

SABLEJIT: A RETARGETABLE JUST-IN-TIME COMPILER

by

David Bélanger

School of Computer Science
McGill University, Montreal

August 2004

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Copyright © 2004 by David Bélanger

Abstract

In this thesis, we introduce SableJIT, a retargetable just-in-time compiler for the SableVM Java virtual machine. Our design attempts to minimize the amount of work required to port (or to retarget) our compiler to a new platform. We accomplish this in three ways. First, we introduce a retargetable backend where the amount of work required to port it is reduced to the implementation of simple well-defined primitive functions. Second, we keep all code related to the internals of the virtual machine in the frontend, making knowledge of the target architecture sufficient for a port. Finally, we provide a good development environment that favours incremental development and testing, in part through a robust runtime system that can recover from compilation failures.

We demonstrate the portability of SableJIT by supporting three processor architectures and various operating systems. In particular, we describe the experience acquired in porting our compiler to the Solaris/SPARC platform.

Résumé

Dans cette thèse, nous présentons SableJIT, un compilateur juste-à-temps recible pour SableVM, une machine virtuelle Java. Notre désign tente de minimiser l'effort requis pour porter (ou recibler) notre compilateur à une nouvelle plateforme. Nous accomplissons ceci en trois façons. Premièrement, nous présentons un dorsal recible dans lequel le montant de travail requis pour le porter est réduit à l'implémentation de plusieurs fonctions primitives bien définies. Deuxièmement, nous gardons tout le code concernant les détails du fonctionnement interne de la machine virtuelle au niveau du frontal, de sorte que seules les connaissances de la plateforme ciblée soient nécessaire lors d'un portage. Troisièmement, nous fournissons un environnement facilitant le développement incrémentiel, en partie grâce à un système d'exécution pouvant se relever des défaillances lors de la compilation.

Nous démontrons la portabilité de SableJIT en supportant trois architectures de processeurs et divers systèmes d'exploitation. En particulier, nous décrivons l'expérience acquise en portant notre compilateur à la plateforme Solaris/SPARC.

Acknowledgments

I would like to thank my co-supervisors Professor Laurie Hendren and Etienne Gagnon. I thank Laurie for her advice, feedback and financial support during the course of my graduate studies. I thank Etienne for his feedback and technical support.

I would also like to thank Christian Arcand at UQAM for his work in porting SableJIT to the Solaris/sparc platform and for his feedback.

The McGill Sable Research Group members deserve special thanks for making the Sable lab a pleasant work environment and a place for technical discussion. In particular, I would like to thank Bruno Dufour for his help in the collection of empirical data by providing sample scripts.

Finally, I would like to thank the School of Computer Science at McGill for providing financial assistance through an IT Fellowship at the beginning of my graduate studies and NSERC for funding subsequent semesters.

Contents

Abstract	i
Résumé	ii
Acknowledgments	iii
Contents	iv
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Challenges	2
1.2 Thesis Contributions	3
1.3 Thesis Organization	5
2 Background	6
2.1 Interpreters	6
2.1.1 Switch	6
2.1.2 Direct-threaded	7
2.1.3 Inline-threaded	8
2.2 Virtual Machine Instruction Set	9
2.3 Preparation Sequences	10

3	Overview	12
3.1	Introduction to SableJIT	12
3.2	Architecture Overview	13
3.2.1	SableJIT Runtime	13
3.2.2	Compiler Frontend	15
3.2.3	Compiler Backend	16
3.3	Summary	17
4	SableVM/SableJIT Interface	18
4.1	Method Invocation Bytecode Instructions	18
4.2	Interpreter-Only Mode	20
4.3	Bootstrapping the Virtual Machine and Compiler	20
4.3.1	Bootstrapping the Virtual Machine	21
4.3.2	Bootstrapping the Compiler	21
4.3.3	The Compiler Class Loader	22
4.4	Invoking the Compiler	23
4.4.1	Where Compilation May Take Place	23
4.4.2	When Compilation Occurs	25
4.4.3	Data Required for Compilation	25
4.4.4	Invoking the Compiler	34
4.5	Invoking Compiled Code from the Interpreter	35
4.6	Method Invocation in Compiled Code	37
4.6.1	Interpreted Methods	37
4.6.2	JNI Native Methods	39
4.6.3	Method Invocation Overhead	40
4.7	Summary	40
5	The Baseline Compiler	41
5.1	A Simple Baseline Compiler	41
5.1.1	General Overview	41
5.1.2	Main Components of the Baseline Compiler Frontend	42

5.1.3	The Compilation Process	44
5.1.4	Secondary Data Structures	46
5.1.5	Dynamic Recompilation, Decompilation and Preparation Sequences	47
5.2	Using Runtime Information	49
5.2.1	Stack Sizes	49
5.2.2	Method Synchronization	50
5.2.3	Initialization of Non-Parameter Reference Type Local Variables	51
5.3	Towards an Optimizing Compiler	52
5.4	Compiler Robustness and Failure Recovery	53
5.4.1	Failure Classes	53
5.4.2	Recoverability	55
5.4.3	The Recovery Process	56
5.5	Flexible Method Entry Point and Compilation Unit	57
5.5.1	Flexible Entry Point	57
5.5.2	Flexible Compilation Unit	58
5.6	Summary	59
6	Exception Handling	60
6.1	General Exception Handling	60
6.1.1	Interpreter-only Exception Handling	60
6.1.2	Joint Interpreter and Compiled Code Exception Handling	64
6.2	Signal-based Exception Handling	64
6.2.1	Signal-based Exceptions in Interpreted Code	66
6.2.2	Signal-based Exceptions in Compiled Code	67
6.2.3	Efficient Array Bounds Check	69
6.2.4	Hardware Support on Various Architectures	69
6.3	Summary	70
7	Memory Management	71
7.1	Memory Partitions in SableVM	71

7.2	Memory Requirements of SableJIT	72
7.3	SableJIT Memory Manager	74
7.4	Garbage Collection	75
7.5	Garbage Code Collection	76
7.6	Memory Management Within the Compiler	76
7.7	Summary	77
8	Retargetability	78
8.1	Backend Architecture	78
8.2	RT Code Model	81
8.2.1	Instruction Types	82
8.2.2	Naming Convention	83
8.2.3	Default Implementation	83
8.2.4	RT Instruction Set	84
8.2.5	Example	90
8.3	Register Allocation	92
8.3.1	Register Requirements and Usage	93
8.4	x86: Problems and Solutions	93
8.4.1	2-operand Instructions	94
8.4.2	Register Specific Usage	95
8.4.3	Floating Point Register Stack	95
8.4.4	Limited Number of Registers	95
8.5	Architecture Independent Functionality Provided	97
8.5.1	Maintaining Code Arrays	97
8.5.2	Architecture Independent Branch Patching	98
8.5.3	Platform Independence of the Memory Layout of Data Structures	99
8.6	Summary	100
9	Porting Strategy and Experience	101
9.1	Suggested Porting Strategy	101
9.1.1	Porting the SableJIT Runtime	102

9.1.2	Porting the SableJIT Compiler	103
9.2	Evaluation of Porting Effort	106
9.2.1	Evaluation in Man Hours	107
9.2.2	Evaluation in Lines of Code	107
9.3	Summary	108
10	Experimental Results	109
10.1	Test Platforms	109
10.2	Benchmarks	110
10.3	Results	110
10.3.1	Interpreters and Compilers	110
10.3.2	Interpreter-Only Mode	115
10.4	Summary	116
11	Related Work	117
11.1	Retargetable Code Generators	117
11.1.1	Code-Generator Generators	117
11.1.2	VCODE	118
11.1.3	GNU Lightning	120
11.1.4	LIL	121
11.1.5	The Virtual Processor	121
11.2	Portability in Virtual Machines and JITs	122
11.2.1	OpenJIT	122
11.2.2	Kaffe	123
11.2.3	Jikes RVM	124
11.2.4	Sun's HotSpot	125
11.3	Summary	126
12	Conclusion and Future Work	127
12.1	Summary and Conclusions	127
12.2	Future Work	128

Appendices

A SableJIT User Guide	130
A.1 Getting and Installing SableJIT	130
A.2 Customizing SableJIT	131
A.2.1 Source Organization	131
A.2.2 Configure Options	132
A.3 Running SableVM with SableJIT	133
A.4 -C Options	134
Bibliography	135

List of Figures

2.1	Switch Interpreter	7
2.2	Direct-threaded Interpreter	8
2.3	Inlined Sequence	9
2.4	Inline-threaded Interpreter	9
2.5	Preparation Sequence	11
3.1	SableJIT Architecture	14
3.2	Compiler Frontend	16
3.3	Compiler Backend	17
4.1	Method Invocation Bytecodes and Transitions	19
4.2	Prototype of <code>Compiler.compile</code>	26
4.3	Direct-Threaded Code to Switch Code Conversion	28
4.4	Inline-Threaded Code	30
4.5	Inline-Threaded Code to Switch Code Conversion	33
4.6	Java Stack	36
4.7	Stubs for Compiled Code	38
5.1	Implementation of <code>build_iaload</code>	43
5.2	Compiled Code Organization	44
5.3	Caller Work Delegated to the Compiled Callee	52
5.4	Compiler Frontend	53
5.5	Compiler Frontend with a MIR Builder	53
5.6	Partial Class Hierarchy of Exceptions and Errors	54

6.1	Compiled Code Signalling the Occurrence of An Exception	62
6.2	Interpreter-only Exception Handling	63
6.3	Compiled Code / Interpreter Mixed Exception Handling	65
6.4	GETFIELD_INT Implementation <i>With</i> Signal-Based Exceptions	67
6.5	GETFIELD_INT Implementation <i>Without</i> Signal-Based Exceptions	67
6.6	Mapping Native Addresses to Bytecode PCs	68
7.1	Native Code and Data Layout.	73
8.1	Compiler Backend	79
8.2	Code Segments Affected by ABI Differences	80
8.3	Differences in Native Stack Layout on the PowerPC	80
8.4	Native Stack Layout Used by Compiled Code on PowerPC	89
8.5	Memory Layout of Arrays	91
8.6	Implementation of <code>build_iaload</code> in RT Code	92
8.7	Native Stack Layout on x86	96
8.8	RT Code Segment Using a Label	99

List of Tables

6.1	Signal-based Exception Support on Various Architectures	70
8.1	Classes Implementing the Backend of Supported Platforms	80
8.2	RT Code Instruction Types	83
8.3	Selected RT Instructions - Part 1	85
8.4	Selected RT Instructions - Part 2	86
8.5	Register Usage on x86	97
9.1	Architecture Dependent Primitive Functions in the Runtime	102
9.2	Lines of Code for SableJIT	108
10.1	Performance Results for Linux/x86	111
10.2	Performance Results for Linux/ppc	111
10.3	Performance Results for Solaris/sparc	111
10.4	Compilation Times for Linux/x86	113
10.5	Runtime Overhead in Interpreter-Only Mode	115
10.6	Size of Executables	115

Chapter 1

Introduction

In recent years, Java has become a popular object-oriented programming language [GJSB00]. Java was designed with mobile computing and platform independence in mind. In particular, Java applications can be downloaded via a network to hosts in a heterogeneous environment, that is, varying in hardware and software platforms. Java programs consist of Java source files that are compiled into class files. Class files contain architecture-independent bytecodes. These bytecodes are then executed by a Java Virtual Machine (JVM). The JVM provides a runtime environment and it isolates the application bytecodes from the native architecture specifics.

Initially, virtual machines interpreted bytecodes. Interpreters are quite portable and various existing interpreters in virtual machines such as Kaffe [Kaf] and SableVM [Sabb] support several platforms. The drawback of interpreters is their high overhead leading to poor performance. Interpretation techniques such as direct-threading [Ert] and inline-threading [PR98] remove some of the overhead and such efficient interpreters can still be quite portable as demonstrated in [Gag02]. Performance can be further improved by compiling Java bytecodes into native code with an *ahead-of-time* (AOT) compiler such as GCJ [GCJ] and then executing the resulting binary directly on the native platform. This tends not to be the ideal solution mainly due to the highly dynamic nature of Java. Classes not available at compilation time could be dynamically loaded at execution time. GCJ solves this issue by resorting to an interpreter for such code. These problems can be overcome with a *just-in-time compiler*

(JIT). A JIT compiles Java bytecodes to native code at runtime. Since compilation happens dynamically at runtime as opposed to statically ahead-of-time, JITs are constrained as compilation time becomes part of execution time. They need to generate good code quickly. The main focus of JIT development has mainly been performance. By nature, JITs are quite platform-specific as is the case with all code generators. Unlike an interpreter, porting a JIT to a new target platform (or *retargeting* a JIT) involves a non-negligible amount of work. The term *retargeting* is sometimes preferred to porting as a significant part of the compiler usually needs to be rewritten for each new target.

Due to the difficulty in retargeting a JIT, efficient virtual machines (i.e. including JITs) are available for popular platforms such as Win32 and Linux on PCs whereas less popular platforms are usually left aside. Although JVMs are available on a variety of hardware/software platforms, *efficient* JVMs are so at a far lesser degree. End users interested in a high performance JVM are limited to a smaller choice of platforms although Java applications are designed to be run everywhere.

In this work, we study portability issues in JITs. In order to do so, we designed SableJIT, a retargetable JIT for SableVM.

1.1 Challenges

In this thesis, we would like to study and solve the following challenges:

- We would like our retargetable compiler to be a natural extension of SableVM. In particular, it should take advantage of existing SableVM features and it should fit nicely into the SableVM design.
- We are interested in studying the relationship between the inline-threaded interpreter (the fastest of three available interpreters) in SableVM, with a naive JIT. In particular, we would like to answer the question: *how much performance improvement could be achieved by going to the next level, that is, by removing the remaining instruction dispatch overhead in the fastest interpreter?*

- We would like to keep the amount of work required to port our compiler minimal. We would also like to study the feasibility of using a retargetable code generation engine based on VCODE [Eng96].
- We would like to provide a good development environment. Since a variety of architectures exist, we would like to make it as pleasant as possible for other developers to add support for additional architectures. Also, we would like to be able to easily integrate new features in the current design such as an optimizing compiler.
- We would like to apply our framework to several architectures in order to test its design and to experiment with various retargetability issues. In particular, we would like to answer the question: *how easily can SableJIT be retargeted to a new architecture?*

1.2 Thesis Contributions

The result of this research culminates with the design and implementation of SableJIT, a retargetable compiler for SableVM.

The main contributions of this thesis are:

- The implementation of a retargetable JIT that works almost seamlessly with the interpreter. Support for garbage collection is simple and our implementation for signal-based exceptions uses the virtual machine infrastructure already in place.
- The implementation of a baseline compiler that generates code in one pass plus a patching phase. Our baseline compiler mimics the interpreter and thus takes us one level further from the inline-threaded interpreter as all instruction dispatch overhead is removed by compilation.
- The design of a retargetable backend based on VCODE [Eng96], a retargetable code generator providing assembly-like instructions. VCODE was originally

implemented as a set of C macros and functions. Our implementation, in Java, is made suitable for a Java JIT context by being fully re-entrant, by using strong types and by having Java-like semantics for instructions. In addition, for greater flexibility, our implementation generates relocatable code.

Our backend uses a register-based RISC model. Architectures with a small number of registers such as the x86 offer a challenge. To get around the limited register set of the x86 architecture, our implementation uses part of the native stack as *virtual registers* as suggested in [Eng96]. These virtual registers are used transparently by the compiler frontend. Our experimental results were surprising: with a clever use of its small register set along with virtual registers, the x86 architecture outperformed the two supported RISC architectures: PowerPC and SPARC.

The work required to retarget SableJIT is kept minimal. In particular, support for branch patching and jump tables is provided. Also, knowledge of the virtual machine internals should not be required.

- The design of a robust compiler runtime with compilation failure recovery and the implementation of a testing framework. These features are useful for porting our compiler to a new platform and for developing new features. Failure recovery is particularly useful in that unimplemented features or features known to be buggy can be skipped for the time being by simply throwing a compilation exception. Methods that cannot be compiled are marked as *uncompilable* and are interpreted instead. This favours incremental development and testing as compilation failures do not lead to the immediate termination of the virtual machine.
- The implementation of a backend supporting three processor architectures and various operating systems: x86 (Linux, FreeBSD), PowerPC (Linux, Mac OS X) and SPARC (Solaris). In particular, we contribute our porting experience to the SPARC architecture, the PowerPC and x86 mainly serving as development platforms.

1.3 Thesis Organization

In chapter 2, we present some background information. Chapter 3 introduces the design and architecture of SableJIT. In chapter 4, we cover the SableJIT runtime: the interface between the virtual machine and the compiler. In chapter 5, we present the baseline compiler. In chapter 6, we describe exception handling. In chapter 7, we present memory management. In chapter 8, we present the retargetable backend. In chapter 9, we describe our experience in porting SableJIT to the Solaris/sparc platform. In chapter 10, we present experimental results. In chapter 11, we review the literature on retargetable dynamic code generators (i.e. generating code at runtime) and we study the portability in various Java virtual machines. Finally, we conclude in chapter 12.

Chapter 2

Background

In this chapter we present the required background information for understanding subsequent chapters. Topics covered are the three interpreters in SableVM, the extended instruction set used by the interpreters, and finally the preparation sequences for efficient code execution. Further information on these topics can be found in [Gag02].

2.1 Interpreters

SableVM is an efficient and portable Java virtual machine. SableVM has three interpreters: a switch, a direct-threaded, and an inline-threaded interpreter. The three interpreters have different portability / performance tradeoffs.

