation.

Java Emulator Analysis

The current performance of the Java-based emulator is approximately 15 times slower than the C++ version of the emulator[3]. In order to investigate the reason for the additional performance slow down, profiling of the Java emulator needs to be done.

The Java version of the emulator was original written with a C++ front end that loads a `JVM` and then calls the Java code. In order to profile the Java code, the emulator must be modified to start as a Java program.

Re-implementing the emulator only required approximately one working day and allowed the use of a Java profiler on the code. It is also a cleaner implementation as it only uses `JNI` to use C++ binary loading libraries instead of relying on C++ code much more as in the original version.

Performance comparison shows that this new version performed comparably to the original version.

The profiler built into the JDK 1.4 was used to analyse the emulator. However, the profiler was not initially working correctly as no data was produced by the profiler.

After investigation, it was found that the reason behind the profiling problems was due to an incorrect interpretation of the system call `exit`. Normally, the emulator will trap the exit system call and then will perform an exit. Unfortunately, this action will also kill the profiler process before it has a chance to output the data it gathered.

To remedy this, the emulator will perform a Java `System.exit` call instead of the normal exit system call. This will gracefully shut down the profiler and allow the profiler to operate correctly.

[3] From previous performance evaluation by Nathan Keynes

```
#ifdef JAVA
    cls = env->FindClass("java/lang/System");
    func = env->GetStaticMethodID( cls, "exit", "(I)V");
    env->CallStaticVoidMethod(cls, func, o0);
    break;
#else
    exit( o0 ); err = 0; break;
```

The above is the code segment inside the system call handler that was changed in order to exit the Java version of the emulator correctly.

Performance of JNI vs. Unsafe

A `JNI` version of the emulator was developed that did not require the use of `Unsafe` classes. The `Unsafe` classes, introduced in JDK 1.4, were used to allow the Java emulator to directly access memory addresses and also make type cast that Java would normally not allow. Creating a JNI version of the emulator allowed comparison in performance between the `Unsafe` classes to the JNI equivalent.

Program Executed	Unsafe-based	JNI-based	% Extra Time JNI requires
sieve-3000	288m 12s	333m 4s	16%
fibonacci-32	3m 42s	4m 28s	21%

Figure 2.7: Performance difference of JNI and Unsafe based versions of emulator. Test performed on a 4 CPU Sun Ultra-80, with low load

This data clearly show the overhead introduced by using JNI over using Unsafe. The advantage of the JNI version is that it does not require the use of JDK 1.4, which is still in beta testing.

Profile of Java Emulator

Using the Java profiler, the following data was gathered from executing the `sieve` program through the Java version of the emulator. The data gathered was very similar to the ones gathered from the C++ version of the emulator. The methods `execute` and `executeOneInstruction` still dominate the total time spent by the emulator.

```
rank   self  accum   count trace method
   1 27.64% 27.64% 1432686509 133 sparcemu.execute
```

```
   2 20.15% 47.79% 1432686509  31 sparcemu.executeOneInstruction
   3  9.46% 57.25% 1432686510  98 sparcemu.getMemint
   4  6.90% 64.15% 3421444300  36 sparcemu.decodereg_or_imm
   5  5.60% 69.75% 3421444300  52 sparcemu.getMemint
   6  5.35% 75.11% 1432686510  46 sun.misc.Unsafe.getInt
   7  2.98% 78.09% 3421444300 110 sun.misc.Unsafe.getInt
   8  2.50% 80.59% 1507734608  56 sparcemu.setMemint
   9  2.41% 83.00% 1507734608 149 sparcemu.getMemint
  10  1.80% 84.80% 737504371   34 sparcemu.executeSUBCC
  11  1.80% 86.61% 848237913  153 sparcemu.executeOR
  12  1.60% 88.21% 1507734608  26 sun.misc.Unsafe.putInt
  13  1.43% 89.63% 94233413    65 sparcemu.executeRESTORE
  14  1.39% 91.02% 94233413   138 sparcemu.executeSAVE
  15  1.30% 92.33% 1507734608  42 sun.misc.Unsafe.getInt
  16  0.93% 93.25% 401879787  156 sparcemu.executeSRL
  17  0.85% 94.10% 376998205   20 sparcemu.executeSETHI
  18  0.83% 94.93% 383153738  127 sparcemu.executeORCC
  19  0.55% 95.48% 401864307   27 sparcemu.executeBL
  20  0.46% 95.95% 342881000   78 sparcemu.executeADDCC
  21  0.45% 96.40% 200981367  145 sparcemu.executeSLL
  22  0.39% 96.79% 189104847   49 sparcemu.decodeeaddr
  23  0.38% 97.18% 188499799   23 sparcemu.executeJMPL
  24  0.37% 97.55% 282667911  121 sparcemu.executeBCS
  25  0.32% 97.87% 189104847   96 sparcemu.getMemint
  26  0.29% 98.16% 188499257   43 sparcemu.executeCALL
  27  0.28% 98.44% 223767090    7 sparcemu.executeADD
  28  0.26% 98.70% 188445153  151 sparcemu.executeBLA
  29  0.24% 98.95% 194702191   61 sparcemu.executeBA
  30  0.22% 99.17% 94282993    66 sparcemu.executeSUB
  31  0.17% 99.34% 189104847  118 sun.misc.Unsafe.getInt
  32  0.14% 99.48% 100492772   79 sparcemu.executeBLEU
  33  0.13% 99.61% 94242481   137 sparcemu.executeBNE
  34  0.13% 99.75% 100487320  120 sparcemu.executeBGE
  35  0.13% 99.87% 94280252    41 sparcemu.executeBEA
  36  0.12% 100.00% 94321671   69 sparcemu.executeBE
CPU TIME (ms) END
```

An interesting discrepancy between the Java and C++ versions of the emulator is the amount of times the `execute` and `executeOneInstruction` functions are called. However, that is probably

due to the difference in profilers only and is not alone enough to discount the accuracy of the data gathered.

From the profiling results, it can be seen that the Java version of the emulator has the additional overhead of using functions such as `getMemint`, `Unsafe.getInt`, and `Unsafe.putInt`. These functions in total account for more than *31%* of the total time spent.

Although 31% extra overhead is significant, it does not account for the approximate *15* times slowdown of the Java emulator when compared to the C++ emulator.

Dynamic Behaviour of Java VM

An area that can perhaps account for the *15* times slowdown is the dynamic behaviour of the Java virtual machine. If the Java VM (`Hotspot 1.4`) was able to discover all the hot paths and make them into compiled code, then the performance of the Java version should be very close to a natively compiled C++ version. Further more, the Java VM could even be smart enough to recognize the nature of the code and realize that most of the functions are good candidates for inlining further improving performance. The following is a breakdown of the dynamic behaviour of the Java VM.

```
Flat profile of 17670.26 secs (879033 total ticks): main

  Interpreted + native   Method
 75.7% 663823  +  1335   sparcemu.execute
  0.0%     0  +     7    emumain.main
  0.0%     0  +     2    sparcemu.doTrap
  0.0%     0  +     1    sparcemu.decodereg_or_imm
  0.0%     0  +     1    java.util.zip.ZipFile.getEntry
  0.0%     0  +     1    java.lang.String.charAt
  0.0%     1  +     0    sparcemu.executeBL
  0.0%     1  +     0    sparcemu.executeSLL
  0.0%     1  +     0    sparcemu.executeADDCC
  0.0%     0  +     1    java.util.zip.ZipFile.open
  0.0%     1  +     0    sparcemu.setMemdouble
  0.0%     1  +     0    sparcemu.executeBGU
  0.0%     0  +     1    java.lang.Shutdown.halt
  0.0%     1  +     0    sparcemu.executeSAVE
  0.0%     0  +     1    java.io.FilePermission.newPermissionCollection
  0.0%     1  +     0    sparcemu.executeRESTORE
  0.0%     1  +     0    sparcemu.getMemdouble
 75.7% 663831  +  1350   Total interpreted
```

```
    Compiled + native   Method
 4.6% 40134  +     0    sparcemu.getMemint
 3.1% 27390  +     0    sparcemu.executeOneInstruction
 2.8% 24458  +     0    sparcemu.decodereg_or_imm
 1.6% 14220  +     0    sparcemu.executeSUBCC
 1.5% 12769  +     0    sparcemu.run
 0.8%  7368  +     0    sparcemu.executeRESTORE
 0.8%  6753  +     0    sparcemu.executeADDCC
 0.7%  6113  +     0    sparcemu.executeSAVE
 0.6%  5025  +     0    sparcemu.setMemint
 0.6%  5001  +     0    sparcemu.executeOR
 0.5%  4444  +     0    sparcemu.executeORCC
 0.3%  2470  +     0    sparcemu.executeSETHI
 0.3%  2352  +     0    sparcemu.executeSRL
 0.3%  2250  +     0    sparcemu.executeBL
 0.2%  2007  +     0    sparcemu.decodeeaddr
 0.2%  1806  +     0    sparcemu.executeBCS
 0.2%  1708  +     0    sparcemu.executeBLA
 0.2%  1460  +     0    sparcemu.executeADD
 0.1%  1198  +     0    sparcemu.executeCALL
 0.1%  1146  +     0    sparcemu.executeJMPL
 0.1%  1084  +     0    sparcemu.executeSLL
 0.1%  1030  +     0    sparcemu.executeBA
 0.1%   908  +     0    sparcemu.executeBNE
 0.1%   891  +     0    sparcemu.executeBEA
 0.1%   693  +     0    sparcemu.executeSUB
20.1% 176577  +     3   Total compiled (including elided)

        Stub + native   Method
 0.0%     0  +    12    sparcemu.doTrap
 0.0%     0  +    12    Total stub

 Runtime stub + native  Method
 2.0% 17440  +     0    interpreter_entries
 2.0% 17440  +     0    Total runtime stubs

 Thread-local ticks:
 0.0%     1             Blocked (of total)
 0.0%     2             Class loader
```

```
  2.2% 19706             Interpreter
  0.0%     9             Compilation
  0.0%    68             Unknown: running frame
  0.0%     1             Unknown: calling frame
  0.0%     1             Unknown: no last frame
  0.0%    32             Unknown: thread_state


Global summary of 17670.26 seconds:
100.0% 879033            Received ticks
  0.0%     8             Compilation
  0.0%     2             Class loader
  2.2% 19706             Interpreter
  0.0%   102             Unknown code
```

The above data shows that the majority of the time was spent interpreting code instead of executing compiled code. The function `sparcemu.execute`, which when interpreted, accounts for 75.7% of the execution time. However, it was not found to be hot by the VM and thus was not compiled. This interpretation probably accounts for a significant portion of the slowdown of the Java version of the emulator compared to the C++ version. Also, many of the functions that the VM decided to compile such as `sparcemu.executeSLL` and `sparcemu.executeBEA` are really not executed that frequently and the compilation overhead for these functions may not be justified. Further investigation with fully compiled Java code would be useful to prove or disprove these theories.

Chapter 3

Instrumentation of an Interpreter via Specifications

Design: Cristina, David; Implementation: David; Documentation: David Ung [May 2001], Cristina [Jan 2002]

This chapter describes the specification file format used to automatically add instrumentation code to the Walkabout based emulator (described in Chapter 2). Example instrumentation files are also given to demonstrate the type of instrumentation that it can create and how it is integrated into the emulator.

3.1 Instrumentation

The goal of this research is to determine an inexpensive and easy to use way to add instrumentation to the emulator. The type of instrumentation to be added describes ways to identify what sections of the source program are hot (i.e. frequently executed), so that code generation can be done to improve the overall performance of the execution.

Existing tools such as EEL [17] and ATOM [10] provide an interface to add code by specifying the level where instrumentation should take place and what to instrument. Such tools reconstructs the application to be instrumented to an intermediate representation in memory and then modifies its structure (through CFGs and basic blocks) to add the instrumentation into the application. Finally, the binary is rebuilt and an instrumented version is emitted.

There are two fundamental problems with the approach described above when trying to add instrumentation to the Walkabout emulator:

1. When the emulator is running, it is emulating the behaviour of a program. The execution of the program determines which paths to take during runtime and hence indirectly affects which part of the emulator is invoked. Adding instrumentation to the emulator will instrument the emulator. Although this can indirectly give information about the runtime behaviour of the source program, the information gathered will be more in the scope of the emulator, thus losing emphasis on the source program. In particular to the attempt to find hot traces, instrumentating the emulator may tell us that the function `executeBNE()` is hot, but the information about which instruction cause this in the source program is not revealed. It is possible to obtain information about the source program through the instrumenting the emulator, but the task is not an easy one. It requires knowledge about the internal workings of the emulator and makes the works of instrumentation difficult to use.

2. The emulator is a very low level data processor. Instead of multiple level of abstractions found in existing tools, the only abstraction of the emulator is at the instruction level. This low level abstraction greatly limits the amount of calls that can be made to predefined functions provided by high level instrumentation tools. For example, high level functions such as FOREACH_EDGE() and FOREACH_BB().

To provide a flexible and powerful instrumentation at the instruction level, the emulator itself should provide the instrumentation. Since the emulator is automatically generated, this motivates the idea of automatically adding instrumentation code as part of the emulator. Being part of the emulator code, the instrumentation has access to variable and locations internal to the emulator. This approach allows direct control over what to instrument. The goal is to instrument the source program, not the emulator.

3.1.1 Existing instrumentation tools

The use of instrumentation provides opportunities in binary editing, emulation, observation, program comprehension and optimization. Although many tools exist that can modify binaries, the implementation of the actual code modification are typically fused in detail with the application or executable itself. But several tools exist that provide a high level interface (typically through a set of libraries) to easily access its instrumentation facilities. The following are some examples of such tools:

1. Srivastava and Wall's OM system [7], a library for binary modification. It requires relocation from object files to analyse control structure and to relocate edited code.