2.1.1 Switch

The switch interpreter is the simplest of all three. Figure 2.1 illustrates the basic implementation of a switch interpreter. This figure as well as others presented later in this chapter are adapted from [Gag02]. In a switch interpreter, the bytecode is contained in a code array. The code is dispatched by reading the next bytecode instruction (indicated by the `pc` pointer) from the code array and then executing the switch case implementing that bytecode instruction.

```
char code[] = { ICONST_1, ICONST_2,
               IADD, ... };
char *pc = code;
int stack[STACKSIZE];
int *sp = stack;

...

while (true) {
    switch (*pc++) {
        case ICONST_1: *sp++ = 1; break;
        case ICONST_2: *sp++ = 2; break;
        case IADD: --sp; sp[-1] += *sp; break;
        ...
    }
}
```

Figure 2.1: Switch Interpreter

A switch interpreter has a high dispatch cost. Three branch instructions are required per bytecode instruction: one to get back to the loop head, one to test if the instruction opcode is within bounds of the cases in the switch statement, and finally one to jump to the corresponding case statement.

2.1.2 Direct-threaded

A direct-threaded interpreter improves over a switch interpreter by decreasing the dispatch cost. This is done by putting, in the code array, addresses to bytecode implementations rather than bytecode (integer) opcodes. The address of an implementation can be obtained by using the GNU *label-as-value* C extension or through other means such as assembly code. The GNU C `&&` operator is used to obtain the address identified by a label. Figure 2.2 shows the general structure of the interpreter. The `pc` points to the next instruction to execute. Execution is started by loading the first address from the code array and then performing an indirect jump to that location. Note that some dispatch code follows each bytecode implementation.

```
void *code[] = { &&ICONST_1, &&ICONST_2,
                &&IADD, ...};
void **pc = code;
int stack[STACKSIZE];
int *sp = stack;

goto **(pc++);

ICONST_1: *sp++ = 1; goto **(pc++);
ICONST_2: *sp++ = 2; goto **(pc++);
IADD: --sp; sp[-1] += *sp; goto **(pc++);
...
```

Figure 2.2: Direct-threaded Interpreter

The direct-threaded interpreter removes the loop and the switch statement overhead. A single branch instruction is now required to dispatch each bytecode instruction. Note that in the implementation found in SableVM, the direct-threaded code array has the same format as the switch code array with the exception of integer opcodes that have been replaced by addresses.

2.1.3 Inline-threaded

An inline-threaded interpreter further reduces the number of dispatches over the direct-threaded interpreter. The implementation of several bytecodes are copied consecutively in memory by using the *label-as-value* extension. These sequences of bytecodes are called *inlined sequences*.

Figure 2.3 illustrates the inlined sequence corresponding to the bytecode sequence: ICONST_1, ICONST_2, and IADD. Note that this removes the instruction dispatch between instructions within a sequence. The dispatch code is located at the end of the sequence. Figure 2.4 illustrates how such sequences are dispatched. The address of sequences are stored in the code array. The first sequence is dispatched by loading its address from the array and doing an indirect jump to it. Further sequences are dispatched by the dispatch code located at the end of the sequence implementation.

```
ICONST_1 body: *sp++ = 1;
ICONST_2 body: *sp++ = 2;
  IADD body: --sp; sp[-1] += *sp;
dispatch body: goto **(pc++);
```

Figure 2.3: Inlined Sequence

```
/* buf - pointer to an inlined sequence */
void *code[] = { buf, ...};
void **pc = code;
int stack[STACKSIZE];
int *sp = stack;

goto **(pc++);
```

Figure 2.4: Inline-threaded Interpreter

The inline-threaded interpreter improves over the direct-threaded interpreter by reducing the dispatch cost from one branch instruction per bytecode instruction to one per sequence of bytecode instructions. Not all instructions can be inlined. Inlinability of an instruction depends on the platform and the compiler version. The inline-threaded interpreter is the least portable of the three. A framework is provided with SableVM to help developers port the inline-threaded interpreter to new platforms.

2.2 Virtual Machine Instruction Set

For performance reasons, SableVM does not interpret pure bytecodes. Instead, an extended bytecode instruction set is used. The main differences are:

- The instruction array is an array of words rather than an array of bytes. The word size matches the natural word size of the native architecture.
- Some constant operands are inlined into the code array instead of being read from the constant pool. This provides a more direct access to the values.
- Several bytecode instructions have been split according to their type. For example, the `getField` bytecode is split into several instructions: `GETFIELD_BOOLEAN`,

GETFIELD_INT, ...

- Bytecode instructions that might trigger class loading or that might require method or field resolution are available in two variants. A slower variant that performs any required preparation work before executing the actual core functionality of the bytecode and a faster variant without any preparation work. The slow variant is patched with the fast variant after it has been executed once. For example, each `getfield` bytecode is further divided into a slow `PREPARE_GETFIELD_type` instruction and a fast `GETFIELD_type` instruction. This is discussed in further details in section 2.3.
- Branch instructions have been split in two variants: a normal version and a `CHECK` version. The check version has a garbage collection check point. The check point consists of saving the current state and checking if the current thread should be stopped for garbage collection. These `CHECK` instructions are used on loop back edges when the loop body does not contain any check points thus ensuring one check point per loop iteration. For example, the `goto` bytecode instruction has the `GOTO` and `GOTO_CHECK` variants.

Throughout this thesis, the terms *interpreter bytecode* and *interpreter code* refer to this extended bytecode set.

2.3 Preparation Sequences

SableVM uses preparation sequences for efficient execution. Preparation sequences are short sequences of code containing the slow variant of instructions requiring preparation work (see section 2.2). These sequences are located at the end of the code array. They allow for longer inlined sequences as most fast variants can be inlined whereas the slow variants cannot. Preparation sequences are also used to solve efficiently the *two-values replacement* problem. In multithreading contexts, replacing two or more values non-atomically creates a potential race condition. This race condition is avoided without additional synchronization code. When executed, preparation

2.3. Preparation Sequences

Before first execution	After first execution
opcode_1: <code>...</code> <code>GOTO</code> <code>&sequence_1</code>	opcode_1: <code>...</code> <code>GETSTATIC_INT</code> <code>&sequence_1 (skipped)</code>
operand_1: <code>NULL</code> next: <code>...</code>	operand_1: <code>pointer to field value</code> next: <code>...</code>
sequence_1: <code>PREPARE_GETSTATIC_INT</code> <code>pointer to field info</code> <code>&operand_1</code> <code>REPLACE</code> <code>&opcode_1</code> <code>GETSTATIC_INT</code> <code>GOTO</code> <code>&next</code>	preparation sequence the same (possible dead code)

Figure 2.5: Preparation Sequence

sequences set all operands of the fast variant in the code array before replacing atomically the jump to the sequence with the faster instruction.

Figure 2.5 illustrates an example adapted from [Gag02]. On the left, we have a sequence of code before its first execution and on the right we have the same code array segment after the preparation sequence has been executed. The code accesses a static integer field. The field access site is located at the location labelled with `opcode_1`.

Before the code is executed, we have a `GOTO` instruction at the site that performs a jump to the preparation sequence labelled with `sequence_1` and located at the end of the code array. Note that space is reserved after the `GOTO` instruction for one operand (labelled with `operand_1`).

The first time the code is executed, the `goto` transfers execution to the preparation sequence. The first instruction of the sequence, `PREPARE_GETSTATIC_INT`, resolves the field reference, performs class initialization, sets a pointer to the field value at the location identified by `operand_1`, and finally executes the actual `getfield` functionality. The next instruction, `REPLACE`, substitutes the `GOTO` instruction located at `opcode_1` with a `GETSTATIC_INT` instruction. Finally, a `GOTO` instruction transfers the control back to the instruction after the `getstatic` site (identified by label `next`). Note that future executions of the `getfield` site at `opcode_1` will use the fast `GETSTATIC_INT` implementation rather than executing the preparation sequence.

Chapter 3

Overview

In this chapter we introduce SableJIT, our retargetable JIT. We start by enumerating the main characteristics of SableJIT. Then we present its three main components: the runtime, the compiler frontend, and the compiler backend. Several topics introduced in this chapter are discussed in greater depth in subsequent chapters.

3.1 Introduction to SableJIT

SableJIT is a *retargetable* just-in-time compiler (JIT) for SableVM. The main objective is to have a JIT that is relatively easy to port or to *retarget* to a new platform whether it is a new operating system or a new CPU architecture. SableJIT is designed to be used in a *mixed mode* environment. That is, the code is first interpreted then, to improve the performance, frequently executed code is compiled at runtime for the native platform. We briefly summarize several key characteristics of SableJIT:

- SableJIT is mostly written in Java with some C code interfacing the compiler with the virtual machine.
- SableJIT is able to fully self-compile.
- SableJIT is fully reentrant. Several compilations may be in progress at the same

time¹.

- SableJIT can be used with any of the three available interpreters.
- SableJIT comprises a baseline compiler and a retargetable backend. An optimizing compiler could be added for improved performance.
- The design of SableJIT favours incremental development and testing. Porting to a new platform may be done incrementally and requires relatively little effort.
- The code generated by SableJIT is fully relocatable.
- SableJIT is robust in that it can recover from some compilation failures. This robustness favours incremental development as unimplemented features or instructions do not crash the virtual machine during unit testing.

3.2 Architecture Overview

The architecture of SableJIT is divided into 3 main components: the runtime, the compiler frontend, and the compiler backend. Figure 3.1 illustrates the main components. The compiler frontend and the compiler backend are written in Java and are isolated from the virtual machine internals. The runtime, mostly written in C with some assembly, is statically compiled within the virtual machine. It provides an interface between the virtual machine and the compiler as well as runtime support for compilation. We now further describe each main component and how they interact with each other.

3.2.1 SableJIT Runtime

The SableJIT runtime provides runtime support for both compilation and execution of compiled code. The tasks related to the compiler and to the compilation process are:

¹As a current limitation, at most one thread may compile code at any time though several compilations may be in progress in this thread.

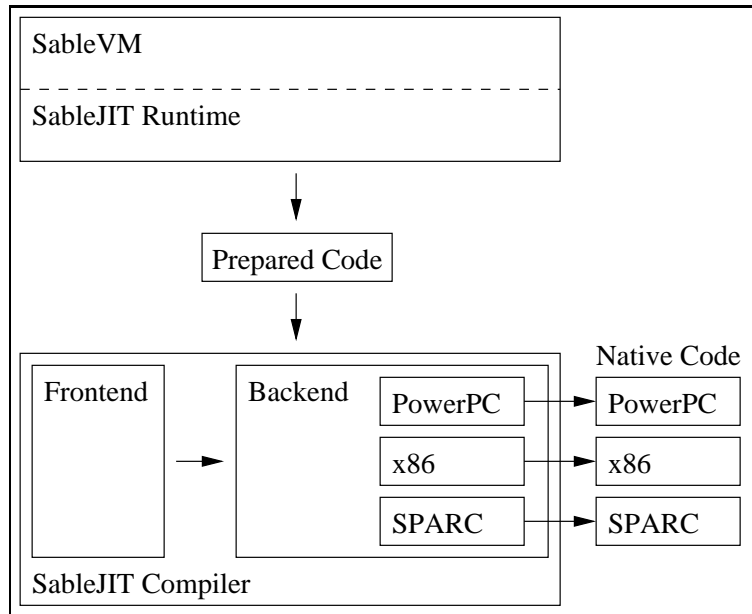


Figure 3.1: SableJIT Architecture

- Bootstrapping the compiler.
- Determining whether a method should be compiled.
- Preparing the interpreter code in a format suitable for compilation.
- Computing data required by the compiler such as exception maps.
- Invoking the compiler.
- On successful compilations, allocating the code in fixed memory.
- On compilation failures, performing a failure recovery procedure.
- Managing memory: allocating and freeing compiled code and associated data.

The tasks related to the execution of compiled code are:

- Providing hooks into the virtual machine.

- Providing stubs to safely and as efficiently as possible switch from dynamically compiled code to interpreted code or to JNI native methods.
- Handling of exceptions originating from compiled code.

Most of these tasks are covered in chapter 4. Exception handling and memory management are covered in chapter 6 and 7 respectively.

3.2.2 Compiler Frontend

The compiler frontend receives the array of code to compile as well as additional data from the SableJIT runtime. The frontend is mainly responsible for the compilation process. The frontend consists presently of a baseline compiler that generates code without optimization in a single pass plus a branch patching phase. Methods from the backend are invoked directly to generate architecture-specific native code. The addition of an optimizing compiler is left as future work. The compiler frontend is architecture independent.

Figure 3.2 shows the main classes involved in the frontend and their relationship as an *unified modelling language* (UML) diagram. The `Compiler` class is the single class interfacing with the runtime. The `OnePassGenerator` class implements the baseline compiler. This implementation of the `IRBuilder` interface is special in that no intermediate representation (IR) data structure is built. Instead, methods in the `Architecture` class located in the backend are directly invoked to generate the native code.

Two outcomes are possible from the compilation. If the compilation completes successfully, the compiled code is returned to the runtime. If compilation fails, the frontend will attempt to restore its state and will signal the runtime. If the failure can be recovered from, the method is marked as *uncompilable* and it is interpreted. Otherwise, the virtual machine is terminated.

The frontend is described in further details in chapter 5.

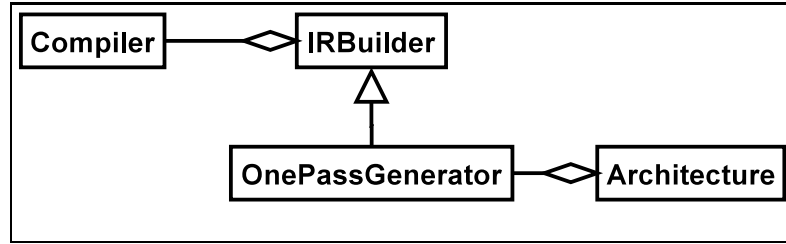


Figure 3.2: Compiler Frontend

3.2.3 Compiler Backend

The compiler backend is responsible for code generation. It consists of both an architecture-independent and an architecture-dependent part. As most functionality has been made as architecture independent as possible, it is relatively easy to retarget the backend to a new platform. Platforms currently supported are x86 (Linux and FreeBSD), PowerPC (Linux and Mac OS X), and SPARC (Solaris).

Figure 3.3 is an UML diagram illustrating the main backend classes and their relationships. The **Architecture** class provides the architecture-independent interface between the compiler frontend and backend. It also performs tasks that can be shared between all architectures such as maintaining the code arrays and providing a framework for branch patching. Each supported processor architecture has a corresponding subclass of **Architecture**. This class provides the implementation of architecture-specific functionality. The *abstract binary interface* (ABI) of a platform specifies the calling conventions and the register usage conventions among other things. It is specific to an operating system. The ABI of each platform is implemented in a separate class that implements the ABI interface. This organization favours retargetability as two operating system on the same processor architecture can differ in their ABI although the native instruction set is the same.

The backend as well as retargetability issues are covered in chapter 8. Chapter 9 presents a porting strategy and our experience in porting SableJIT to the Solaris/sparc platform.

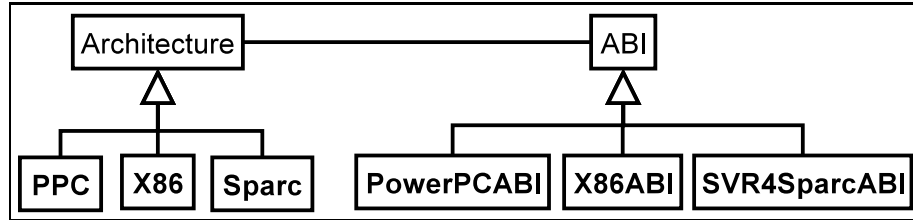


Figure 3.3: Compiler Backend

3.3 Summary

In this chapter we have introduced the different components of SableJIT. In particular, we have seen that SableJIT is composed of three components: the runtime, the compiler frontend, and the compiler backend. The runtime controls the compilation process and provides an execution environment for compiled code. It is the interface between the compiler and the other components of the virtual machine. The compiler frontend receives, from the runtime, the code to compile. It compiles it, making use of the compiler backend to generate architecture-specific native code.

Chapter 4

SableVM/SableJIT Interface

In this chapter we present the interface between the virtual machine and the compiler. We start by describing how the interpreter instruction set is extended in order to reduce compilation overhead and to provide an interpreter-only mode with little overhead. Then we explain how the virtual machine and the compiler are bootstrapped. We follow with a discussion on the constraints on where compilation can occur due to the fact that the implementation of our JIT is in Java. We describe the data required for compilation and how the compiler is invoked. We explain how compiled code, interpreter code, and JNI native methods are dispatched from the various execution contexts. Finally, we study the overhead of method invocations, with a special emphasis on the transitions between execution contexts such as compiled code and interpreter code.

4.1 Method Invocation Bytecode Instructions

The interpreter has four invoke instructions: `INVOKEVIRTUAL`, `INVOKESPECIAL`, `INVOKEINTERFACE` and `INVOKESTATIC`. Each one corresponds to one type of bytecode invocation. In addition to these SableJIT introduces the following instructions to reduce compilation overhead.

- `INVOKEtype_JITCOMPILE_BOOTSTRAP`: These instructions, one for each invocation type, checks if the compiler has been bootstrapped before attempting to

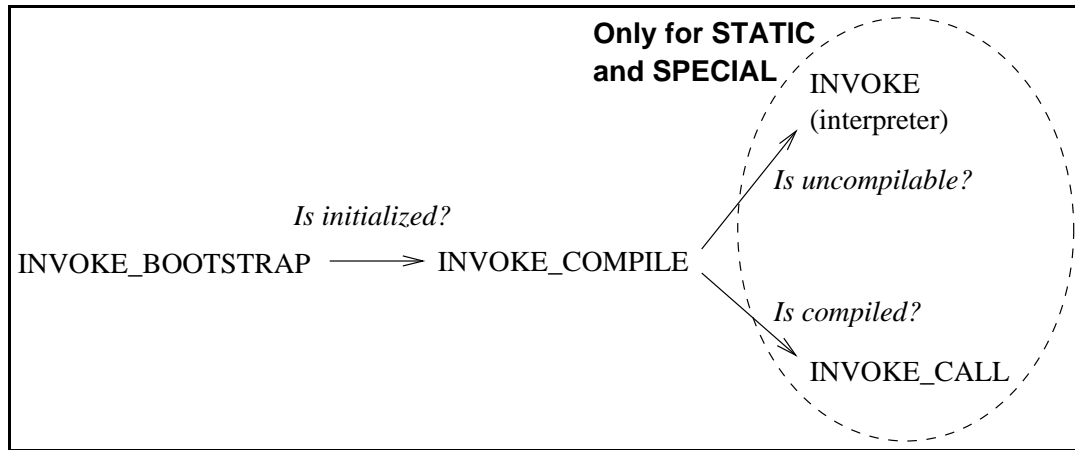


Figure 4.1: Method Invocation Bytecodes and Transitions

compile the callee method. If the compiler has already been bootstrapped, this instruction is replaced with `INVOKEtype_JITCOMPILE` to avoid subsequent checks.

- `INVOKEtype_JITCOMPILE`: These instructions are used once the compiler is known to have been bootstrapped. If the callee has not been compiled, it checks if its code should be sent to the compiler for compilation. This variant is also available for all invocation types.
- `INVOKEtype_CALLCODE`: This variant is available only for special and static invocations. It is used once the callee has been successfully compiled. The compiled code of the callee is invoked without performing any compilation-related checks.

Figure 4.1 illustrates transitions between the variants as execution and compilation proceed. The reason why the “*is uncompileable*” and “*is compiled*” transitions are done only for static and special invocations is that these invocations are always *monomorphic*, that is, the call site has a single target. The method implementation to use (in our case compiled code) is therefore known and unique at such sites. In the next few sections we elaborate on how each invoke instruction is used.

4.2 Interpreter-Only Mode

Even if SableVM is compiled with SableJIT support, it is still possible to run the virtual machine in interpreter-only mode¹. In that mode, the compiler is never used nor bootstrapped.

To achieve an efficient interpreter-only mode, the normal interpreter `INVOKEtype` instructions are used when preparing the interpreter code rather than the `INVOKEtype_JITCOMPILE_BOOTSTRAP` and `INVOKEtype_JITCOMPILE` instructions. No overhead is added to the execution of interpreter bytecodes. However, there is a small extra cost during method preparation as a check is done to select the appropriate invoke variant to use. It is important to have an interpreter-only mode with an overhead as low as possible as it could serve as a basis for comparisons in some experiments.

Experimental results show that the performance penalty is from 0% (i.e. unnoticeable) to 5% with an average of 1-2% with our benchmarks set. The increase in the binary sizes on disk are from 21.3% for the switch interpreter up to 22.2% for the inline-threaded interpreter. Although, the increased binary sizes will cause more memory to be used, the portion of memory actively used remains basically the same as code related to the SableJIT runtime and compiler is never executed. The experimental setup and detailed results can be found in section 10.3.2.

4.3 Bootstrapping the Virtual Machine and Compiler

Our compiler is written in Java. This means that the classes of the compiler cannot be initialized early in the virtual machine start up. Compilation may only be enabled after the virtual machine has been bootstrapped and is ready to run arbitrary Java code.

We first describe the steps involved in bootstrapping the virtual machine. Then we carry on with the different steps required to bootstrap the compiler, what they involve and when they occur.

¹The interpreter-only mode is specified by passing the `-C int` option on the command line.

4.3.1 Bootstrapping the Virtual Machine

As a restriction of SableVM, classes loaded during the bootstrap process cannot have static initializers. The bootstrapping of the virtual machine consists of the following steps.

1. The following types are created, linked and initialized: `java.lang.Object`, `java.io.Serializable`, `java.lang.Cloneable`, byte array, int array², `java.lang.VMClass`, and `java.lang.Class`. `Class` instances corresponding to these types are not created yet.
2. The constructor of `java.lang.Class` is resolved.
3. `Class` instances are created for the classes loaded in step 1.
4. Several other types are now loaded. These include additional types in `java.lang`, primitive array types, exception types, and error types. For these and any future types, `Class` instances are created as they are loaded.
5. Several methods and fields are resolved.
6. Instances of `Class` for primitive types are created.
7. Instances of exception and error classes are created. These are used if such instances cannot be created during exceptions.

SableVM is now ready to execute arbitrary bytecode.

4.3.2 Bootstrapping the Compiler

The SableJIT runtime and compiler are initialized as follows:

1. Initializing the runtime

Several data structures in C are initialized based on default values and any options provided on the command line. This step is performed before bootstrapping the virtual machine. It does not involve any execution of Java code.

²This one is required for SableJIT.

2. Bootstrapping the compiler

The compiler is bootstrapped right after the virtual machine has been bootstrapped. The runtime performs the following steps:

- (a) It loads, links and initializes the `sablejit.SableJITClassLoader` class.
- (b) It creates a single instance of the class loader of the previous step.
- (c) Using the class loader just created, it loads, links and initializes the `sablejit.Compiler` class.
- (d) It creates a single instance of `sablejit.Compiler`.
- (e) It resolves two methods from `Compiler`: `compile` and `compileCompile`.

The compiler may now be enabled.

The singleton³ [GHJV95] in step 2d is the main interface between the runtime and the compiler. The two methods resolved in step 2e are used to compile Java methods. The `compileCompile` method is used only to compile the `compile` method in a transparent way. It has the same prototype as `compile` and simply calls `compile`.

At the end of its bootstrap, the compiler is ready to compile arbitrary⁴ code including itself. Note that it is ready before the application classes are loaded.

Once the compiler has been bootstrapped, `INVOKEtype_JITCOMPILE` instructions are used when the code arrays are prepared instead of the `INVOKEtype_JITCOMPILE_-BOOTSTRAP` instructions. Previously prepared methods will have their `JITCOMPILE_-BOOTSTRAP` instructions patched with the faster version next time these instructions are executed.

4.3.3 The Compiler Class Loader

There are several advantages of using a separate class loader for the compiler classes rather than to simply use the virtual machine bootstrap class loader. First, it allows

³A singleton is a class designed to have a single instance.

⁴Not quite arbitrary, the compiler needs to be temporarily disabled at some specific points. See section 4.4.1.

the separation of the compiler classes from the bootstrap classes in both namespace and in file locations. The compiler classes are bundled together in a jar file. Only the `SableJITClassLoader` class file needs to actually reside in the bootstrap classpath. A second use for a separate class loader is that it provides an interface to the virtual machine internals. Since the `SableJITClassLoader` class is loaded by the bootstrap class loader, native methods located in that class have access to the data structures and functions of the virtual machine while native methods in the other compiler classes do not. This interface to the internals is required to access some configuration information and to perform some basic actions such as stopping and restarting the compiler.

4.4 Invoking the Compiler

We have seen how the compiler is bootstrapped and brought to a state ready for compilation. In next few sections we cover the role of the runtime in the compilation process. We study the constraints on where compilation can take place. We explain how the runtime makes the decision to compile a method. We describe the data computed by the runtime and fed to the compiler. Finally, we explain how the compiler is invoked from the runtime.

4.4.1 Where Compilation May Take Place

Since garbage collection (gc) could be triggered during the compilation of a method, compilation should occur only at points that are gc safe, that is, at points where the required state information has been saved in the event that gc happens.

This restriction is not very limiting as all method invocations (method entry points), that are common compilation entry points, are already gc safe. To enable compilation at other points such as on loop back edges, it is required to ensure that these are gc safe points. This is relatively easy to do. SableVM already uses a variant of branch instructions with gc check points if the loop body does not contain any instruction with such points. Similar instructions could be used to add compilation

entry points on loop back edges. An experimental implementation of compilation on loop back edges has been implemented in SableJIT. It is however not yet stable enough for use.

It is necessary to temporarily disable the compiler when executing particular code segments that could trigger class loading. We list the regions where compilation is disabled with a short justification:

1. User Class Loading

If the current class loader is the SableJIT class loader, the compiler is disabled while executing Java code in the user class loader functions (`_svmf_usercl_create_class` and `_svmf_usercl_create_array`). The Java code comprises both a method to create a `String` argument and the invocation of `VirtualMachine.createClass` or `VirtualMachine.createArray`.

This is done to avoid a circular dependency. Consider the case where we are currently compiling some method m . During compilation, a SableJIT class X could be loaded. If compilation was enabled during that loading, the compiler could be invoked again and this would trigger the loading of the same class X . A circular dependency is created: in order to load class X , X would have to have been already loaded, as it is needed to compile methods invoked during the loading and initialization of X .

2. Class Initialization

The compiler is disabled during the initialization⁵ of a class if the class loader of that class is the SableJIT class loader.

This is done to avoid executing code in classes that have not been yet fully initialized. As the compiler can be invoked within static initializers, these implicit⁶ method invocations could change the compiler source code semantics.

These two cases were obtained as follows. Compilation was first avoided for large regions then these regions were shrunk by using some reasoning and by trial and

⁵More precisely, while executing `Class.initialize(int)`.

⁶JIT compiler method invocations are not explicit in the source code.

error. We leave as future work the study whether these restrictions could be entirely removed or at least alleviated.

Experimental results show that these restrictions do not have a major impact. For our benchmarks set⁷, we found that a total of about 16.6 million bytecode instructions are interpreted while our compiler is temporarily disabled within these regions. This number is relatively constant through all our benchmarks. We also measured the total number of bytecode instructions interpreted when the benchmarks are run in interpreter-only mode. We found that the regions where the compiler is temporarily disabled account from 0.13% up to 1.82% with an average of 0.71% of the total bytecode instructions executed.

4.4.2 When Compilation Occurs

Once bootstrapped, the compiler is ready to compile methods. It is not absolutely necessary for a method to be first interpreted, although it is recommended. The current default is to compile a method just before its second invocation. The number of times a method is interpreted before being considered as a *hot method*, that is, a method frequently executed, can be specified on the command line.

4.4.3 Data Required for Compilation

In this section, we present the data required for compilation. All data necessary for compilation is passed as arguments to the compiler. Figure 4.2 shows the prototype of the `compile` method. We summarize the arguments below and we describe some of them in more details later.

- `bytecode`: The code to compile.
- `methodName`: The name of the method we are compiling. This is used for debugging and testing purposes.

⁷See section 10.2 for a description of our benchmarks set.

4.4. Invoking the Compiler

```
public Object compile(int[] bytecode,
                    String methodName,
                    int[] stackOffsets,
                    boolean isSynchronized,
                    int startPC,
                    int[] bpc2inlinedpc,
                    int bytecodeSize,
                    boolean isStatic,
                    int preparedCodePointer,
                    int preparedCodePointerCurrentPC,
                    int classInstancePointer,
                    int thisNbArgs,
                    int methodInfoPointer,
                    int nonParamRefLocalsCount,
                    int[] exceptionHandlerOffsets
                    ) throws Exception
```

Figure 4.2: Prototype of `Compiler.compile`

- **stackOffsets**: The Java operand stack height at the start of each bytecode instruction.
- **isSynchronized**: A flag that indicates if we are compiling a synchronized method.
- **startPC**: The address of a second entry point in the code. This is used to switch from the interpreter code to compiled code within a method body.
- **bpc2inlinedpc**: A (*switch bytecode pc*⁸ \rightarrow *inlined pc*) mapping. This is used only with the inline-threaded interpreter for exception handling among other things.
- **bytecodeSize**: This is the size of the normal part of the array, that is, it excludes the preparation sequences at the end.
- **isStatic**: A flag that indicates if we are compiling a static method.
- **preparedCodePointer**: The address of the C array corresponding to the code we are compiling. It is used to translate absolute addresses in the code array to relative addresses.

⁸It is actually an index into the switch code array, not an absolute address *pc* as used in the interpreter.

4.4. Invoking the Compiler

- **preparedCodePointerCurrentPC:** The address of the code array used by the interpreter. This is used to compute the current interpreter PC.
- **classInstancePointer:** An indirect pointer to the `Class` instance of the declaring class of the method we are compiling.
- **thisNbArgs:** The number of arguments that the method we are compiling is taking.
- **methodInfoPointer:** A pointer to the C data structure of the method we are compiling.
- **nonParamRefLocalsCount:** The number of local variables of reference type that are not parameters.
- **exceptionHandlerOffsets:** An array where each element is the starting offset of an exception handler in the code array. Offset i corresponds to the beginning of the i^{th} handler.

The code fed to the compiler is the interpreter code (as described in section 2.2). We use this code rather than pure bytecode to take advantage of information already computed by the virtual machine. Some of this information is computed at method preparation. It includes additional type information for some instructions (such as the type of field operations) as well as information used for garbage collection and exception handling. Some information is also obtained from the code array at a later time once the code has been interpreted. As seen in section 2.3 some slower variants of instructions performing any required preparation work such as class loading, method resolution and field resolution are patched to faster variants. The compiler takes advantage of this information as it does not need to generate code to perform this preparation work for instructions having already been executed.

<pre> static int foo(int n) { if (n < 0) { return 100; } else { return n; } } </pre>	<pre> 0 1 2 3 4 5 6 7 L1: </pre>	<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <tr><td style="padding: 2px;">&&ILOAD_0</td></tr> <tr><td style="padding: 2px;">&&IFGE</td></tr> <tr><td style="padding: 2px;">L1</td></tr> <tr><td style="padding: 2px;">&&LDC_INT</td></tr> <tr><td style="padding: 2px;">100</td></tr> <tr><td style="padding: 2px;">&&IRETURN</td></tr> <tr><td style="padding: 2px;">&&ILOAD_0</td></tr> <tr><td style="padding: 2px;">&&IRETURN</td></tr> </table>	&&ILOAD_0	&&IFGE	L1	&&LDC_INT	100	&&IRETURN	&&ILOAD_0	&&IRETURN	<pre> 0 1 2 3 4 5 6 7 L2: </pre>	<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <tr><td style="padding: 2px;">ILOAD_0</td></tr> <tr><td style="padding: 2px;">IFGE</td></tr> <tr><td style="padding: 2px;">L2</td></tr> <tr><td style="padding: 2px;">LDC_INT</td></tr> <tr><td style="padding: 2px;">100</td></tr> <tr><td style="padding: 2px;">IRETURN</td></tr> <tr><td style="padding: 2px;">ILOAD_0</td></tr> <tr><td style="padding: 2px;">IRETURN</td></tr> </table>	ILOAD_0	IFGE	L2	LDC_INT	100	IRETURN	ILOAD_0	IRETURN
&&ILOAD_0																				
&&IFGE																				
L1																				
&&LDC_INT																				
100																				
&&IRETURN																				
&&ILOAD_0																				
&&IRETURN																				
ILOAD_0																				
IFGE																				
L2																				
LDC_INT																				
100																				
IRETURN																				
ILOAD_0																				
IRETURN																				
(a) source code	(b) direct-threaded code	(c) switch code																		

Figure 4.3: Direct-Threaded Code to Switch Code Conversion

With the switch interpreter, the interpreter code is simply copied to a Java `int []` array⁹ before being passed on to the compiler. For the direct-threaded and the inline-threaded interpreter, we would also like to take advantage of the information obtained by previously interpreting the code. However, direct-threaded code and inline-threaded code are not convenient inputs to the compiler as the code arrays contain addresses to implementations of instructions rather than instruction integer opcodes. For these two interpreters, we therefore compute an equivalent switch code array. Note that by doing so, the compiler is mostly the same for all three interpreters. We now describe how the conversion to switch code is done.

Direct-JIT Mode

The direct-threaded code differs from the switch code in that it uses implementation addresses for instructions rather than integer opcodes. The format of the code array as well as the instructions operands remain the same. Figure 4.3 illustrates the conversion from direct-threaded to switch code. In figure 4.3(a) we have the source code of a small method. Figure 4.3(b) and 4.3(c) illustrate the corresponding direct-threaded and switch code respectively. Note that as explained, the only difference is in the representation of the instructions opcodes.

⁹SableJIT currently supports only 32-bit architectures and it assumes that the code array elements are 32-bit wide.

4.4. Invoking the Compiler

To convert efficiently direct-threaded code to switch code a splay tree data structure¹⁰ is precomputed when the runtime is initialized. Each node in this tree stores the relationship between an implementation address and its corresponding integer opcode.

Converting the direct-threaded code of a method then involves the following tasks:

1. For instructions, convert the implementation address to the corresponding integer opcode using the precomputed splay tree.
2. For address operands referring to code array elements, copy an adjusted address. Address operands included in this category are all branch targets, jump table target entries and patching addresses in preparation sequences.
3. For all other operands, copy them verbatim.

The adjustment in the second task is required as these address operands are absolute memory addresses. We want the addresses in the corresponding switch code array to point to elements in that array and not to the original direct-threaded code array. Since switch code and direct-threaded code have the same layout, the adjustment consists of simply adding a constant offset to each operand involved to reflect the memory location of the new code.

The conversion process is done once. The code is kept for future recompilations. If compilation fails or the code is not subjected for future compilation, then it is discarded.

The conversion process is quite efficient. A single pass through the direct-threaded code array is sufficient. A lookup in the splay tree takes $O(\log M)$ amortized time where M is the size of the tree. The overall runtime is $O(n \log M)$ amortized where n correspond to the code array size. Note that the size of the tree M corresponds to

¹⁰A splay tree data structure was chosen to favour code reuse. This data structure is also used for the inline-threaded code to switch code conversion where it is more suitable. The inline-threaded conversion is discussed later in this section. Also, a reusable implementation of a splay tree was readily available in SableVM as it is used by the virtual machine for various tasks. For the direct-threaded code conversion, a simpler data structure such as a hash-table could have performed as well, if not better by a constant factor.