2. ATOM [10] provide an interface to the OM system. Very high level of abstraction, simplifies the writing of tools.

3. QPT [16] by Larus and Ball, a profiling and tracing tool.

4. EEL [17], also a library for building tools to analyze and modify binaries.

Both EEL and ATOM are large libraries that provide a rich set of routines for instrumentation. Different levels of the abstraction in these tools allows control over what level the tool wants to instrument through calls to those library rountines. The following example EEL code shows the use of the library:

```
executable* exec = new executable("test_program");
exec->read_contents();

rountine* r;
FOREACH_ROUTINE (r, exec->rountines()) {
  cfg* g = r->get_control_graph();
  bb* b;
  FOREACH_BB(b, g->blocks()) {
    edge* e;
    FOREACH_edge(e, b->succ()) {
      count_branch(e);
    }
  }
}
```

The types `executable`, `rountine`, `cfg`, `bb` and `edge` are data structures provided by the library for different level of abstraction of the binary. The tools simply make calls to library routines such as `read_contents()`, `FOREACH_BB` and `get_control_graph()`. The only function that needs to be written by the tool builder is `count_branch()`. The concept in ATOM is similar to EEL in that the tool builder make uses of library rountine to access the different levels of abstraction in the binary.

3.2 Instrumentation specification

The instrumentation code is written in a separate specification file that is linked as part of the emulator at the time of generating the New Jersey Machine Code (NJMC) matching file. Code can be added at the instruction level under the `DEFINITION` section. The list of instructions that you wish to act on is specified as a table. Instrumentation code is then specified for a table by adding relevant code with respect to the main body of their emulation rountines. For example, if you want

to count the number of times a particular set of branches is taken in an X86 program, the instructions to be monitored are specified in a table 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" ]
```

In order to count occurrences of the branch instructions listed in the `jump32s` table, the semantics of the branch instructions is extended to increment a counter. This is expressed in the INSTR language as follows:

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

where the function `increment_counter` is defined in the `IMPLEMENTATION_ROUTINES` section of the specification file, and `SSL_INST_SEMANTICS` refers to the semantics of the instruction as specified in the semantic description file SSL. For illustration purposes, we show the section `IMPLEMENTATION_ROUTINES`, where the function `increment_counter` is implemented.

```
IMPLEMENTATION_ROUTINES

#include <map>
#include <iostream>

// map edge to execution counts
map< pair<unsigned, unsigned>, int > edge_cnt;

// increments the branch count for edge (addr1, addr2)
void increment_counter(int addr1, int addr2) {
    // construct the edge.
    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())
    {
        // not found in map, add it and set count to 1
        backedge_cnt[edge] = 1;
    } else
    {
        (*i).second ++;      // increment counter by 1
    }
}
```

To build the instrumented SPARC emulator, the SLED, SSL and the instrumentation file (INSTR) are included as part of the build to generate the matching `.m` file:

```
tools/genemu -i sparc.backbranches.inst machine/sparc/sparc-
core.spec
    machine/sparc/sparc.ssl sparcemu.m
```

The `-i` option includes the instrumentation file `sparc.backbranches.inst` into the build of the SPARC emulator. The contents of `sparc.backbranches.cnt500.inst` is found in the Appendix to this Chapter.

3.2.1 Syntax of the INSTR Language

The instrumentation file consists of two 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 loop. The support code section contains additional code that the program may call as part of instrumentation. This code is expressed in the C language. The EBNF for the language is:

```
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)* SSL_INSTR_SEM }

support_code:       IMPLEMENTATION_ROUTINES c_code
```

where `action` contains any valid C/C++ code written by the user. This piece of code is attached to the instructions specified in the table along with the instruction's semantics. The following special symbols may be inserted into `action`:

1. `SSL_INST_SEMANTICS`: stands for the semantics of the instruction, as described in the SSL specification file,
2. `PARAM(string)`: indicates the value of the operand `string` of the current instruction. E.g. `PARAM(label)` of the `BA` instruction is the instruction's first operand.
3. `SSL(%register name)`: indicates one of the machine registers specified in the SSL file. E.g. `SSL(%pc)` is the location holding the value of the emulated PC register.

3.3 Appendix

3.3.1 Listing of an Instrumentation File

```
# File: sparc.backbranches.cnt50.inst
# Desc: This file contains the list of instructions and actions
#       used for instrumentation.
#       The file instruments backbranches and invokes the trace
```

```
#       builder to build hot traces used for optimisation.

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" ]
branch label
{
    int oldpc = SSL(%pc)
    SSL_INST_SEMANTICS

    // check back branches
    if (oldpc > PARAM(label)) {
        if (trace_mode) {
            end_build_trace();
} else {
            // branch is taken
            if (SSL(%npc) == PARAM(label)) {
                increment_counter(oldpc, PARAM(label));
            }
        }
    }
}

FETCHEXECUTE
{
    if (trace_mode) {
        add_to_trace(SSL(%pc));
    }
    SSL_INST_SEMANTICS
}

INSTRUMENTATION_ROUTINES
#include "emu/backbranches.cnt500.cc"
```

3.3.2 Listing of Support Code: backbranches.cnt500.cc

```
#include <map>
#include <list>
#include <iostream>
```

```
// an edge is made up by a pair of addresses (branch_inst, target_inst)

// the upper value that an edge must reach to trigger the build-
ing of traces.
// this is value is incremented by 500 each time start_trace is called.
int trigger = 500;

// map edge to execution counts
map< pair<unsigned, unsigned>, int > backedge_cnt;

// informs the emulator whethere it is in trace mode.  If so,
// it will add the current instruction at %pc to the trace.
int trace_mode = 0;

// holds the list of instructions in the current trace
list<unsigned> trace;


// ************************************************************************
// Function definitions
// ************************************************************************

// prints the trace to stderr and exit
void print_trace() {
    cerr << "Trace found: " << endl;
    list<unsigned>::iterator i;
    for (i = trace.begin(); i != trace.end(); i++) {
        cerr << hex << (*i) << endl;
    }
    abort();
}

// signal end of trace and invoke optimizer
// at the moment it just prints the trace to the screen
void end_build_trace() {
    trace_mode = 0;
    print_trace();
}

// add instruction curr_inst to the trace list
// if curr_inst is the back branch that started trace building,
// then trace building
void add_to_trace(unsigned curr_inst) {
    trace.push_back(curr_inst);
    // check to see if it is the end of trace
```

```
    if (trace.front() == curr_inst) {
        end_build_trace();
    }
}

// start building a hot trace from back branch edge
void start_trace(pair<unsigned, unsigned> edge) {
    cerr << "Starting trace finding!" << endl;
    trace_mode = 1;
    // increase the trigger count or interate through the entire
    // map and reset all counters or just empty the map
    trigger += 500;
    trace.clear();
    // add edge to the current trace
    trace.push_back(edge.first);
}