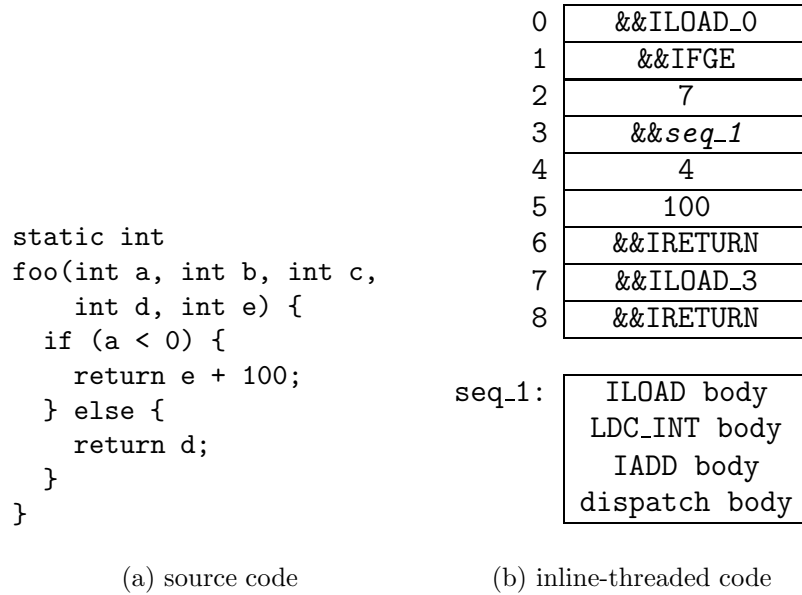


Figure 4.4: Inline-Threaded Code

the number of interpreter instructions. This number is constant and is less than 400. The overall running time is therefore $O(n)$.

Experimental results¹¹ show that the direct-threaded to switch conversion accounts from 0.48% to 1.46% of the compilation time, with an average of 1.00%.

Inlined-JIT Mode

The inline-threaded interpreter takes the direct-threaded interpreter one step further by using the addresses of implementations of sequences of instructions. Figure 4.4 illustrates a small method and its corresponding inline-threaded code. It will serve as an example throughout this discussion.

The inline-threaded code array contains three types of elements: addresses of instruction implementations (such as `&&ILOAD_0`), addresses of sequence implementations (such as `&&seq_1`), and operands of instructions (7, 4, and 100). Implementation addresses of instructions work as in direct-threaded code. Sequences consist of the

¹¹See chapter 10 for a description of our experimental setup. These experiments were conducted on our Linux/x86 platform.

implementation of several instructions copied consecutively in memory. The code in our example has a single sequence composed of the implementations of `ILOAD`, `LDC_INT`, and `IADD`. Dispatch code is located at the end of the sequence to transfer the control to the next instruction or sequence in the code array. There is no dispatch code between the implementations of instructions. Note that the operands of all instructions in a sequence are located consecutively in the code array. In our example, the consecutive integers 4 and 100 correspond to the operands of the `ILOAD` and `LDC_INT` instructions, respectively.

The conversion process of inline-threaded code is more complex than direct-threaded code. Sequences must be replaced by their individual instructions. These instructions must be inserted in the resulting code array with their operands in-between. Additional complexity arises from the fact that preparation sequences in the inline-threaded code might have more instructions than the (`PREPARE_instruction`, `REPLACE`, `GOTO`) triplet found in switch code. These differences make the layout of inlined-threaded code quite different from the layout of switch code. Address operands cannot be simply *adjusted* by adding a constant offset as is the case for the direct-threaded code conversion. Instead, a second pass through the code is required to patch them.

In addition to computing the switch equivalent code, a (*switch pc* \rightarrow *inlined pc*) mapping needs to be computed and passed to the compiler for exception handling. This same data structure is also used in the conversion process to patch the address operands.

The actual implementation of the conversion has quite a lot of technical details. We limit the presentation to an outline of our algorithm rather than describing every possible case at each step.

Precomputed Data As is the case for the direct-threaded conversion, a splay tree is used to map addresses to instructions. It stores the mapping (*address* \rightarrow $\{instruction, sequence\}$) where the value is either an instruction integer opcode or a pointer to the corresponding sequence data structure. This data structure contains the list of instruction integer opcodes making up the sequence. The splay tree is

computed when the runtime is initialized. It is updated as new inlined sequences are created during method preparation.

Conversion The conversion of inline-threaded code is done in two passes. In the first pass, the resulting switch code array is filled with the equivalent instruction opcodes and their operands. The second pass fixes some memory address operands.

To translate an implementation address, a lookup is done in the splay tree. The node in the tree indicates whether it contains the opcode of a single instruction or if it contains a pointer to a sequence data structure. In both cases, the instruction opcode(s) and their operands are copied to the right locations in the switch code array. In particular, operands of several instructions in a sequence that appear consecutively in the inline-threaded code array will, in the switch code array, appear after their associated instruction.

Figure 4.5 illustrates the conversion to switch code. The resulting switch code array at various intermediate steps is shown. In figure 4.5(a), the `ILOAD_0` integer opcode has been copied. In figure 4.5(b), the `&&IFGE` address has been processed by writing the integer opcode and the address operand verbatim. Note that in the actual implementation, address operands are absolute memory addresses pointing to some element in the code array. We use relative addresses in our example for simplicity. Addresses pointing to the inline-threaded code will be fixed in the second pass. In figure 4.5(c), the first instruction of the sequence `seq_1`, `ILOAD`, has been copied. In figure 4.5(d), the second instruction of the sequence has been processed. Note that the 4 and 100 operands do not appear consecutively in the switch code. In figure 4.5(e), the conversion of the sequence has been completed. Figure 4.5(f) illustrates the switch code array after the first pass. Note that the address operand of the `IFGE` instruction is still incorrect. The second pass is now performed. Figure 4.5(g) shows the resulting array at the end of the conversion. All address operands have been patched as necessary.

The first step of the conversion can be performed in a single pass over the code. However, it is likely to be less efficient than the direct-threaded code conversion by a constant factor since most operations are not as straightforward and several cases

4.4. Invoking the Compiler

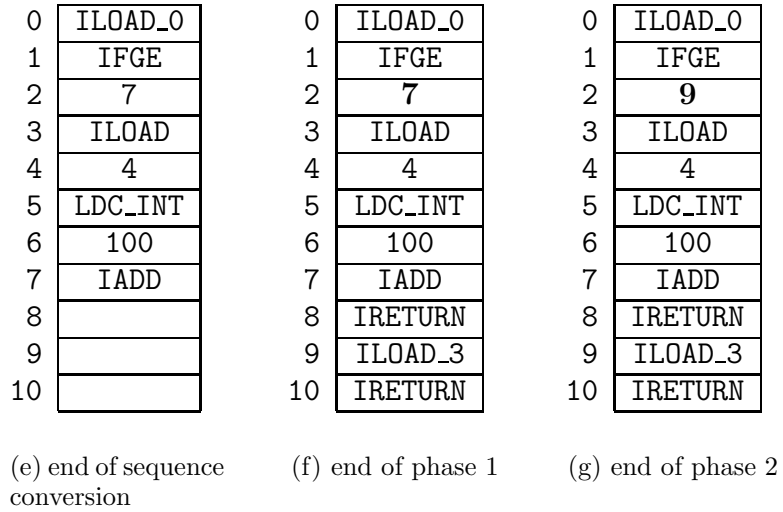
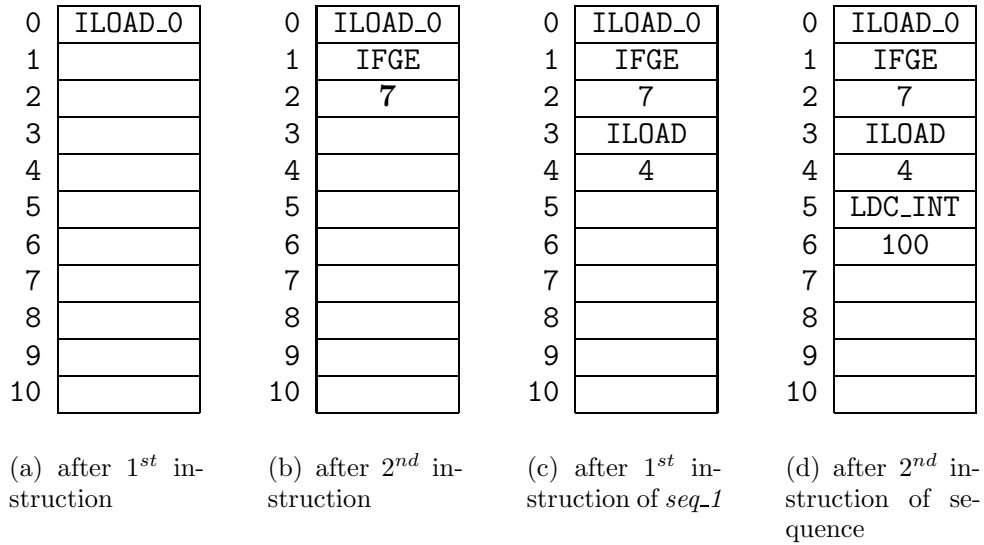


Figure 4.5: Inline-Threaded Code to Switch Code Conversion

need to be checked and handled differently. During that first pass, the `bpc2inlinedpc` code array is computed as well. It will be used in the second pass.

A second pass is done through the switch code array generated in the first pass to fix address operands pointing to elements in the original inline-threaded code array. The `bpc2inlinedpc` array is used to map these addresses to the corresponding elements in the switch code array. The value of the i^{th} element in `bpc2inlinedpc` is the inline-threaded PC corresponding to the i^{th} element in the switch code array. Since the values in the array are in non-decreasing order, a binary search is used to find a particular address. The array index of the address found indicates the corresponding address in the switch code.

This marks the end of the conversion process. The conversion of a method is done once and kept if needed later for recompilation.

We define the input size n as the sum of the size of the inline-threaded array and all instructions in sequences used in the array. The first step is performed in $O(n)$ worst case time. The resulting switch code array size is $O(n)$. In the second step, each binary search is done in $O(\log n)$. To fix all address operands requires $O(n \log n)$ time. The overall running time for the conversion is therefore $O(n \log n)$.

Experimental results¹² show that the inline-threaded to switch conversion accounts from 0.48% to 1.62% of the compilation time, with an average of 1.15%.

4.4.4 Invoking the Compiler

Once all data necessary for compilation has been computed, the runtime invokes the `compile` method located in `sablejit.Compiler` and passes to it all the data as arguments. If the method is successfully compiled, the compiled code is returned to the runtime. Otherwise, the runtime will handle the error accordingly. Error recovery is discussed in section 5.4.

¹²See chapter 10 for a description of our experimental setup. These experiments were conducted on our Linux/x86 platform.

4.5 Invoking Compiled Code from the Interpreter

All the `INVOKE $type$ _JITCOMPILE_BOOTSTRAP`, `INVOKE $type$ _JITCOMPILE` and `INVOKE $type$ _CALLCODE` variants introduced earlier in this chapter are able to call the compiled code of the callee. Most steps involved in the preparation of the calling context proceed as usual as both the interpreter code and the compiled code share the same Java stack. This makes the transition from interpreted code to compiled code relatively easy and efficient.

For virtual and interface invocations, the actual method is computed by obtaining a pointer to the method data structure with a lookup in the virtual table. For special and static invocations, this pointer is one operand of the instruction. This data structure contains method-related information. One such piece of information is the `iscompiled` flag. This flag indicates whether the method has already been compiled or not. The compiled code, if any, is accessible through a function pointer stored in the member `compiled_code` of the method data structure. If the method does not have compiled code, the invocation proceeds as usual and the method is interpreted. If it does have compiled code, the callee stack frame is built as usual. However, some tasks that have been moved from the caller to the callee such as method synchronization are not performed in the caller. The compiled code is called as follows:

```
stack_inc = method->compiled_code(env, locals, stack);
```

where `env` is an environment pointer, `locals` is a pointer to the callee local variables and `stack` is a pointer to the operand stack (see figure 4.6). The `env` pointer is mostly used for calling hooks in the virtual machine and for manipulating the Java stack. The `locals` and `stack` pointers could be computed from the `env` pointer but it is more convenient and probably more efficient to send them as arguments as they are readily available in the caller. The value returned and stored in `stack_inc` represents the net effect that the return value of the callee method has on the the operand stack of the caller. It can take a value of 0 (for `void`), 1 (for most types) and 2 (for `long` and `double`). This value is used to update the stack size variable in the interpreter. The actual return value of the Java method, if any, is pushed on the operand stack

4.5. Invoking Compiled Code from the Interpreter

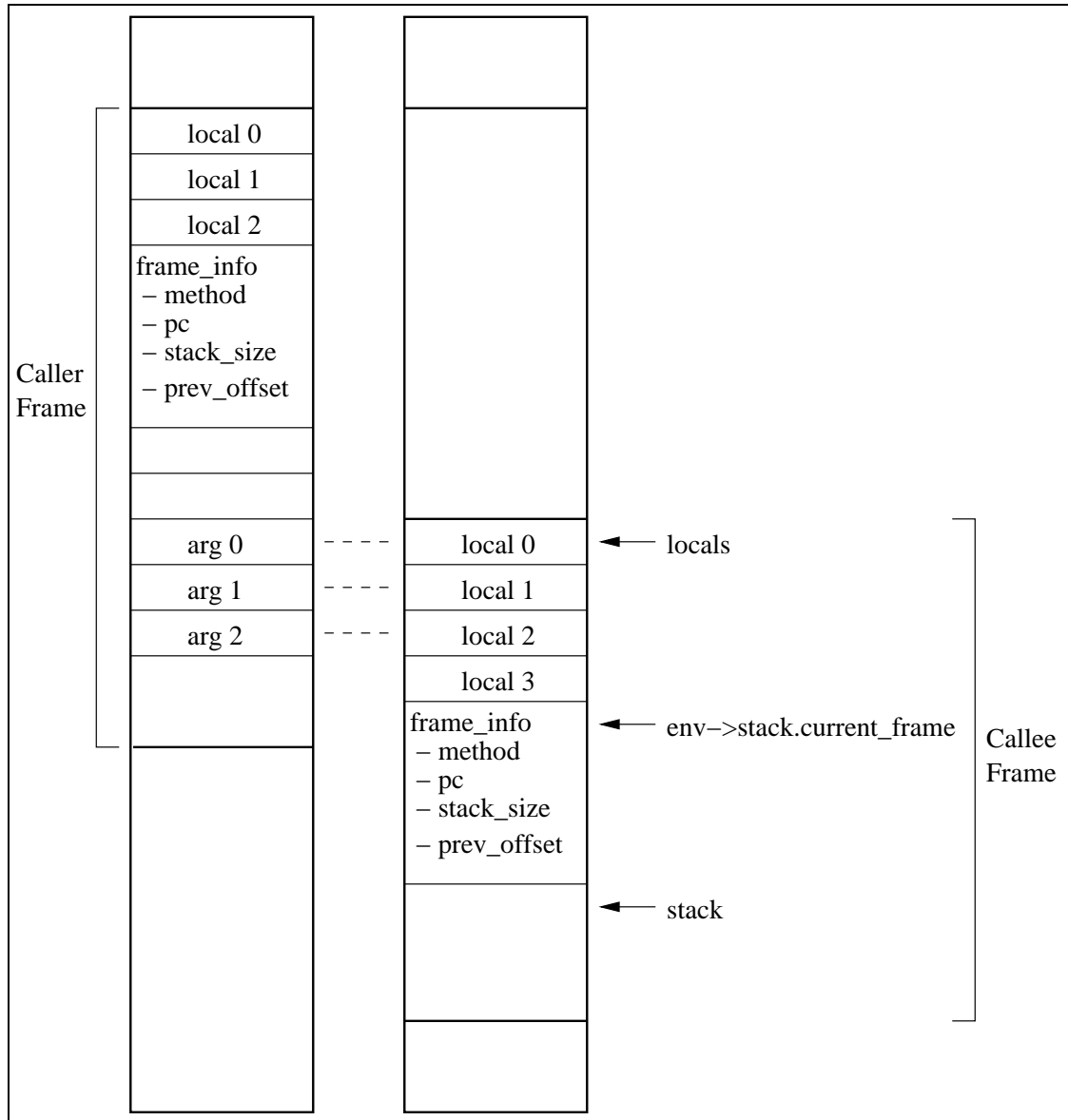


Figure 4.6: Java Stack

by the compiled code. If `stack_inc` has the special value of `-1`, it indicates that the method exited abruptly due to an uncaught exception.

4.6 Method Invocation in Compiled Code

Method invocations in compiled code are always handled in the same way by the caller no matter if the callee is compiled or not. The stack frame is pushed and the function pointer `compiled_code` of the callee is called. No tests are done to see if the callee has compiled code. The runtime makes sure that `compiled_code` has a valid value at all time. Its value is a pointer to either compiled code or one of several stub functions in the runtime. In the next sections we cover how non-compiled callees are handled as well as JNI native calls. Then, we discuss the overhead of the various method invocations.

4.6.1 Interpreted Methods

As mentioned in section 4.6, the `compiled_code` of the callee is always called. It might be the case that the callee has not been compiled yet. In that case, the interpreter should be invoked unless it is decided that the callee should be compiled before its execution. In any case, when method data structures are initialized, the `compiled_code` field is set to point to an interpreter stub function. This function, written in C, has the exact same prototype as the compiled code function. That is, it takes the same three arguments (`env`, `locals` and `stack`) and it returns a value (ranging from `-1` to `2`) with the same meaning. From the caller point of view, it behaves in exactly the same way as compiled code. Figure 4.7 illustrates the various values that the `compiled_code` pointer accepts. It may either point to dynamically allocated compiled code or to one of the stubs compiled statically with the runtime. The interpreter stub is used for methods to interpret. The two other stubs are used for JNI native calls and are discussed in section 4.6.2.

The interpreter stub is structured as follows. It contains a compilation entry point. That is, it checks whether it should compile the callee method, and if yes,

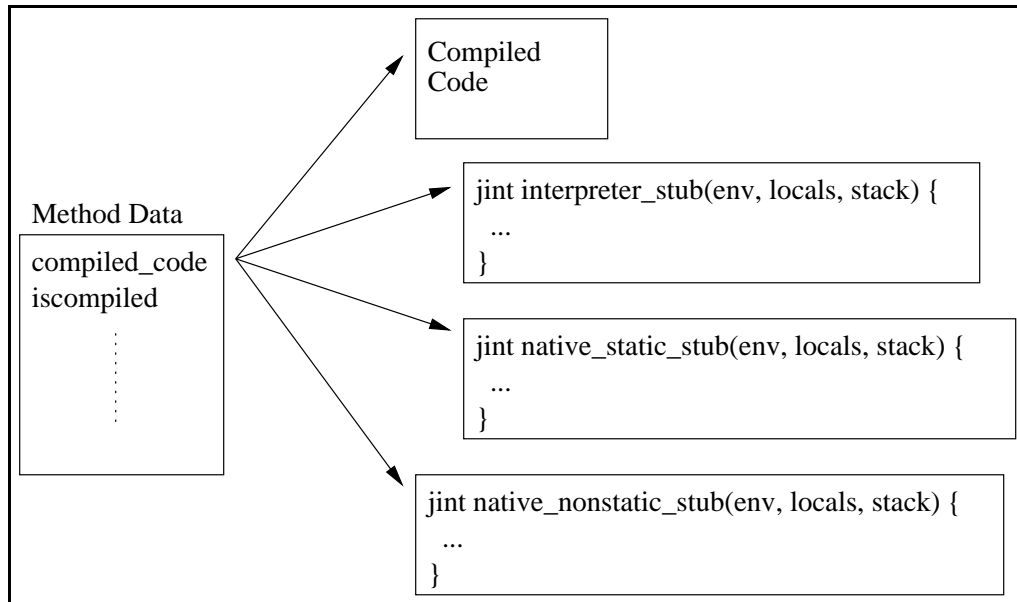


Figure 4.7: Stubs for Compiled Code

the compiler is invoked. The second major component is a control transfer point. If the callee method is compiled (either by the previous compilation entry point or by some other thread), it jumps to the compiled code. Otherwise, the interpreter function is called to interpret the method. In both cases, a meaningful return value, usually depending on the result of the execution of the callee, is returned to the caller. Note that when a method is successfully compiled, the `compiled_code` pointer is updated to point to the compiled code such that the stub will not be called for future invocations of that method. This summarizes the basic roles of the interpreter stub.

The stub, acting as glue code between the interpreted and the compiled world, must also do some adjustments to the calling contexts. For example, in interpreter code, method synchronization, when necessary, is done in the caller whereas in compiled code it is done in the callee. Transitions between compiled code to interpreter code and vice-versa need to take into account these differences to avoid doing the work twice or not at all. The reasons behind some of these changes are related to efficiency and are discussed in chapter 5.

A second required adjustment in the stub consists of inserting an internal call frame on the Java stack between the caller and the callee stack frame. This special frame ensures that execution control exits the interpreter function once the interpreted method exits therefore returning the control to the stub. Note that when the stub is called, the callee frame has already been pushed by the caller. It is simply moved to make room for the additional frame.

4.6.2 JNI Native Methods

In Java, some methods can be declared as *native*. These methods are usually written in the C or C++ language. They are compiled into native shared libraries. The virtual machine will then load the libraries and link the methods during program execution. Although, native methods are not written in Java, SableJIT does not currently have the ability to invoke them directly from compiled code. To call these native methods, it is necessary to copy Java arguments located on the Java stack in registers and/or on the native stack, transfer control to the native function and retrieve the return value translating it back to a Java argument and storing it on the Java stack. This glue code required for both the interpreter and compiled code is implemented in the third party `libffi` shared library. Although, SableJIT did not duplicate this functionality and specialize it for each native method invocations, it still avoids invoking the interpreter to handle them. Two stubs are used, one for static native methods and one for non-static native methods. Their implementation consists simply of calling the function in the virtual machine that handles native methods. It does not require an internal call frame or any other special setup. It does however need to execute synchronization code if the native method is declared to be `synchronized`. Though, this is not additional work, it was simply not done by the compiled caller.

The `compiled_code` field is set to the stub only once the native method has been linked. Linkage occurs at the first invocation and it is always done by the interpreter. In summary, the `compiled_code` pointer is always set to the interpreter stub when method data structures are initialized. If the method turns out to be native, it is

replaced by one of the two native method stubs after it is linked.

4.6.3 Method Invocation Overhead

Additional costs can be incurred during the transitions between different types of method code: interpreter code, compiled code, and JNI native code. A compiled method invoking a compiled method does not incur additional overhead compared to an interpreter invocation. Invoking a compiled method from the interpreter is also lightweight. The overhead consists mostly of additional checks such as testing if the callee is compiled. However, invoking an interpreted method from a compiled method has quite some overhead. It includes the overhead of calling a stub as well as the work inside the stub such as inserting a special stack frame. Therefore, if the caller is compiled, it is good to compile the callees that are frequently invoked. Invoking JNI native methods from compiled code should not be more expensive than invoking them from the interpreted code. However, these invocations could be improved in the future by bypassing the `libffi` general code and by generating specialized glue code at each call sites since the native method prototype is known at compilation time.

4.7 Summary

In this chapter we have presented the SableJIT runtime: the interface between the virtual machine and the interpreter. We have introduced interpreter instructions used for the compilation and execution of compiled code. We have then seen how these instructions are used to provide an efficient interpreter-only mode. We have presented how the virtual machine and the compiler are bootstrapped. In particular, we have covered issues that arise from the fact that the compiler is written in Java and is allowed to self-compile. We have discussed how the code and data are prepared by the runtime before being fed to the compiler. We have described the dispatch of various code forms: interpreter code, compiled code, and JNI native code. Finally, we briefly presented the overhead of transitions between the various code forms.

Chapter 5

The Baseline Compiler

In this chapter we discuss the compiler frontend. We start by providing an overview of the compilation process including the general structure of generated code as well as the main frontend components involved. We then proceed by describing how some runtime information is used in code generation. We discuss how an optimizing compiler could be added to our implementation. We explain how the compiler is able to recover from different kind of failures that could occur and how it helps development. Finally, we describe briefly how partial method compilation and multiple compilation entry points have been implemented experimentally in SableJIT.

5.1 A Simple Baseline Compiler

5.1.1 General Overview

We implemented a *baseline* compiler for SableJIT. This baseline compiler, as opposed to an optimizing compiler, does not perform any optimization. Instead, it generates code directly, without using an intermediate data structure, through a single pass followed by a patching phase. The patching phase is required to patch forward branches as no information is precomputed for final native code instruction locations. The code generated by our compiler mimics the interpreter. Both the compiled code and the interpreted code share the same Java stack. The Java stack includes Java method

stack frames. The Java local variables and the operand stack of the corresponding method are located within the frames. Sharing the same Java stack simplifies the implementation significantly, especially for garbage collection¹ and exception handling². Also, transitions from interpreted code to compiled code and vice-versa can be relatively easily accomplished at points other than method entry and exit. Individual interpreter bytecode instructions are compiled using a template scheme: for each bytecode a set of instructions implementing that bytecode is generated in the code array. We could almost say that the baseline compiler takes the inline-threaded³ interpreter one step further by removing the remaining interpreter dispatch overhead. The baseline compiler, however, does not offer competitive performance compared to optimizing JITs.

5.1.2 Main Components of the Baseline Compiler Frontend

The compiler frontend consists mainly of two classes: `sablejit.Compiler` and `sablejit.OnePassGenerator`. The `sablejit.Compiler` class interfaces with the runtime. It receives the method code from the runtime as well as additional information required for compilation as discussed in chapter 4. The `OnePassGenerator` class implements the `IRBuilder` interface. In fact, the `OnePassGenerator` is a special case of an `IRBuilder` in that no intermediate representation (IR) data structure is built; the code is directly generated. The `IRBuilder` interface declares a series of methods named with the pattern `build_inst`, each one corresponding, with few exceptions, to an interpreter code instruction. Figure 5.1 lists the source code of a possible implementation of the `build_iaload` method that corresponds to the `ILOAD` (*integer array load*) instruction. This particular baseline implementation is located in the `OnePassGenerator` class. It basically generates code by directly invoking methods from the backend rather than building an IR. The `ILOAD` instruction pops an array index and an array reference from the Java operand stack (the code generated by lines 1 and 2 accomplishes this). It then performs a null check on the array reference and

¹Memory management including garbage collection is covered in chapter 7

²Java exception handling is covered in chapter 6

³For an overview of the different interpreters and how they differ, refer to section 2.1

```
// arrayref index --> value
public void build_iaload(int currentPC) {

1  m4sj_popI(tmp2); // index
2  m4sj_popI(tmp1); // ref

3  checkNullPointerException(currentPC, tmp1);
4  checkArrayIndexOutOfBoundsException(currentPC, tmp1, tmp2);

   // multiply index by 4
5  arch.shlI_rrn(tmp2, tmp2, 2);
   // find element
6  arch.addI(tmp1, tmp1, tmp2);
   // load element
7  arch.loadI(tmp1, ALIGNED_ARRAY_INSTANCE_SIZE, tmp1);

   // push it on stack
8  m4sj_pushI(tmp1);
}
```

Figure 5.1: Implementation of `build_iaload`

an array bounds check. Exceptions are thrown if the array reference is null or if the array index is outside array bounds (see lines 3 and 4). The element address is then computed and the value located at that address loaded (see lines 5 to 7). Finally, the value just loaded is pushed on the operand stack (see line 8). We delay the details of how individual instructions are used until we discuss the retargetable backend in chapter 8.

The `Compiler` class consists mainly of a loop iterating over the interpreter bytecode array and invoking a `build_inst` method for each corresponding `inst` instruction. Instruction operands are usually passed verbatim to the `build_inst`. Exceptions include branch target addresses that are converted from an absolute address to an index into the input code array.

It is important to note that the implementation in both the `Compiler` class and the `OnePassGenerator` class is architecture independent and is common for all supported architectures.

Push callee frame Save callee-saved registers	Prologue
Move args to saved regs (env, locals, stack) Enter monitor (if method is synchronized) Set local variables to NULL	Java-Related Actions
[Code for each bytecode instruction]	Method Body
Handle exception	Exception Handling Code (only if some exceptions handlers)
Restore callee-saved registers Pop native frame and return	Epilogue
[Same code as prologue, plus:] Move args to saved regs Jump to 2 nd entry point	Second Prologue (optional)
Map for signal-based exceptions	Data (no code)

Figure 5.2: Compiled Code Organization

5.1.3 The Compilation Process

We describe the steps involved in the code generation of a method. Figure 5.2 illustrates the general logical structure of the code generated by the baseline compiler.

Compilation starts by generating the function prologue. The actual tasks done by the prologue is architecture dependent. In general, a stack frame is allocated, necessary linking is performed (such as saving the return address) and sufficient space is reserved in the instruction stream for *store* instructions to save the callee-saved registers. As in VCODE [Eng96], the baseline compiler uses a pessimistic approach: it assumes that all callee-saved registers that could be used will be used. It allocates enough space on the stack frame to store all the callee-saved registers that could be potentially used.

Before actually compiling the method body, some additional code is generated. This includes code to save into callee-saved registers the three arguments passed to the compiled code when it is called: the `env` pointer, the `locals` pointer and the `stack` pointer⁴. For synchronized methods, additional code includes code to acquire a lock. For methods with local variables (excluding parameters) of reference

⁴As we have seen in section 4.5, `locals` is a pointer to the Java local variables, `stack` is a pointer to the Java operand stack, and `env` is a pointer used to access the data of the current thread.

type, code is generated to initialize each such variable with a `null` value. This is a requirement imposed by the current implementation of the garbage collector. Method synchronization and reference local variables handling are discussed in further details in section 5.2.

The code implementing each interpreter bytecode instruction is consecutively generated after the function prologue and the additional setup code. For some instructions such as forward branches, the actual code is only emitted at the end of code generation. Sufficient space is reserved in the generated code array for them.

Once the code of the method body has been generated, it is patched for forward branches⁵. This first patching phase only handles branches that are at the bytecode level. A second patching phase is performed later for branches added internally, by the implementation of particular bytecodes for example.

If the method has at least one exception handler, then general exception handling code is generated along with a jump table of exception handler offsets.

Then the code generation is finalized. The function epilogue is generated. The actual content of the epilogue is architecture-specific. It normally includes code for restoring any used callee-saved registers, popping the stack frame, and returning from the function. The second patching phase is performed to patch branches within bytecode implementations or added by the compiler such as branches from method exit points to the epilogue. Also, the prologue is patched with *store* instructions for used callee-saved registers.

A second prologue can optionally be generated after the epilogue. This second prologue is used as a second entry point into the compiled code. This second entry is useful to switch to the compiled version of a method being interpreted without first having to exit it and later reenter it. The code of this second prologue consists of the code of the first prologue and of code to move the arguments received into registers. A jump is located at the end of the second prologue to transfer the control to the second entry point in the method body section, the head of a loop for example. To use that second entry point, the runtime executes the second prologue instead of the

⁵In this section, we use the term patching branches rather generally. In addition to branch instructions, we also include targets found in the various jump tables.

usual first prologue. This feature has been implemented experimentally. More details can be found in section 5.5.

This marks the end of actual code in terms of instructions. However, in order to simplify memory management⁶, some data might be allocated in the generated code array after the instructions. In particular, if the compiler is configured to use signal-based exceptions⁷, a table containing native program counter (pc) to bytecode pc mapping information is generated.

5.1.4 Secondary Data Structures

Although no intermediate data structure is built for the code, a number of auxiliary data structures are built during the compilation process which are used in addition to the information received from the runtime. We briefly describe these data structures and their role:

- **PCMapper:** An array mapping interpreter bytecode instruction indices to the start of their compiled code implementation. This array is built as compilation proceeds. It is used for branch and jump table patching. It is also used to construct the exception handler jump table and to help debugging.
- **BranchInfo:** A list of branches and other sites that require some form of patching. Elements are added during code generation and are used later during the patching phase.
- **ExceptionMapping:** A linked list of mapping elements. Each element is a (*native pc offset*, *bytecode pc*) pair. These are only used if the compiler is configured for signals-based exceptions. Elements are added to the list during code generation of the method body. The list is then used to generate mapping data in the data section located at the end of the code array.

⁶Memory management is covered in chapter 7

⁷Exception handling is covered in chapter 6.

5.1.5 Dynamic Recompilation, Decompilation and Preparation Sequences

Dynamic recompilation [BD98] consists of dynamically, i.e. at run-time, compiling a method that has been previously compiled. Modern high performance Java virtual machines such as Jikes RVM [AAB⁺00] usually have several compilers or several level of optimizations with different compilation time / code efficiency tradeoffs. Fast compilers performing no or little optimizations are useful for interactive applications as well as short running applications that require a fast startup. A fast baseline compiler could be first used, obtaining better performance than the interpreter while not costing much in compilation time. Later, for methods that are found to be particularly *hot* (i.e. with frequently executed code), the method could be recompiled with an optimizing compiler that generates more efficient code at the cost of extra compilation time. Note that the extra time spent in optimizing a method could sometimes outweigh any benefits gained.

Dynamic recompilation can also be used for debugging purposes [THL02]. For example, methods are recompiled with debugging code to watch for the occurrence of some event.

With dynamic recompilation, in addition to the interpreter code, we might have *several* compiled code versions. Moreover, for multithreaded applications or for single-threaded applications with recursion, several versions of compiled code might be in use. Freeing memory allocated for slower versions of code must be delayed until they are no longer used.

Decompilation (or *deoptimization*) of a method consists of switching from the faster compiled code back to slower code (interpreted code in our case). This is useful for debugging code with a debugger [HCU92]. It is also used for undoing optimizations [Hot] that became invalid after some event. For example, a virtual invocation site that was once proven to be monomorphic (i.e. with a single target) could have become polymorphic (i.e. with more than one target) after new Java classes are loaded. In that case, any optimization done based on the previous analysis is no longer valid.

Decompilation could also be used when there are some concerns about memory usage. Consider a method that was once hot but is no longer. Reverting to interpreting code for such a method allows for freeing the memory allocated to the compiled code or reusing it for code of methods that became recently hot. This can be especially important for long running applications. With recompilation and decompilation, a system can better adapt to its current method work load.

SableJIT supports recompilation. Recompilation is achieved by invoking the compiler again and by updating a pointer in the method internal data structure in the virtual machine to point to the newly generated code. The old code is not freed right away, but from time to time the runtime *garbage code collector*⁸ checks if some old code can be safely freed and, if so, returns all its associated memory to the system. A usage of recompilation is discussed later in this section. SableJIT does not currently revert to interpreted code, but this functionality could be easily implemented. Reverting to interpreted code would consist of simply updating the compiled code pointer in the method data structure to point to a stub that invokes the interpreter and to mark the method as not compiled.

The compiler presently does not generate code to patch the slower preparation sequences (see section 2.3) with a faster variant. Instead, it generates profiling code and code to patch the interpreter code⁹. If some threshold is reached, the compiler is invoked to recompile the method. There are two advantages in recompiling instead of patching the native code directly. The first advantage is simplicity. Generating code that patches native code in a portable way seems to be rather difficult. We would like to keep the compiler as architecture independent as possible without resorting to architecture-specific code for each prepare sequence / architecture combinations. It would, however, be interesting to study in future work if this could be done in a retargetable and efficient way. The second advantage is space efficiency. Since the compiler does not generate code for dead preparation sequences, the recompiled code uses less memory. A disadvantage of not patching the code as it is executed is that

⁸Garbage code collection is covered in chapter 7.

⁹Actually the code fed to the compiler. For the direct-threaded and inlined threaded interpreter, this is the equivalent switch code that was computed.

we cannot take immediate advantage of the faster code, we need to wait until its next compilation. A second disadvantage is the extra compilation time incurred by recompiling the method; patching would be less costly.

5.2 Using Runtime Information

As mentioned in the previous sections, the baseline compiler does not perform optimizations per se. It does however use some precomputed and available information in the virtual machine. We will describe three of them: the operand stack sizes at each bytecode instruction, the synchronized modifier, and the number of non-parameter local variables of reference type.