// increments the branch count for edge (addr1, addr2)
// if count reaches trigger, call start_trace to begin trace building.
void increment_counter(int addr1, int addr2) {
    // construct the edge.
    pair<unsigned, unsigned> edge =
                        pair<unsigned, unsigned>(addr1, addr2);
    map< pair<unsigned, unsigned>, int >::iterator i;
    if ((i = backedge_cnt.find(edge)) == backedge_cnt.end()) {
        // not found in map, add it and set count to 1
        backedge_cnt[edge] = 1;
    } else {
        (*i).second ++;              // increment counter by 1
        if ((*i).second >= trigger) {
            start_trace(edge);
        }
    }
}
```

3.3.3 Sample Output

Given the specification file in this Appendix, the newly generated `.m` file will now contain the extra instrumentation code as part of the emulation functions. For example, the code for the branch always, BA, and the branch not equal, BNE instructions is:

```
void executeBA( sint32_t reloc )
{
```

```
  int oldpc = regs.r_pc;

  regs.r_npc = (((1) == (0)) ? regs.r_npc : reloc);

  // check back branches
  if (oldpc > reloc) {
    if (trace_mode) {
        end_build_trace();
    } else {
        if (regs.r_npc == reloc) {
            increment_counter(oldpc, reloc);
        }
    }
  }
}

void executeBNE( sint32_t reloc )
{
  int oldpc = regs.r_pc;

  regs.r_npc = (((regs.r_ZF) != (0)) ? regs.r_npc : reloc);

  // check back branches
  if (oldpc > reloc) {
    if (trace_mode) {
        end_build_trace();
    } else {
        if (regs.r_npc == reloc) {
            increment_counter(oldpc, reloc);
        }
    }
  }
}

...

void executeOneInstruction()
{
    if (trace_mode) {
        add_to_trace(regs.r_pc);
    }

    sint32_t tmp;
    tmp = regs.r_pc;
    regs.r_pc = regs.r_npc;
```

```
    regs.r_npc = ((regs.r_npc) + (4));
    regs.rd[0] = 0;
    execute(tmp);
}
```

Chapter 4

PathFinder

Design: David, Cristina; Implementation: David; Documentation: David Ung [Aug 2001], Cristina [Jan 2001]

This file describes the internals of PathFinder; a SPARC to ULTRASPARC code execution system.

4.1 Architecture Overview

The PathFinder's architecture is illustrated in Figure 4.1. The SPARC interpreter is that automatically generated by `genemu_i` when using an instrumentation file that determines hot paths based on a given set of termination conditions. We instrumented branches to count how many times a given branch was executed; when the counter reached a trigger, a trace of the hot path could be generated.

One of the instrumentation modes replicated as close as possible Dynamo's [1] next executing tail (NET) method to determine hot paths, as a way to evaluate it in comparison to other methods. Hence, the trace was generated during the next execution of the code and the trace selection termination conditions were:

- A back branch is met, or
- The trace buffer reaches its limit size

The code generator is then called to generate code based on the trace that was collected. During tracing, various data is also collected. (memory references of the indirect jumps, target of loads and stores, return addresses, etc) which are stored in a data structure called `reference_map`, that will

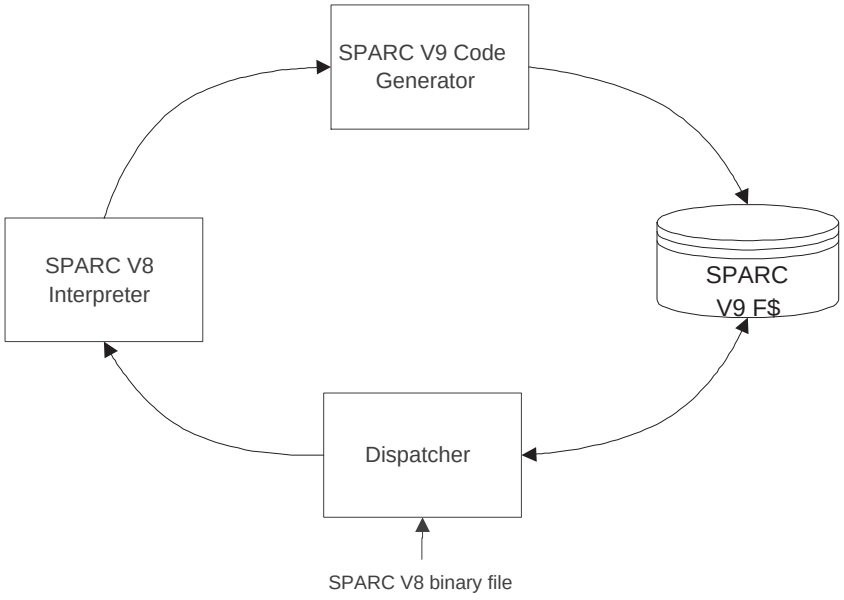

Figure 4.1: PathFinder: The Implementation of the 2001 Walkabout Framework

be used by the code generator. The code generator uses the NJMC to encode assembly instructions into binaries. It builds labels for every source address that is in the trace, these label will be used by the fragment to determine the relative offset of instructions such as branches. The trace instructions are then iterated through a second time though this time with each instruction, it calls `encode()` to write the instruction to the relocatable block (a piece of memory set aside to do encoding, which eventually gets copied into the fragment cache). Code layout, inversion and other optimizations are done as part of the encoding. At the end of trace encoding, all portals[1] that were generated are encoded at the end after the code layout instruction starting with the out portal of the back branch itself, followed by a series of other portals. The relocatable block is then copied into the fragment cache and an entry into the fragment is added to the map of entry points.

Execution does not immediately enter the newly generated fragment. The semantics of the back branch is executed before the code generator is called, hence the delay slot instruction is the next instruction to be executed. Control is passed back to the emulator, and another interation is done under the emulator before the fragment is called. When the same back branch is reached, Pathfinder does a context switch to get ready to enter the fragments. A context switch involves the moving of

[1]Portals are edges going into or out of a hot path region. For an edge (a,b), an in-portal is the node ‘a’ such that node ‘b’ belongs to the hot path. Similarly, an out-portal is a node ‘a’ that is part of the hot path. For examples refer to [25].

values from the virtual registers to the real host machine registers[2] Other registers including integer and floating point condition code are written to the correct fields in the host state registers. Once control is in the fragment, no emulation or instrumentation takes place. PathFinder gets control only when one of the exits of the fragment is taken. The exit (out portal) will do a save to preserve integrity of the fragment state. The contents of real registers and condition codes are written back to the virtual locations. To issue control to the emulator, one could jump directly to it. Instead, the Pathfinder sets up a virtual PC and a virtual nPC, and sets CTI to 1. A return from the routine will end up in the main loop of the emulator. Fragment linking is done as part of jumping from fragment to emulator. Some details of the implememtation is described in Section 4.4.

4.2 Relocator

The relocator was the first attempt to use an instrumentation profile to drive the emulator to generate traces. Basically, the relocator instruments back branches and generates traces using the next executing tail scheme. The following are design decisions with the relocator:

1. The address reserved by the emulator to hold the context of the source program remains unchanged. This memory block is the member VM of one of the systems personalities. The variable `mem` is the base address of the VM. Each memory reference is adjusted by `mem` as part of the macros in the `emu.m` file. The relocator preserve this behaviour, hence the fragment code generated need to adjust every load and store instruction to correct the accessing of the source program's data and text area.

2. Delay slot relocation. If the delay slot of a control transfer instruction is a load or a store, relocation can be difficult. Since the relocation will involve a series of adjustment instructions, it will result in code increase that is too big to fit in a delay slot. Some careful rearrangement of instructions is required for this case.

3. Mirrored stack. A mirrored stack is essential for the correct execution of the relocator. At the time when the program is in the fragment, the host machine's stack pointers are pointing to a virtual address space within the VM (set aside for source program mapping, see above). However, physically, the VM is mapped on top of the emulator's context, hence the stack pointers are physically pointing inside the emulator, not the source programs space. This is okay, since every memory reference (including stack references) are relocated to the VM. The only problems with this approach is when a flush spill trap happens. Since it is not under the control of the emulator or the fragment (both which are relocated memory accesses), the

[2]The current implementation uses some global registers as scratch registers, though some space has been set aside in `host_context.temp[16]` to hold 16 temporaries during context switching. One could change the code so that those global registers are not overwritten.

trap will write the area physically pointed to by %FP and %SP. A mirrored stack is created to combat this by mmapping a block of memory so the spill handler can use as temporary storage.

4.3 Building

The PathFinder is link by it's instrumentation profile to the emulator. The instrumentation file lives in the directory specified as part of the configure parameter `--with-instrm=<instrm_dir>`, where `instrm_dir` will have a minimum of 2 files with the following exact names:

- profile.inst
- make.rules

`profile.inst` holds the instrumentation rules that hooks to the emulator. See Chapter 3 for syntax of the instrumentation profile. `make.rules` are extra rules that need to be incorporated into the `make` file so that the correct dependencies can be determined. Also, any extra defines and link options are included.

4.4 Future Work

Here's a list of things that can be done to improve the system.

4.4.1 Fragment Linking

The way the fragments are linked can be improved at branch exits and on indirect branch exits.

1. Branch exit: The current implementation of full fragment linking involves patching of instructions to jump to the corresponding fragment entry. An out portal has the form of:

```
save..
call +2
nop
read condition code
setup parameters
jump to (jump_out_fragment)
```

The patching takes place by the Dispatcher (part of `jump_out_fragment`). If the edge addresses have a corresponding fragment generated, the exit is patched so that subsequent exits will jump directly to it. The instructions at the out portal are patched with a branch to the target address; either

```
sethi %hi(target_fragment) ....
jmpl %lo(target_fragment) ...
nop
```

or

```
ba target_fragment
nop
```

To find where the out portal is, the patcher uses `%i7`. Note that the portal has a call instruction, which when executed will store the current `%PC` into `%o7`. Improvement can be done instead of patching the portals, patch the branch exit that jumps to the portal! This will give an estimated of 20% improvement in speed for compress95-O4. It removes 2 instructions in the fragments and one of them is a branch. Other benchmarks will probably benefit greatly as well, about 10%.

2. Indirect exit: Indirect jumps are generated to be compared with a predicted value (profiled at tracing time). Exits from an indirection suggest another value or a more accurate value (the next executing tail profiled value is not correct). An extra compare to the new value of the exit should be inserted to reflect the changes or program behaviour. Unfortunatly, space for the extra compare is not something an existing out portal would have. Currently, the PathFinder does half of fragment linking in that in does not patch the indirection out portal, but just jumps to it. The will incur the cost of context switch not every exit. To correct this, re-encode the exit at a separate location and add the extra compare to it. Also, patch the exit when appropriate.

4.4.2 Optimisations for V9

This will make the tool more like an SPARC to ULTRASPARC optimiser, to experiment with whether such approach is feasible in practice with large application programs.

1. Branch with predition: this was implemented, there was little or no improvement in the executed code.

2. Data prefetching: mostly safe and will probably give the most improvement.
3. Conditional moves: not useful for next executing tail, but can be useful for other schemes.
4. Replacing of V7 library calls with V9 floating point instructions.

Chapter 5

Debugger

Design: Bernard Wong [Oct 2001]; Documentation: Bernard Wong [Jan 2002]

The experiments with the emulator have so far been very positive and show much promise in enabling a more robust dynamic method to perform binary translation. Currently, the emulator works very well with most small conventional user programs on the SPARC. However, large amounts of time was required to debug the emulator in order for it to reach this working state for the SPARC Solaris platform. This is because it is very difficult to debug the emulator, as it often requires the programmer to read pages and pages of SPARC assembly code to spot the one mistake the emulator makes.

The goal of the graphical debugger for the emulator is to reduce the time and effort necessary in debugging the emulator for each new platform which it is to support. This goal drives the two main necessary requirements in the design of the debugger. These requirements are the following:

1. To allow easier and quicker debugging of the emulator, and
2. To run and support every platform which the emulator supports.

Please note that the emulator debugger currently depends on the the Unsafe package which is only available with Java 1.4 (in Beta testing at the time of writing).

5.1 Overview of Design

The debugger is written in the Java language and makes use of the Swing package. The program can be broken up into the following sections:

1. Graphical Section - `emuDebug.java`
2. Emulator Connection Section - `emuProcess.java`, `emuLib.skel`
3. Disassembler Section - `disasm.java`
4. Debugger Preprocessor - `emuDebugGen`

5.1.1 Graphical Section

The graphical section, as its name implies, is a collection of classes that deals with the different graphical parts of the debugger. Any modification to GUI should be made to this section. There are currently 5 main windows to the debugger: Disasm Output, Command Window, Register Window, Trace Window and Misc. Window.

Some of the GUI components' functionality are not currently implemented. These include the Relocation radio buttons, the Trace Window, the Float Registers frame and the View at Mem Address box. The Relocation buttons and Trace Window are intended for use by the PathFinder, as it would give the PathFinder tool the ability to display addresses and assembly instructions of the collected traces which would greatly help debugging that tool as well.

5.1.2 Emulator Connection Section

The emulator connection section is a separate Java program which interacts with the emulator and communicates with the rest of the debugger via sockets. Every time a breakpoint is reached the emulator connection section will send the current emulator state information to the graphical section serving as a bridge between the debugger and the emulator.

The main reason for separating the emulator connection section into a separate program from the graphical section is because of the stability of the emulator. During many of the development stages of the emulator, it would often crash if it encountered certain combination of instructions. If the emulator, which is a native C program, crashes, it will cause the emulator connection section to also crash as the emulator connection section is connected to the emulator via the Java Native Interface (JNI). By separating the emulator connection section from the graphical section, it allows the debugger to gracefully recover from the crash and collect all the relevant emulator state information just before the crash occurred without requiring a restart of the entire debugger.

5.1.3 Disassembler Section

The disassembler section uses the existing automatically generated disassembler and formats the disassembled information into a format compatible with the Graphical Section. Since the

automatically generated disassembler is generated by the same tool as the automatically generated emulator, any platform which the emulator supports would also be supported by the disassembler. Therefore, using this disassembler will help the debugger satisfy its requirement to support all the platforms which the emulator supports.

5.1.4 Debugger Preprocessor

Finally, the debugger preprocessor is an effort to allow the debugger to easily support every platform which the emulator supports. Places in the debugger that require platform specific information are marked with special preprocessor symbols that are later replaced with platform specific code that is automatically generated via the use of specification files. A very simple parser used to perform the replacements can be found in the `emuDebugGen` directory. The specification file is `machine/sparc/emuDebugSPARC.spec` (for SPARC machines) and contains the following section headings: `ConditionCodes`, `IntegerRegisters`, `ProgramCounter`, and `MiscRegisters`. The specification definition is currently only sufficient to support SPARC instructions. Additional headings support will need to be added in order to support other platforms.

Performing the command