5.2.1 Stack Sizes

The Java specification requires that the operand stack size at an instruction be the same no matter what computation path is taken to arrive at that instruction [LY99]. During the preparation of the interpreter code of a method, SableVM computes these sizes as they are required to compute information for garbage collection.

The information on stack sizes is quite useful for a baseline compiler. If we did not have such information, we would have to increment/decrement the operand stack pointer when elements are pushed/popped on/from the stack. For example, our implementation to pop the stack and store the popped value into a register would look like this:

```
// decrement stack pointer
arch.addi(stackPointer, stackPointer, -stackWidth);
// load top element
arch.loadI(reg, 0, stackPointer);
```

This is likely to be two instructions on a RISC architecture although some architectures support a *load with update* that could perform it in a single instruction.

If the stack size information is used, the compiler implementation becomes:

```
deltaStack--;
arch.loadI(reg,
```

```
(stackOffsets[pc] + deltaStack) * stackWidth,  
stackPointer);
```

where `deltaStack` is a variable used to keep track of the changes in the stack size within an instruction and where `stackOffsets[pc]` is the stack size at the start of this instruction. The actual code that is generated for this is a single load. The stack pointer points at the stack bottom at all time and an offset computed at compile time is used to access the element.

5.2.2 Method Synchronization

In the Java programming language, a method can be marked as *synchronized*. For *synchronized methods*, a monitor must be entered before executing the method body and must be exited when leaving the method. In the interpreter, the monitor is entered in the `INVOKE` instruction of the *caller* (see figure 5.3) and it is exited in the `RETURN` instruction of the *callee* or in the exception handler if the method exited abruptly due to an exception. Since the `INVOKE` and `RETURN` implementations are shared between all interpreted methods, an explicit check for synchronization is required in the interpreter. However, when the compiler is invoked, it is known whether the method we are compiling is synchronized or not. By moving the synchronization code from the caller to the callee, the compiled code can be specialized. No dynamic check at runtime (to see if the method is synchronized or not) is required. Rather, the synchronization code is simply either generated or not depending on whether the method is synchronized or not. In addition to eliminating the dynamic check, space is saved. If the synchronization code had not been moved to the callee for compiled code, it would have been necessary to generate it at each call site (as opposed to once per synchronized method) unless we could have determined that no synchronized method would ever be invoked at that call site.

Note, however, that changes in the calling conventions of Java methods make the transition between interpreted code and compiled code slightly more complex since these differences need to be taken into account. A method cannot be synchronized

twice in a single invoke. The caller, in interpreted code, must not execute the synchronization code if the callee is a synchronized compiled method. Otherwise, it would be synchronized by both the caller and the callee. Similarly, the stub allowing compiled code to invoke interpreted methods must do the synchronization if the method is synchronized before calling the interpreter. Otherwise, the method will not be synchronized as the caller (compiled code) assumes the callee will do it and the callee (interpreted code) assumes the caller has done it.

On a side note, it would be possible for the interpreter to perform all the synchronization in the callee by adding a bytecode instruction at the beginning of synchronized methods.

5.2.3 Initialization of Non-Parameter Reference Type Local Variables

The garbage collector implementation in SableVM requires that all object references either point to a valid object in the heap or otherwise have the `null` value. This requirement must hold wherever garbage collection can occur. Garbage collection can occur before a method local variable is assigned a valid reference or null. In order to satisfy this requirement, all local variables of reference type that are not parameters are initialized to null before executing the method body. Reference parameters passed to the method already have valid values and the operand stack that is initially empty also satisfies this requirement. Setting local variables of reference type to null is implemented efficiently in SableVM by reordering variables in order to regroup consecutively variables of reference type. Then, knowing the number of such variables is sufficient for initializing them with a simple `for` loop in the `INVOKE` implementation of the interpreter (see figure 5.3). This is done in the caller before executing the code of the callee. Since the number of non-parameter locals for a given method is fixed and known at compilation time, it is possible to move this task from the caller to the callee in a way similar to what we did with the synchronization code. Additionally, since the number of iterations is constant and relatively small, the loop is fully unrolled resulting in a series of native *store* instructions, one per variable.


```
Interpreter loop:

INVOKE:
  if (method->synchronized) {
    /* enter monitor */
  }
  /* nullify ref locals */
  for (i = 0; i < max; i++) {
    locals[java_args_count + i].reference = NULL;
  }

...

RETURN:
  /* exit monitor */

Compiled code:

prologue
[synchronization code]
set refs locals to NULL
body
[synchronization code]
epilogue
```

Figure 5.3: Caller Work Delegated to the Compiled Callee

5.3 Towards an Optimizing Compiler

Since SableJIT has only a baseline compiler, one could ask: how difficult would it be to add an optimizing compiler and how should one proceed? Figure 5.4 shows the UML diagram of the frontend that was first introduced in chapter 3. The baseline compiler implementation is located in the `OnePassGenerator` and directly uses methods of the backend (class `Architecture`). To add an optimizer the approach would consist of adding a class, say `MIRBuilder` to the hierarchy as shown in figure 5.5. The `Compiler` class would then invoke the `build_inst` methods of that class instead of the ones in `OnePassGenerator`. Each `build_inst` implementation would add one or more IR nodes to the IR data structure for the `inst` instruction. Additional classes would be added to perform analysis and optimizations on the IR built. Finally, another class could emit native code from the optimized IR by making use of the retargetable backend.

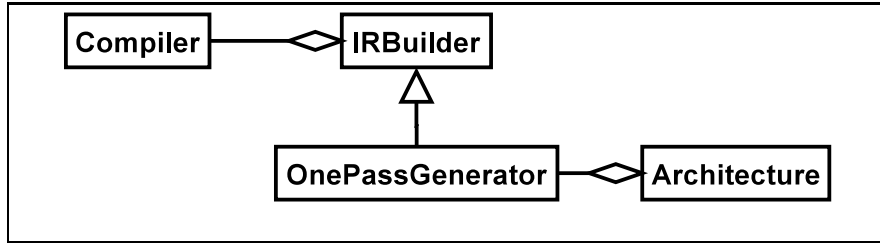


Figure 5.4: Compiler Frontend

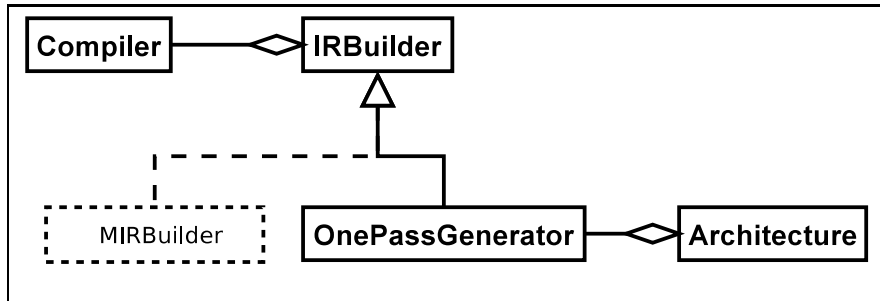


Figure 5.5: Compiler Frontend with a MIR Builder

5.4 Compiler Robustness and Failure Recovery

The compiler is quite robust in that it can recover from several types of compilation failures. This feature is especially useful during development and we will come back on this point in chapter 9 where we describe the suggested strategy to retarget the compiler to a new platform. We first describe several types of failures that can occur. Then we discuss when recovery is possible. Finally, we explain how the compiler and the runtime recover from compilation failures.

5.4.1 Failure Classes

Figure 5.6 illustrates the different types of exception classes and how they are related through inheritance. Failures can be grouped into the following categories:

1. **Unexpected unchecked exceptions:** Examples in this category include exceptions of type `NullPointerException` and `ArrayIndexOutOfBoundsException` that correspond to dereferencing a null reference and to accessing an array

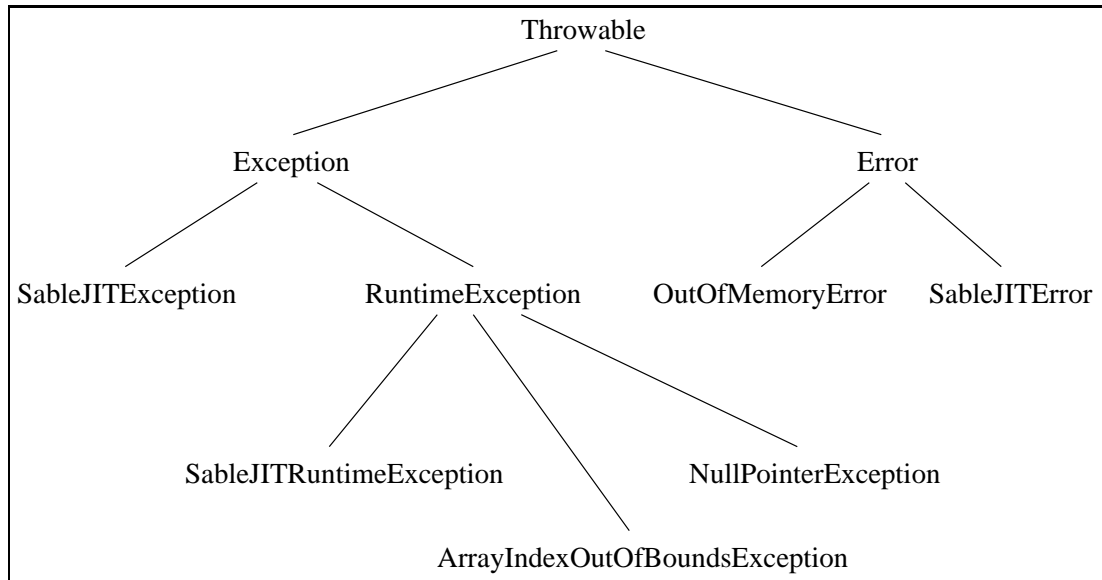


Figure 5.6: Partial Class Hierarchy of Exceptions and Errors

with an invalid index, respectively. These exceptions are subclasses (either direct or indirect) of type `RuntimeException`. These exceptions are said to be *unchecked* as it is not necessary for a method to declare that it might throw them. Also, there is no explicit `throw` statement for exceptions in this category. These are due to bugs in the compiler code.

2. **Virtual machine errors:** Errors in this category are subclasses (either direct or indirect) of type `Error` that are not also subclasses of type `SableJITError`. These errors usually originate from an unexpected event that is difficult to recover from. For example, if the virtual machine runs out of memory, an `OutOfMemoryError` error is thrown. Errors of this type are never thrown explicitly by the compiler code.
3. **SableJIT checked exceptions:** Exceptions of type `SableJITException` (or subclasses) are expected in the sense that they are thrown by an explicit `throw` statement. They are also *checked*; methods that can throw them must declare them. The compiler uses very few checked exceptions, most exceptions are unchecked.

4. **SableJIT unchecked exceptions:** Exceptions of type `SableJITRuntimeException` (or subclasses) that are also thrown by an explicit `throw` statement. However, a method need not declare that it throws such an exception. Most exceptions are of this type. We enumerate them with a brief description or example:

- `AssertionFailedException`: An assertion or safety check failed.
- `MappingException` and `TargetException`: Unable to map a program counter (pc) and unable to find a branch target respectively.
- `UnusedBytecodeException`: The compiler received an unused (or unknown) interpreter bytecode instruction.
- `RegisterException` and its subclasses: Errors related to register allocation such as running out of registers, or related to register usage such as providing an invalid register number.
- `NotImplementedException`: Indicates a not yet implemented feature. Usually these occur when retargeting or developing the compiler incrementally.

5. **SableJIT errors:** Errors in this category are of type `SableJITError` (or subclasses). These indicate that a non-recoverable fatal error occurred. Examples of this type are:

- `UnrecoverableExceptionError` and its subclasses: They occur when an exception is thrown in a context where it is known that no exception can be recovered from. The exception is caught and encapsulated in an error object of this type that is then thrown. An example is described next.
- `ExceptionInJITInitError` (a subclass of `UnrecoverableExceptionError`): An exception occurred during the bootstrap of the compiler.

5.4.2 Recoverability

SableJIT performs *backward error recovery* [DVLP98]. In backward error recovery (or simply *backward recovery*) the system state is restored to a previous state that is

known to be error-free. On the other hand, *forward error recovery* would consist of fixing the errors to allow the system to continue forward in time.

Backward recovery in our JIT involves two things: restoring the compiler data structures to a valid state and falling back to the last code version successfully obtained for the method that failed to compile. For efficiency, some objects in the compiler are reused for several compilations. Since compilation is abruptly interrupted when an exception is thrown, some compiler data structures and objects could have been left in an inconsistent state. The compiler needs to be restored to a consistent state. Our JIT can fall back to either interpreted code or, if recompilation is enabled, previously compiled code.

The runtime assumes that the compilation failure is deterministic, that is, if a method fails to compile once, it is likely that it will fail again. All methods that fail are marked as *uncompilable*, thereby preventing any attempt to compile them again.

In general, failures that throw exceptions, i.e. objects that are of type `Exception` or one of its subclasses, are recoverable (categories 1, 3 and 4) whereas failures where errors are thrown, i.e. objects of type `Error` or its subclasses, are not (categories 2 and 5). Exceptions occurring during the compiler bootstrap or during the initialization of compiler classes are treated as fatal. In these cases, if the exception is not of a subtype of `Error`, it is wrapped into an error object of type `sablejit.UnrecoverableExceptionError` and will be treated as an unrecoverable error.

5.4.3 The Recovery Process

Part of the recovery is accomplished by the compiler and part by the runtime. The recovery code of the compiler is located in the `Compiler.compile()` method since it is the compiler entry point and therefore the last point in the Java code that an exception or error can be caught before the control is returned to the runtime. This method has most of its body covered by exception handlers. Exceptions of type `Exception` (or one of its subclasses) and of type `SableJITError` (or one of its subclasses) are caught by these handlers. No attempt is made to intercept throwable objects of type `Error` (or a subclass) that are not of type `SableJITError` (or one of its subclasses). Failures

resulting in `SableJITError` (or a subclass) are not recovered; the compiler simply temporarily disables compilation¹⁰ and prints a stack trace before terminating the virtual machine. For recoverable exceptions, the compiler catches them and executes recovery code to bring the compiler data structures back to a consistent state. Any exception occurring in recovery code is treated as non-recoverable.

Regardless of whether the failure is recoverable or not, if the `Compiler.compile()` exits abruptly due to an unhandled exception, the runtime will detect it. If the exception is of type `Error` or one of its subtypes, the runtime terminates the virtual machine immediately. Otherwise, it knows that the compiler has successfully recovered from the exception. It then marks the method as *uncompilable* so no further compilation attempts are made. The runtime frees all the memory allocated for the compilation of the method. Then, the runtime clears the exception status and falls back to either the last successfully compiled code (if dynamic recompilation failed) or to the interpreter. The execution of the Java program now continues normally.

5.5 Flexible Method Entry Point and Compilation Unit

The implementation of the features described in this section is considered experimental. We emphasise the usefulness of such features and we present our experimental prototype.

5.5.1 Flexible Entry Point

Currently, compiled code can have two entry points. One at the beginning of the method body and a second one that can be basically at an arbitrary location in the method body. The implementation could be adapted relatively easy to support more than two entry points, although each entry point requires additional code and costs some space. A second entry point in the middle of a method is useful to switch from the interpreted code to the compiled code of a method without having to wait for

¹⁰Compilation is disabled as Java code is executed to print the stack trace. Doing otherwise, could lead to compilation of further methods and result in an endless loop.

the next time the method is invoked. Consider a method that has a *hot loop*, a loop with a large number of iterations and where a significant portion of execution time is spent. This method could be invoked only once during the course of the program. If compilation and invocation of compiled code occur only at method entry points, then, unless this method is compiled before its first invocation, it will never be compiled. However, if we add compilation points on loop back edges and allow switching from interpreter code to compiled code at these points, then it would be possible to take advantage of just-in-time compilation for that method.

SableJIT compilation on loop back edges is still considered experimental and it is therefore not enabled by default. The implementation is quite simple for the baseline compiler as it mimics the interpreter and operates directly on the Java stacks. It consists of generating a second prologue, called the *stub prologue*, that is described in section 5.1.3. The stub prologue does the work of a native function prologue (pushing a stack frame, saving callee-saved registers, ...), moves the compiled code arguments (`env`, `locals`, and `stack` pointers) to the appropriate registers and, finally, jumps to the back edge target. Once the method is compiled, the stub prologue rather than the normal prologue is used to switch to the compiled code. When the compiled code returns, the method is known to have exited.

5.5.2 Flexible Compilation Unit

The simplicity of the baseline compiler, combined with the fact that both compiled and interpreted code share the same Java stack, makes it easier to use a different compilation unit than a method body. Methods can have rarely executed code. For example, code in exception handlers could be infrequently executed. *Partial method compilation* [Wha01] consists of compiling frequently executed parts of the method rather than the full method. The main advantage is to better allocate compilation time where it matters the most. In SableJIT, partial compilation was implemented for simple loops as a proof of concept only, and is therefore incomplete. Profiling code is added to the loop back edges. After some threshold is reached, the loop code (not the full method) is compiled and the execution switches from the interpreter to the

compiled code for the remaining iterations of the loop. When the loop exits, execution switches back to the interpreter. Switching from the interpreter to compiled code and back to the interpreter in this way has little overhead since both work directly on the Java stack and perform the operations in the same way.

5.6 Summary

In this chapter we have presented the baseline compiler. We have discussed the frontend of the compiler and have described the steps involved in the compilation of a method. We have explained dynamic recompilation and decompilation. We have seen how the baseline compiler uses runtime information to improve performance. We have described how an optimizing compiler could be added and where it would fit in the compiler design. We have explained how the compiler and the runtime can recover from compilation failures. Finally, we have briefly studied how the baseline compiler simplicity makes it relatively easy to experiment with different entry points in the compiled code and with partial method compilation.

Chapter 6

Exception Handling

In this chapter, we discuss how exceptions occurring in compiled code are handled. We start by describing a simple implementation where all handling is done by the interpreter. Then, we present an improved implementation where exceptions can be handled without resorting to the interpreter. We explain how some checks such as null checks can be made implicit by the use of hardware traps. Finally, we conclude by summarizing the hardware traps available on various architectures and their use for exception checks.

6.1 General Exception Handling

6.1.1 Interpreter-only Exception Handling

In early versions of SableJIT all exceptions were handled by the interpreter. Since both the compiled code and the interpreter use the same stack, this solution is relatively simple. It does not require any additional data structure except for the inlined interpreter. We delay the discussion of this last point to later in this section. Throwing an exception in the compiled code is done in three simple steps:

1. Set the exception object in `env->throwable`.
2. Save the bytecode *program counter* (PC) in the Java stack frame.

3. Return -1 to signal that an exception occurred.

The first step simply stores a reference to the exception object such that it can be retrieved later by the exception handler. The *bytecode PC*, in the second step, is a pointer to one element past the current element in the interpreter code array. It is used to identify where the exception occurred. When the compiler is used in combination with the inlined interpreter, a table that maps the equivalent switch code PC (used as input to the compiler) to the corresponding inlined code PC is required. With the direct interpreter, a mapping table is not required as the mapping can be easily computed since both the direct and the switch code arrays have the same layout. In the last step, the return statement is not a Java return statement but a native function return. The return value of the compiled code function is always checked by the caller, whether the caller is the interpreter or some other (possibly itself) compiled method. If the caller is the interpreter function and the compiled code returns -1 , a jump is made to the exception handling code (see figure 6.1). If the caller is a compiled method and the callee returns -1 , then the caller returns immediately -1 . Eventually, the interpreter function will be reached and the exception will be handled there.

Figure 6.2 illustrates the process with a simple example. The effects on the Java stack and the native stack are shown after each step. Initially, the last 3 methods on the Java stack are *A*, *B*, and *C* (see figure 6.2(a)). *B* and *C* are compiled, the native stack has their native stack frame. *A* is an interpreted method, its corresponding native stack frame is the virtual machine interpreter function. The exception is thrown in method *C* and the first matching handler is located in method *B*. After constructing the exception object and saving the bytecode PC in the stack frame, the compiled code of *C* returns -1 to signal the exception (see figure 6.2(b)). The compiled method *B* notices that *C* has returned -1 and it simply does the same (see figure 6.2(c)). The control is now back to the interpreter. The interpreter notices that *B* returned -1 and it jumps to the exception handling code. Since, *C* has no matching handlers, the *C* Java stack frame is popped (see figure 6.2(d)). The next method on the stack, *B*, is checked for a matching handler. One is found and the

```
switch (bytecode) {
    case INVOKE*:
        ...
        stack_inc = compile_code(...)    [-1 is returned]
        if (stack_inc == -1) {
            goto exception_handler;
        }
        ...
}

exception_handler:
    /* Code to handle exception */
```

Figure 6.1: Compiled Code Signalling the Occurrence of An Exception

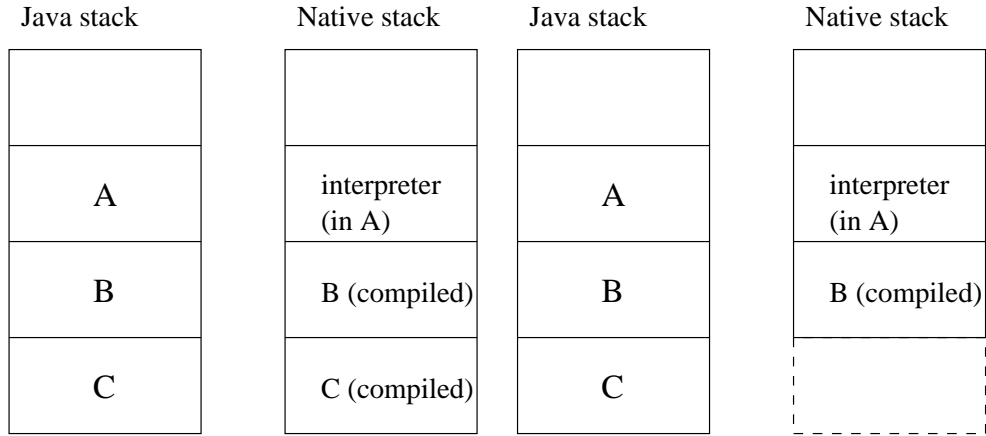
interpreter continues the execution of the Java program from the handler.

It is important to note that in this implementation the Java stack is not popped when compiled methods are returning -1 , it is left untouched. Only the native stack is popped. The end result is that the interpreter does not have to distinguish whether the exception occurred in compiled code or in interpreted code as in both cases the Java stack is exactly the same.

The main advantage of this method of handling exceptions is its simplicity. It allowed us to support exceptions at a very early stage in the development of SableJIT. However, it does handle exceptions quite inefficiently. Consider the case when several compiled methods are in progress with no intervening interpreted method on the call stack. If an exception is thrown by the bottommost compiled method and it happens that there is a matching exception handler in that same method, the native stack is still popped several times until an interpreter function is reached. Another inefficiency is that the exception handler code is always interpreted. These two disadvantages might not degrade the performance of an application much provided exceptions are rarely thrown, that is, if they happen only in exceptional cases¹. Nevertheless, an improvement over this implementation fixing these two issues has been implemented and is presented in the next section.

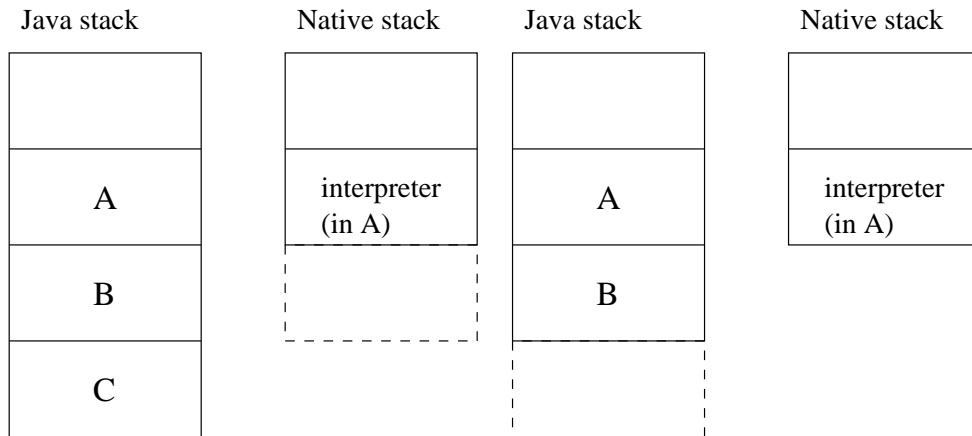
¹Several Java programs do however use exceptions on their normal control flow.

6.1. General Exception Handling



(a) Initial conditions. Method C throws an exception and the handler is in method B.

(b) Native code C returns -1 .



(c) Native code B also returns -1 . The control is now back to the interpreter.

(d) The exception handling loop of the interpreter pops the Java stack frame of method C. The control is transferred to the handler in B.

Figure 6.2: Interpreter-only Exception Handling

6.1.2 Joint Interpreter and Compiled Code Exception Handling

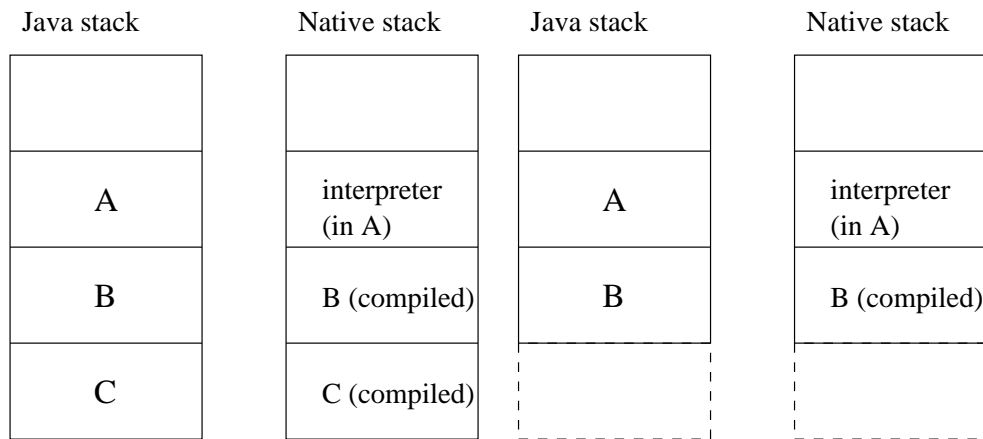
We improved over the previous implementation by allowing a compiled method to handle the exceptions for which it has matching handlers. Step 3 is changed to:

3. If the method has a matching handler, jump to the exception handler. Otherwise, pop the Java stack frame and return -1 .

Figure 6.3 illustrates the application of this implementation on the example presented in section 6.1.1. The compiled code of method C does not find any matching handler, it therefore pops the Java frame as well as the native frame, returning -1 in the process (see figure 6.3(b)). Control is now returned to the compiled code of method B . Since B has a matching handler, a jump is done to the handler inside the compiled method B . There are three important differences from the previous implementation to notice. First, the Java stack frames are popped as the corresponding native stack frames get popped. Second, the native stack frame of the compiled method containing the handler (B) is not popped. Third, execution of the handler starts in compiled code whereas in the previous implementation, the handler is interpreted.

6.2 Signal-based Exception Handling

SableVM has support for signal-based exceptions. Most processors raise a hardware trap (also called exception) on some operations considered invalid such as accessing an invalid memory location or performing an integer division by zero. The processor trap causes a control transfer to the operating system (OS). On Unix operating systems, the OS sends a signal to the process that causes the fault. For example, if a null pointer is dereferenced, the process receives a SIGSEGV (segmentation fault) signal. The default signal handler is to terminate the faulty process. It is however possible to replace the default handler with one of our own. The idea of signal-based exceptions relies on the fact that exceptions seldom occur. We are then interested in implementing the common case (no exceptions) efficiently even at the expense of



(a) The initial conditions are the same as figure 6.2(a).

(b) The native code C cannot handle the exception. It pops the Java stack frame and returns -1 . Control returns to native code B that handles the exception.

Figure 6.3: Compiled Code / Interpreter Mixed Exception Handling

increasing the performance cost of the rare case (exception thrown). Consider the `ILOAD` instruction that loads a value from an integer array into the operand stack. If the array reference happens to be null, a `NullPointerException` *must* be thrown. With signal-based exceptions, the explicit null check can be entirely removed. If the reference is null, accessing the array size located in the array header requires a native load operation on a very small effective address. In such cases, most architectures raise a trap that causes a `SIGSEGV` signal to be delivered to the process. This signal can be caught by the virtual machine process and “transformed” into a null pointer exception. The frequent case (the reference is not null) is executed faster as an explicit null check is no longer required, however, the exceptional case (reference is null) becomes more expensive. If the user program is not well designed and exceptions are thrown regularly, the execution time saved by removing the null checks can be outweighed by the expensive cost of the trap processing.

We have presented the general idea. We now introduce how signal-based exceptions affect the implementation of instructions in the interpreter, then we present how signal-based exceptions work in the presence of compiled code.

6.2.1 Signal-based Exceptions in Interpreted Code

The sample code in figure 6.4 illustrates parts of the implementation of the `GETFIELD_INT` instruction when signal-based exceptions are used. The `instance` variable is a pointer to the object instance and the `offset` variable, the field offset. If `instance` is `NULL`, a small memory address is accessed, causing a segmentation fault on most architectures. Note that there are no explicit null checks for `instance`. For comparison purposes, figure 6.5 shows the code of the same instruction when signal-based exceptions are not used. Even though an explicit check is not required with signals, it is required to always save the bytecode PC, otherwise the exception context would not be known. Since all methods share the same instruction implementations, the segmentation fault address is of no help to compute the context.

```
/* save pc in case exception is raised */
env->stack.current_frame->pc = pc;
/* get integer field */
stack[stack_size - 1].jint =
    *((jint *) (((char *) instance) + offset));
```

Figure 6.4: GETFIELD_INT Implementation *With* Signal-Based Exceptions

```
if (instance == NULL) {
    env->stack.current_frame->pc = pc;
    goto nullpointerexception_handler;
}
stack[stack_size - 1].jint =
    *((jint *) (((char *) instance) + offset))
```

Figure 6.5: GETFIELD_INT Implementation *Without* Signal-Based Exceptions

6.2.2 Signal-based Exceptions in Compiled Code

In section 6.2.1, we have seen how the interpreter uses signal-based exceptions. In particular, we have explained that it is necessary to save the bytecode PC at points where an exception could occur. In compiled code, it is however not necessary to save the bytecode PC as no memory is shared between the different compiled methods. The memory address of the faulting instruction uniquely identifies both a method and the location within that method where the exception occurred. Instead of always saving the bytecode PC in case an exception occurs, the bytecode PC is computed *only* when an exception do occur.

In order to compute the bytecode PC from the native address, it is necessary to map an absolute memory address to its corresponding bytecode PC. This is done by computing a mapping table (*memory address offset*², *bytecode PC*) at compile time and storing it at the end of the compiled code. The storage requirement is two words per location that might raise a hardware exception. The first word is the offset from the beginning of the compiled code and the second, the corresponding bytecode PC. For flexibility and simplicity, the entries are actually interpreted as ranges covering from the native offset (inclusive) up to the native offset of the next entry (exclusive) (see figure 6.6). This is useful as different architectures generate different sequences

²In order to keep the code relocatable, we actually store an offset from the beginning of the compiled code instead of an absolute memory address.

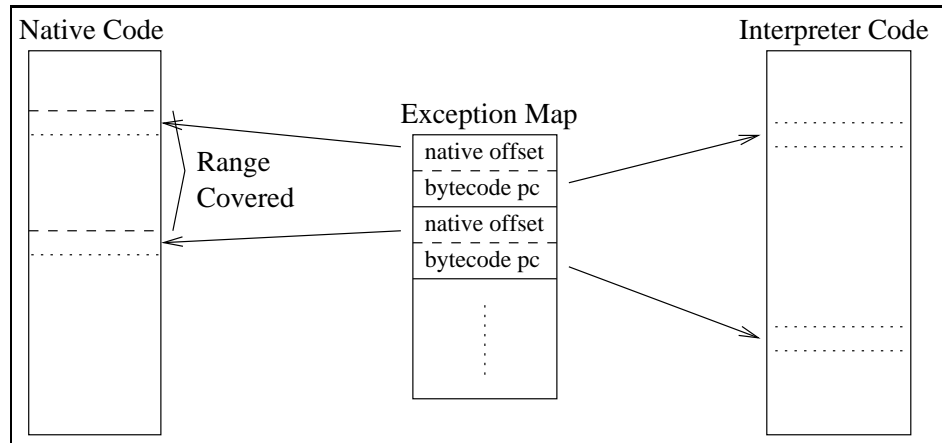


Figure 6.6: Mapping Native Addresses to Bytecode PCs

of instructions. The frontend does not need to know the exact instruction that causes the trap. Basically, when signal-based exceptions are used, instead of generating code for a null check, the compiler adds a table entry. It is important to note that the table requires less space than the code that would be otherwise required to perform a check and throw an exception. Therefore, signal-based exceptions are space efficient and should be time efficient provided that exceptions are not frequent.

During the execution, if a segmentation fault (or other) occurs, the address of the instruction that triggered it is obtained. If the current Java method executing is compiled, the address is compared with the compiled code boundaries of that method. If it occurs within the compiled code, the mapping table located at the end of the compiled code is consulted for the corresponding bytecode PC. The bytecode pc is then saved in the current Java stack frame and control is passed to the interpreter exception handler. It could be possible under some circumstances that the current method is compiled but we are currently interpreting it and therefore the faulting address is located within the interpreter function. In order to handle that case, if the address does not fall within the compiled code, the runtime assumes that the trap originates from the interpreter. In that case, the bytecode PC should have been already been saved so the control is simply passed to the exception handler.

6.2.3 Efficient Array Bounds Check

An array bounds check can be performed with a single unsigned comparison as follows:

```
if (index >=U size) {  
    // throw exception  
}
```

Some architectures provide one or more *trap* instructions that raises a hardware trap when some conditions are satisfied. In particular, the PowerPC architecture provides a single cycle *trap word* instruction with the following format: `tw cond,rs,rt`. This instruction compares the content of the two registers operands *rs* and *rt* and it raises a hardware trap if the specified condition *cond* is true. On the Linux operating system, a SIGTRAP signal is sent to the process. The virtual machine can then handle the SIGTRAP and throw the corresponding `ArrayIndexOutOfBoundsException`. In the next section, we present the different trap mechanisms available on various architectures.

6.2.4 Hardware Support on Various Architectures

Table 6.1 summarizes the signal-based exceptions available on supported architectures. Values in italics are enabled by default. The *null sigsegv* column indicates that a SIGSEGV signal is sent when a null pointer is dereferenced. Similarly, the *div sigfpe* column indicates that a SIGFPE signal is sent on integer division by zero. The *trap null*, *trap div* and *trap bounds* columns indicate trap instruction support is implemented for null checks, division by zero checks, and array bounds checks respectively. Signal-based exceptions on the x86 architecture requires no instruction for null pointer and division by zero checks. On the PowerPC architecture, division by zero gives an undefined result and does not raise any exception. Instead, a single trap word immediate instruction is used to check if the divisor is 0. To distinguish between array bounds exceptions and arithmetic exceptions (as both sends SIGTRAP), we negate the offset of one of them before adding it to the mapping table. Trap instructions are also supported for null pointer checks on the PowerPC, however, the default is not to use any instructions and to catch the SIGSEGV signal as it is done on x86. On the

6.3. Summary

	null sigsegv	div sigfpe	trap null	trap div	trap bounds
ppc	<i>yes</i>	no	yes	<i>yes</i>	<i>yes</i>
x86	<i>yes</i>	<i>yes</i>	no	no	no
sparc	<i>yes</i>	<i>yes</i>	no	no	<i>yes</i>

Table 6.1: Summary by architecture. Values in italics are enabled by default for signal-based exceptions

SPARC architecture, both dereferencing null pointers and dividing integers by zero raise CPU exceptions as it is the case on the x86. In addition, array bounds checks can be done in a way similar to the PowerPC architecture by using two instructions: a compare and a *trap on condition code*. It would be possible to also use trap instructions for null pointer and division by zero checks on the SPARC architecture, though there are no benefits in doing so and hence they were not implemented.