```
java CodeGen src/machine/sparc/emuDebugSPARC.spec \
     src/machine/sparc/emuDebugSPARC.m \
     src/machine/sparc/emuLib.skel
```

will generate the final platform specific Java files from the skeleton and specification files. The included make file will already perform the necessary Java files generation.

5.2 Current Status

The current version of the debugger allows for the execution of the emulator to be controlled via the graphical panels of the debugger. The emulator can be told to emulate one instruction at a time (stepping through emulated machine instructions), or can be told to stop at breakpoints specified by memory addresses. Breakpoints can be added graphically as the assembly instructions of the executing program are shown. However, the assembly instructions when the program enters a library file is not shown. Therefore it is best to compile the executable to be emulated statically during the debugging phases. At each breakpoint, the current integer register, control flags, PC and nPC and other important register values are shown. The debugger can also restart the emulator at any point of execution.

Currently, only the SPARC emulator is supported by the debugger. In order to support other platforms, the specification file definition will need to be expanded. A more mature parser may need to be written in order to easily parse an expanded specification definition (the current parser is only meant to quickly test the feasibility of using specification files to generate Java files).

Support for PathFinder is also not currently implemented. Supporting Pathfinder with the debugger would be a very worthwhile feature as it would greatly help to debug the complex PathFinder code.

Appendix A

Building Walkabout

Documentation: Cristina [Aug 01, Jan 02]

This chapter contains notes on how to configure the Walkabout framework for a given platform. The tools that can be built within the Walkabout framework include an emulator, a disassembler, a pathfinder and a debugger. Most of the examples are for the SPARC architecture as this was our development platform.

A.1 Compilers and Tools Needed to build Walkabout

We use gcc 2.95.3, however, we do not make use of any of the new classes that are not available in 2.95-2, such as sstream. Note that we make use of namespaces sparsely in the code and these are not supported by the gcc 2.8.1 version of the compiler, but they are in egcs-1.1.2 (gcc 2.91.66).

For debugging, gdb 5.0 works well with gcc 2.95.3.

A.1.1 Special tools needed to build Walkabout

Walkabout has many source files that are generated from other source files, or from specifications. It is possible to make Walkabout without installing these tools, but if you want to make significant changes to Walkabout, you will need those tools.

To make Walkabout without the special tools, use the `--enable-remote` configuration script (see above).

The special tools are as follows.

- The New Jersey Machine Code Toolkit, ML version. This tool reads machine specifications, and in association with a matcher (`.m`) file, generates binary decoders. For details and downloading, see `http://www.eecs.harvard.edu/ nr/toolkit/ml.html`.

- Bison++ and Flex++, C++ versions. Note that the GNU tool bison++ is *not* suitable; Walkabout needs the special versions from France, which are C++ aware. If you get lots of errors from running bison++, you have probably got the wrong version! Download these tools from `ftp://ftp.th-darmstadt.de/pub/programming/languages/C++/tools/flex++bison++/LATEST/` You might also try downloading these tools from one of the various mirror sites such as `http://sunsite.bilkent.edu.tr/pub/languages/c++/tools/flex++bison++/LATEST/`. To test if you have the correct version, you should get results similar to:

  ```
  % bison++ --version
  bison++ Version 1.21-7, adapted from GNU bi-
  son by coetmeur@icdc.fr
  ```

  If searching the web for these tooks, include the author's name ("coetmeur") as a keyword.

- The Tcl shell (`tclsh`). This tool is only needed to run the regression test script (`test/regression.test`). `tclsh` and the `tcltest` package are part of Tcl/Tk releases 8.0 and newer. You may well find that these are already installed on your Linux or other system. Otherwise, see web pages such as `http://www.sco.com/Technology/tcl/Tcl.html`.

A.2 Configuration Notes

In order to instantiate a translator out of the Walkabout framework, you need to configure Walkabout to run on your host machine by instantiating a set of source and target machines. Figure A.1 lists the names used within Walkabout to describe machine specifications, and the associated instruction set version that is specified.

Name	Description
sparc	SPARC V8 (integers and floats)
pent	80386 (integers and floats)

Figure A.1: Names of Machines and Versions Supported by the Walkabout Framework

You can get help from the configure program at any point in time by emitting the following command:

```
./configure --help
```

Figure A.2 shows the options used by Walkabout from the `configure` program.

Option	Description
–enable-remote	don't try to regenerate generated files
–enable-debug[=< $what$ >]	enable debugging support, < $what$ > is one of ***
–with-source=< $arch$ >	translate from < $arch$ > architecture, one of sparc, pent
–with-instrm=< dir >	add instrumentation to emulator using files in < dir >

Figure A.2: Configure Options

A.3 Configuring Tools from the Walkabout Framework

At present (Aug 2001), you can generate disassemblers, interpreters and a PathFinder program using the Walkabout framework. In order to generate these tools, the framework has to be configured using different options. These notes describe how to `configure` Walkabout for different purposes. More information about how `configure` works is available in Section A.4. Note that in order to build a different tool you always need to reconfigure your system.

A.3.1 Generating Interpreters

In order to generate interpreters, you need to first build the tool that generates interpreters, `genemu`:

```
configure --with-source=sparc --enable-remote
make dynamic/tools/genemu
```

This generates the file `genemu` in the `./dynamic/tools` directory.

The `genemu` tool will create an interpreter based on the syntax (SLED) and semantic (SSL) specifications for a machine. Both a C-based interpreter and a Java-based interpreter can be generated using `genemu`, although the Java-based interpreter has only been tested with the SPARC specifications.

The options available in `genemu` are:

```
Usage: genemu [options] <sled-filename(s)> <ssl-filename>
Recognized options:
```

```
-c  output C code [default]
-d  disassembler only (do not generate emulator core)
-j  output Java code
-i  inst-filename: use inst-filename to instrument code.
-t  test only, no code output
-m  additionally generate a Makefile to go with the core
-o <file>  write output to the given file
```

Note that one or more SLED files can be given as input, SLED files have the extension `.spec` and SSL files have the extension `.ssl`. Also, some options have not been maintained, it is best to use the configure options (see next sections) or see the configuration files for examples of usage.

C-based Interpreters

To generate a C-based interpreter for a particular machine, use the `make` rule for that machine (normally the name of the machine followed by "emu"). For the SPARC, you would run:

```
make sparcemu
```

This will generate `sparcemu` in the `./dynamic/emu` directory, if you had configured for the SPARC machine.

To run:

```
cd dynamic/emu
sparcemu ../../test/sparc/hello
sparcemu /bin/banner Hi
```

Java-based Interpreters

To generate a Java-based interpreter for the SPARC machine, use the following make rule:

```
make dynamic/emuj
```

You can then run the generated interpreter using a Java VM:

```
cd dynamic/emu
java -cp . sparcmain ../../test/sparc/hello
```

A.3.2 Generating PathFinder

To build a virtual machine that finds hot paths and generates native code for those paths while interpreting other cold paths, you need to configure Walkabout to generate the `pathfinder` virtual machine (VM) by configuring for a particular instrumentation method to determine hot paths within the interpreter. For example, if you want to use Dynamo's next executing tail (NET) method, run the following configure command:

```
   configure --with-source=sparc --enable-remote \
             --with-
instrm=dynamic/pathfinder/sparc.NET.direct/pathfinder
```

Then make the tool by building a target with the name formed by adding a single-character prefix to the word "pathfinder". This prefix character is the first letter of the name of the machine on which the genberated `pathfinder` will run. For example, for SPARC you would run:

```
   make spathfinder
```

This generates the instrumented VM `spathfinder` in the `./dynamic/emu` directory. This instrumented VM uses code profiling as well as code generation to execute code for SPARC V8. Note, however, that the source code for the code generator relies on some SPARC V9 instructions.

To run:

```
   cd dynamic/emu
   spathfinder ../../test/sparc/hello
   spathfinder ../../test/sparc/fibo-O0
   spathfinder /bin/banner Hello
```

A word of caution when building interpreters and VMs. Some of the files used by these tools are the same and some get patched, therefore, it is wise to remove object files before building a new tool. We therefore recommend you do a `make dynclean` before you build your tool. If you are getting strange errors, most likely you need to `make dynclean`. Since you need to run configure, you can do this at the same time:

```
    make dynclean
    configure --with-source=sparc ...  [whatever other options]
```

You can generate other pathfinder tools for the SPARC architecture by using some of the other instrumentation files. The above example made use

of the `dynamic/pathfinder/sparc.NET.direct/pathfinder/profile.inst` instrumentation file. Other instrumentation files can be used, the ones in the Walkabout distribution are in the following locations:

```
   dynamic/pathfinder/sparc.NET.direct/pathfinder-
call/profile.inst
   dynamic/pathfinder/sparc.NET.direct/pathfinder-
recursive/profile.inst
   dynamic/pathfinder/sparc.NET.direct/pathfinder.v9/profile.inst
   dynamic/pathfinder/sparc.NET.relocate/profile.inst
```

A.3.3 Generating the Walkabout Debugger

The Walkabout debugger is a GUI debugger written in the Java language. The debugger was only ever tested with the SPARC architecture, other extensions would be needed to support the display of state information for other architectures.

To configure the debugger, emit the following commands:

```
   make dynclean
   configure --with-source=sparc --enable-dynamic --enable-remote
   make dynamic/emuDebug
```

The make builds the disassembler and the interpreter for the configured machine, and then generates `emuDebug.class` in the `./dynamic/emuDebug/bin` directory.

To run the debugger, execute the bash script `emuDebug` in the `./dynamic/emuDebug` directory (this is a shell script file; make sure the first line refers to the location of your `bash` tool):

```
   cd dynamic/emuDebug
   emuDebug ../../test/sparc/hello
```

A.3.4 Building without the `--with-remote` Option

When configuring the system *without* use of the remote option (i.e. without `--with-remote`), the system will recreate .m and .cc files from the machine specifications and .c files from .y files.

It is recommended that you do a `make realdynclean` before configuring without the remote option, in order to remove all generated files that have already been stored in the distribution.

A.4 How the Configuration Process Works

A complete description of the autoconfigure process is beyond the scope of this document; the interested reader can get more information from publicly available documentation such as `http://www.gnu.org/manual/autoconf/index.html`.

In brief, the developer writes a file called `configure.in`. The program `autoconf` processes this file, and produces a script file called `configure` that users run to configure their system. We have already done that, so unless you need to change the configuration, you only need to run `./configure`. If you do make a change to `configure.in`, then you should run

```
autoconf; autoheader
```

When `./configure` is run, various files are read, including a file specific to the source machine. For example, if you configure with `--with-source=sparc`, the file `machine/sparc/sparc.rules` is read for SPARC-specific information. It also reads the file `Makefile.in`. Using this information and the command line options, `./configure` creates the file `Makefile`. As a result, the `Makefile` isn't even booked in. That's the main reason you need to run `./configure` as the very first thing, before even `make`. It also means that you should not make changes (at least, changes that are meant to be permanent) to `Makefile`; they should be made to `Makefile.in`.

Another important file created by `./configure` is `include/config.h`. This file is included by `include/global.h`, which in turn is included by almost every source file. Therefore, `configure` goes to some trouble not to touch `include/config.h` if there is no change to it (and it says so at the end of the `configure` run). A significant change to the configuration (e.g., choosing a new source or target machine) will change `include/config.h`, and therefore almost everything will have to be recompiled.

A note about the version of `autoconf`; we have found that version 2.9 does not work properly but version 2.13 works fine with our `configure` files.

A.4.1 Dependencies and `make depend`

Building Walkaboutrequires a file called `.depend` that contains file dependencies for the system. The first time you `make` Walkabout, this file won't exist and it will be created automatically for you. The `.depend` file contains entries similar to this:

```
coverage.o: ./coverage.cc include/coverage.h include/global.h \
 include/config.h
```

which says that the `coverage.o` file depends on the files `./coverage.cc`, `include/coverage.h`, and so on. There can be dozens of dependencies; the above is one of the smallest. This information takes a minute or two to generate, and so is only generated (a) by `make` itself if `.depend` does not exist, and (b) if the user types `make depend`.

It is easy to change the dependencies, e.g. by adding a `#include` line to a source file. If you do this, and forget to run `make depend`, then you can end up with very subtle make problems that are very hard to track down. For example, suppose you add "`#include "foo.h"`" to the `worker.cc` source file, so that `worker.cc` can use the last virtual method in class `foo`. Everything compiles and works fine. A week later, you add a virtual method to the middle of `class foo`. The `.depend` file doesn't have the dependency for `worker.cc` on `foo.h`, and so `worker.o` isn't remade. The code in `worker.o` now calls the second last method in `class foo`, instead of the correct final method! However, you are not thinking about `worker.cc` now, since your latest changes are elsewhere. This sort of problem can take a long time to diagnose.

One solution is to "`make clean`" as soon as you get unexpected results. However, you can save a lot of time if instead you just `make depend; make` instead. In fact, it's a good idea to run `make depend` regularly, or after any significant change to the source files.

A.4.2 Warnings from `make`

During the making of Walkabout, it is normal to see quite a lot of output. We try to ensure that ordinary warnings from gcc are prevented, but some warnings are much harder to suppress, and some warnings are quite normal. For example:

```
typeAnalysis/typeAnalysis.y contains 2 shift/reduce conflicts.
```

These are normal, and the bison++ parser automatically resolves these conflicts in a sensible way.

A.4.3 Where the Makefile Rules Are

The Makefile is composed of make rules that come from a variety of different sources:

- The core rules are in the top level `Makefile.in` file,
- Machine-specific rules are in the respective machine directory with the extension `.rules`; e.g., `machine/sparc/sparc.rules`,

- Instrumentation rules are in a subdirectory `pathfinder/make.rules` under the respective instrumentation directory; e.g., `pathfinder/sparc.NET.direct/pathfinder/make.rules`.

A.5 The Walkabout Regression Test Suite

The Walkabout framework includes a set of regression tests for generated interpreters. Tests have been written for the SPARC emulator only.

The script that is used for testing an interpreter is called `dynamic/test/interp-regression.tcl`. This is a Tcl script that allows new tests to be added easily. Existing tests can also be modified easily. Each test includes the test's name and expected result. The script runs each test and compares its output against that expected. If these are not the same, it reports the failure and gives the actual result. The default is to run all tests, but you can specify which tests to run, or which to skip, by giving a regular expression pattern that is matched against test names. At the end of all the tests, there is a report on the number of tests run, and how many passed, failed, or were skipped.

The tests each run one of the SPEC95 benchmark programs. Up to four regression tests can be run for each SPEC95 program. Both optimized (-O4) and unoptimized (-O) versions of each program are run. It is also possible to run each version of the program using both the SPEC95 "test" and "reference" data sets. By default, only the "test" data set is used since it requires less time. The other tests that use the "reference" data set are marked as having the "refInput" contraint; they are only run if you specify `-constraints refInput` on the command line that runs the regression tests (see below).

The expected output of each test is the total number of instructions executed. This count is sensitive to the specific versions of the shared libraries that are used. As a result, you can expect the tests to fail if they are run on a machine other than that used to get the original expected instruction count. We used a Sun 420R with 4 CPUs and 4GB of memory, running Solaris 8.

A.5.1 Running the Regression Tests

Tests can be run sequentially (one after another) or in parallel. This section discusses how to run the tests sequentially. Section A.5.2 describes how to run the tests concurrently.

To run all SPARC interpreter tests using the "test" data sets (the default):

```
$ cd <workspace>
$ cd dynamic/test
```

```
$ tclsh interp-regression.test sparc
```

To run only the SPARC interpreter tests that run the SPEC95 “go” program, you can specify a pattern on the command line. Only tests (“test” data sets only) with names that match the pattern are run:

```
cd dynamic/test
tclsh interp-regression.test sparc -match 'go*'
```

To skip all SPARC interpreter tests whose names match a pattern (again “test” data sets only), run:

```
cd dynamic/test
tclsh interp-regression.test sparc -skip 'go* ijpeg*'
```

To run all SPARC interpreter tests including those that use the “reference” data sets:

```
cd dynamic/test
tclsh interp-regression.test sparc -constraints refInput
```

A.5.2 Running the Tests in Parallel

To run the tests in parallel, the two scripts `dynamic/test/mt-tests.tcl` and `dynamic/test/summarize-mt-tests.tcl` are used. `mt-tests.tcl` takes the same command line arguments as `interp-regression.tcl` and forks processes to do the various tests in parallel. It creates a directory under `dynamic/test` with a name like `walktest-Oct-24-2001-19:15:06` that contains an “*.out” file holding the output from each of the forked tests. One such file, for example, might be `go.v8.base.test.out`.

The second script `summarize-mt-tests.tcl` creates a file that summarizes the results of all the tests. It takes one command line argument, the name of the directory created by `mt-tests.tcl`, and creates a file `summary.txt` in that directory with the concatenated results of the various tests.

For example, to run all SPARC interpreter tests except for “go” using the default “test” data sets, do the following.

```
$ cd <workspace>/dynamic/test
$ tclsh mt-tests.tcl sparc -skip 'go*'
Parallel Walkabout tests
Writing results into directory walktest-Oct-24-2001-16:10:44
```

```
    Forked process 7952 to execute test compress.v8.peak.test
    Forked process 7953 to execute test compress.v8.base.test
    Forked process 7954 to execute test perl.v8.peak.test
    Forked process 7955 to execute test perl.v8.base.test
    Forked process 7956 to execute test go.v8.peak.test
    Forked process 7957 to execute test go.v8.base.test
    Forked process 7958 to execute test ijpeg.v8.peak.test
    Forked process 7964 to execute test ijpeg.v8.base.test
    $ tclsh summarize-mt-tests.tcl walktest-Oct-24-2001-16:10:44
    Summarizing parallel Walkabout test results into walktest-Oct-
24-2001-16:10:44/summary.txt
    appending file compress.v8.base.test.out
    ...
    $ vi walktest-Oct-24-2001-16:10:44/summary.txt
```

Bibliography

[1] 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.

[2] 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.

[3] 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, Sun Microsytems Laboratories, Palo Alto, CA 94303, December 2001.

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

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

[6] 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, ACM, January 1984.

[7] Digital. OM and ATOM. `http://www.research.digital.com/wrl/projects/om/om.html`, 1994. Digital Western Research Labs.

[8] Digital. Freeport Express. `http://www.novalink.com/freeport-express`, 1995.

[9] 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.

[10] 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.

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

[12] 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*, pages ??–?? Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2000.

[13] U. Holzle, L. Bak, S. Grarup, R. Griesemer, and S. Mitrovic. Java on steroids: Sun's high-performance java implementation, August 1997.

[14] 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.

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

[16] J.R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software – Practice and Experience*, 24(2):197–218, February 1994.

[17] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Languages, Design and Implementation*, pages 291–300, June 1995.

[18] 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, Berkeley, USA, April 23–24 2001. USENIX Association.

[19] C. Reeve, D. Deaver, R. Gorton, and B. Yadavalli. Wiggins/redstone (wr): A dynamic optimization and specialization tool. Unpublished manuscript, 2000.

[20] E. Schnarr and J.R. Larus. Instruction scheduling and executable editing. In *Workshop on Compiler Support for System Software (WCSSS)*, pages 288–297, Tucson, AZ, February 1996.

[21] K. Scott and J.W. Davidson. Software security using software dynamic translation. Technical Report CS-2001-29, University of Virginia, Department of Computer Science, Charlotesville, VA, November 2001.

[22] 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.

[23] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Digital Western Research Laboratory, 250 University Avenue, Palo Alto, CA 94301, March 1994.

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

[25] D. Ung and C. Cifuentes. Optimising hot paths in a dynamic binary translator. In *Proceedings of the Second Workshop on Binary Translation*, Philadelphia, Pennsylvania, October 19 2000.

[26] 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.