6.3 Summary

In this chapter, we have presented how exception handling is done in the presence of compiled code. We explained how SableJIT extends signal-based exception handling to compiled code. We concluded the chapter by listing the different hardware trap options available on various platforms.

Chapter 7

Memory Management

In this chapter, we present memory management in SableJIT. We start by briefly describing how SableVM partitions its memory. Then, we study the memory requirements of SableJIT. We present the memory manager. We explain how garbage collection is handled in the presence of compiled code. We discuss how unused compiled code is detected and freed. Finally, we describe object reuse in the compiler.

7.1 Memory Partitions in SableVM

SableVM partitions its runtime memory into several logical parts. Each memory manager uses a scheme that performs well with the allocation patterns of a specific component. Memory areas include the garbage collected heap, the Java stack, the class loading data, and the native references. Of these, the heap and the class loading data area are the most relevant to SableJIT memory allocation. The heap is where all object instances reside. Since the compiler is written in Java, all objects instantiated during compilation are allocated on the heap. Also, several data structures required for compilation (mostly arrays) are allocated on the heap by the runtime. Each class loader has a class loading data area. The class loading data area contains all the classes, methods, and fields data structures required by the virtual machine. It includes method code, both the interpreted code and the compiled code. Memory management inside SableVM and the reasoning behind the partitioning are described

in details in chapter 3 of [Gag02]. Before describing the memory management done inside SableJIT runtime, we discuss its memory needs and its memory allocation behaviour.

7.2 Memory Requirements of SableJIT

Data allocated is classified according to one of three levels of granularity.

- **Per compiled code data:** Data closely associated to a given compiled code version. If the code is recompiled, then this data differs from the previously compiled version. For example, the mapping table used for exception handling is per compiled code data.
- **Per method data:** Data computed for a given method that is required for compilation. If the code is recompiled, this data can be reused. Data in this category includes the equivalent switch code computed when the direct-threaded or inline-threaded interpreter is used.
- **Global data:** All the data required for compilation or runtime support that is not specific to a method. This includes, for example, configuration data, data structures used for the direct-threaded/inline-threaded code to switch code conversion, compiler classes, and object instances.

All the *per compiled code data* is allocated either within the compiled code or at the end of the compiled code array. Figure 7.1 shows the compiled code layout. Jump tables are located within the code following the jump instruction. A (*native offset, bytecode pc*) mapping is located at the end. It is used by the runtime to compute the bytecode pc when signal-based exceptions occur. The last entry in the array indicates the mapping table size. The memory management of per compiled code data is quite simple. If the compiled code is ever freed, its accompanying data is also freed without having to rely on additional data structures or processing.

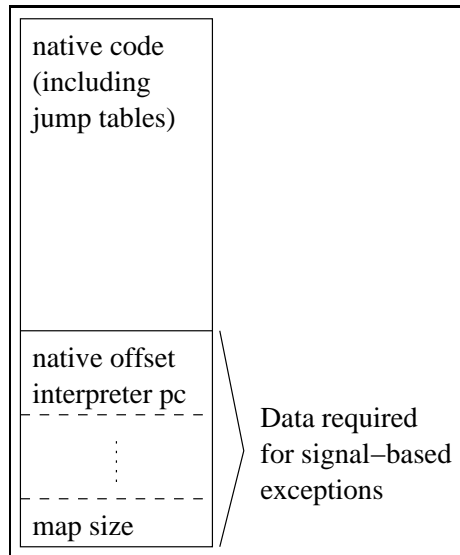


Figure 7.1: Native Code and Data Layout.

We now describe typical types of data allocated by the runtime and the compiler. For each of them, we mention their memory requirements and their allocation patterns.

- **Compiled code arrays:** Once ready, compiled code must be allocated at a fixed memory location. Though the generated code is relocatable, such code cannot be moved while it is being executed by some thread. Code arrays can be freed if no longer needed. Compiled code arrays should be allocated in the class loading data area such that if a class loader and its loaded classes are ever unloaded, all associated memory would be released at once.
- **Stack Offsets and `bpc2inlinedpc` arrays:** These arrays are computed by the runtime and are used by the compiler during compilation of a method (per-method data). They are freed when they are no longer needed, that is, if the method fails to compile or if the method has been successfully compiled and will not be recompiled. In order to be used by the compiler (written in Java), the runtime allocates them on the Java heap as Java `int` arrays.

- **Equivalent switch code arrays:** When the direct-threaded or inline-threaded interpreter is used, an equivalent switch code array is computed by the runtime and fed to the compiler. This array cannot be moved as it could be patched during execution to be reused later for recompilation. This array can be freed if compilation fails or if it will not be reused for compilation. As is the case for compiled code arrays, these arrays should be allocated in the class loading data area.
- **Compiler classes and objects:** Compiler classes are loaded by the `SableJIT-ClassLoader`, all the associated data is therefore allocated in that class loader data area. Compiler objects are allocated on the heap.
- **Global runtime data:** Most of the global runtime data of SableJIT such as configuration data is allocated statically. Dynamically allocated data is allocated in SableVM global memory.

7.3 SableJIT Memory Manager

The SableJIT memory manager does not currently perform custom allocation. That is, it relies heavily on both the garbage collected heap and class loader memory managers to actually obtain memory. It is designed to provide a centralized and unified interface for all SableJIT memory allocation needs. By having a memory manager, we avoid having to adapt and modify the class loader memory manager to suit the needs of SableJIT. Also, this centralization makes it easy to compute statistics on memory usage or to later experiment on custom memory allocation that takes into account the allocation/freeing patterns of SableJIT. The memory manager does play an essential role in managing code arrays especially when these are to be freed.

For allocation that should occur on the Java heap, the memory manager allocates a native reference. Native references provide a handle to objects in native C code. These references also prevent the objects from being garbage collected. The memory manager creates the object instance and assigns it to the local reference. To free the

allocated memory, the native reference is simply freed so the object instance memory can be reclaimed at the next garbage collection.

For allocations that should occur in the class loading data area, the SableJIT memory manager uses the class loader memory manager to allocate enough space to store both a header and data. The header contains the amount of memory allocated. This is necessary as the class loader memory manager is especially designed for allocating memory and freeing subareas. As such, it does not contain size information.

The class loader memory manager might not fit well with the SableJIT model if a lot of allocated data is later freed. It would be interesting to design a custom allocation scheme that could allocate large blocks from the class loader memory manager and then sub-allocates smaller blocks in a scheme that would correspond better to SableJIT memory allocation patterns.

7.4 Garbage Collection

Since SableJIT directly manipulates the Java stack and the locals, we do not need to maintain an extra root set for garbage collection. At a garbage collection point, all references have been written either to an operand stack slot, a local, a field, or some other location traced by the garbage collector. At such points, neither native registers nor native stack locations contain references that are not already part of the root set.

The compiled code stores the state at points where garbage collection could occur. These points are the same as in the interpreter. Saving the state consists of saving the current bytecode pc and the current operand stack size in the current Java stack frame. This information is sufficient for the garbage collector to obtain the gc stack map at the specific point and use it to retrieve references from the operand stack. Other references are retrieved by other means and are independent of the current location within a method. Basically, everything is done as in the interpreter. In addition, the runtime needs to take into account that compilation could trigger garbage collection. Therefore, compilation can only occur at gc safe points.

7.5 Garbage Code Collection

SableJIT is able to safely collect unused compiled code that could occur in the presence of recompilation and when reverting to the interpreted code. This operation must be done with care as older code versions could still be in use by either some other thread or, if the method is called recursively, by the current thread. We refer to this process as *garbage code collection*. Our algorithm relates to a tracing garbage collector in two ways. First, it is executed only periodically and second, it traces the stack of each threads to identify code still in use.

When the runtime requests compiled code to be freed, the memory is not released right away, instead it adds a pointer to the compiled code to a global list of *potentially* unused code. The total size of code on the list is maintained. At garbage collection time, as threads are stopped their native PCs are recorded. If the total memory that could be potentially freed exceeds some threshold, code is garbage collected. The collector uses the native PCs to walk the native stacks and collect the return address in each native stack frame. Then, for each compiled code element in the global list, a check is done to see if it is used by some thread, that is, if a return address falls within the boundaries of a compiled code version. If there are none, the actual memory used by the code can be safely freed.

A second approach would be to save a pointer to the compiled code in the Java stack frame (or a NULL value if interpreted) at each method invocations. This approach would have made each method invocation slightly more expensive but it would have simplified the implementation significantly. Presently, two small primitive functions to walk the stack need to be implemented for each architecture supported.

7.6 Memory Management Within the Compiler

Since the compiler is written in Java, all compiler objects, some compiler arguments and any objects or data structures created during the compilation process are allocated on the garbage collected heap. In order to reduce pressure on the garbage collector, a pool of compiler objects is used. The objects that are reused are all the

main compiler objects in both the frontend and backend (subtypes of `IRBuilder`, `Architecture` and `RegisterAllocator`) as well as the code arrays. Of these, only the `IRBuilder` reference is explicitly stored in the pool, references between these objects prevent the garbage collector from collecting them. The object graph is therefore not recreated for every compilation task, only part of their state must be reset before their next use.

Code arrays are dynamically increased if required. By reusing code arrays for future compilations, we save on allocation and initialization time. Allocating a new array every time would have required that its memory be zeroed by the virtual machine each time.

The maximum pool size is fixed at build time. The actual pool size defaults to the maximum value but a smaller value can be specified with the following command line option:

```
-C pool-size=VALUE
```

The pool size determines the number of compilations that can be in progress at any time.

7.7 Summary

In this chapter we have presented the different memory allocation requirements and patterns of SableJIT. We have described the SableJIT memory manager. We have seen how the baseline compiler simplicity makes garbage collection support for compiled code easy. We have studied how older versions of compiled code are safely garbage collected by the runtime. Finally, we have seen how the compiler maintains a pool of objects and reuses them for efficiency.

Chapter 8

Retargetability

In this chapter, we present the retargetable backend of SableJIT. We start by describing the backend overall design. Then, we introduce the *RT*¹ *code* model that serves as an architecture-independent layer. We discuss register allocation and usage. We study the special case of the x86 CISC architecture and how we have proceeded to make it fit the RISC-based RT code model. We conclude the chapter by presenting various services provided by the backend such as maintaining code arrays and performing architecture-independent branch patching.

8.1 Backend Architecture

SableJIT retargetability comes mostly from the design of its compiler backend. Figure 8.1, first introduced in chapter 3, shows the UML diagram of the core backend classes. In order to facilitate porting, the architecture dependent backend components are mainly divided into two parts: the instruction set and the *abstract binary interface* (ABI). The `Architecture` class serves as the interface to the backend. Subclasses of `Architecture` implement the instruction set for a specific architecture. The ABI consists of a sets of conventions regulating register usage and function calls on a specific platform. On a given processor architecture, these conventions can differ

¹The name RT comes from the word *retargetable*.

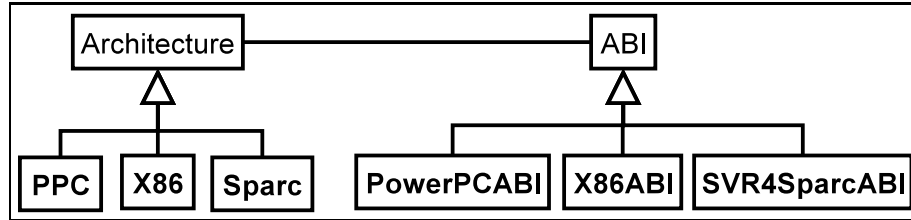


Figure 8.1: Compiler Backend

from one operating system to another. The implementation of the ABI for a platform is located in a class implementing the interface `ABI`. The `Architecture` class contains ABI-related methods that simply delegate the work to the accompanying `ABI` class. The frontend generates the code by invoking methods in the `Architecture` class. In other words, the `Architecture` class presents an architecture independent interface to the frontend.

Table 8.1 summarizes the different CPUs and operating systems (OS) currently supported². Each row defines a platform. The first and second columns are the CPU and ABI respectively representing that platform. The third column lists operating systems using that CPU and ABI combination. The fourth and fifth columns are the name of the `Architecture` subclass and the name of the class implementing the `ABI` interface respectively. Note that on the PowerPC architecture, the System V ABI and the Mach-O ABI did not differ enough to justify the use of two separate ABI classes. Both use the same class and the small set of differences are taken into account. Figure 8.2 illustrates these differences. Most of them are accounted for by a slightly different native stack layout (see figure 8.3). The Mach-O ABI requires that space be reserved in the *linkage area* to store the *condition register* (CR) in addition to the *link register* (LR) and the *stack pointer* (SP). This makes the LR as well as the function parameters accessible with different offsets from the SP.

A third type hierarchy not illustrated in figure 8.1 is the `RegisterAllocator` hierarchy. A `RegisterAllocator` object maintains a list of registers available on

²Linux on both x86 and ppc were used as development platforms. FreeBSD/x86 and Darwin/ppc are testing platforms. Signal-based exceptions are not yet supported on these last two. Solaris/sparc is a recent development platform.

8.1. Backend Architecture

CPU	Convention	OS	Architecture	ABI
ppc	System V	Linux	PPC	PowerPCABI
ppc	Mach-O	Darwin, Mac OS X	PPC	PowerPCABI
x86	System V	Linux, FreeBSD	X86	X86ABI
sparc	System V	Solaris	Sparc	SVR4SparcABI

Table 8.1: Classes Implementing the Backend of Supported Platforms

```
// offset to function parameters
private static final int PARAM_OFFSET = Configuration.IS_DARWIN ? 8 : 4;

...

// frame size
frameSize = Configuration.IS_DARWIN ? 128 : 112;

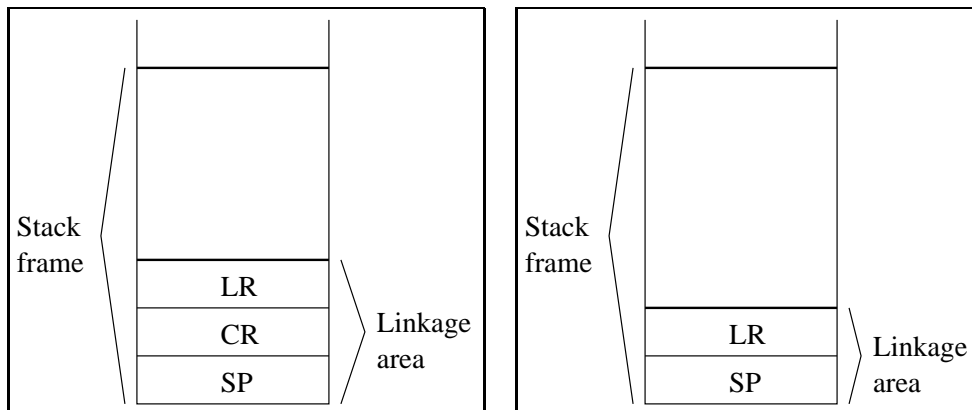
...

// save LR
arch.mflr_ppc(0);
arch.storeI(0, Configuration.IS_DARWIN ? 8 : 4, FP);

...

// restore LR
arch.loadI(0, Configuration.IS_DARWIN ? 8 : 4, FP);
arch.mtlr_ppc(0);
```

Figure 8.2: Code Segments Affected by ABI Differences



(a) Mach-O ABI

(b) System V ABI

Figure 8.3: Differences in Native Stack Layout on the PowerPC

a platform and register categories. The `RegisterAllocator` instance is obtained through the ABI interface.

8.2 RT Code Model

In order to facilitate retargetability, we follow an approach similar to VCODE [Eng96]. A RISC-based *virtual architecture* model is used. We call our model *RT code*. As on RISC architectures, most *RT instructions* are simple and take three register operands: two source operands and a destination operand. Data transfers between memory and registers are accomplished with *load* and *store* instructions available for different data types. Retargeting the backend to a new platform consists mainly of implementing the set of RT instructions for the target platform.

There are two main motivations for choosing a RISC-based model. First, a RISC architecture has a small and simple instruction set. This reduces the number of instructions to implement for each native platform. Second, several common native architectures such as PowerPC and SPARC are RISC architectures. These architectures map very well to our model. An important exception is the x86 architecture that is a CISC architecture. Although the x86 architecture does not fit our model as well as RISC architectures, it does have a large and varied instruction set that makes it possible to map it to our model relatively efficiently as we will see in section 8.4.

The *RT architecture* is implemented by a set of Java methods located in the architecture classes as opposed to a set of C macros and functions as it has previously been done in VCODE and in icode [Kaf]. The prototype of a method corresponding to a typical RT instruction such as a register add is: `addI(int rd, int rs, int rt)`. This method, when invoked by the frontend, generates native code that adds the content of registers `rt` and `rs` and stores the result in register `rd`.

Most RT instructions have Java semantics. In particular, instructions such as integer division, floating point operations and type conversions behave in a well-defined way on all platforms even for exceptional cases. For example, the Java specification [LY99] states that division overflow, that is, dividing `0x80000000` by `-1`, must

give `0x80000000` as the result and must not throw any exceptions. On both the PowerPC and the x86 architectures the implementation of the integer division requires additional native instructions to handle this special case.

To simplify the implementation of the RT instruction set on 32-bit architectures, a pair of registers is provided for operands of type `long` (64-bit integer) in RT methods. For example, the method `addL(int rd1, int rd2, int rs1, int rs2, int rt1, int rt2)` generates code for a `long` addition. On 32-bit architecture, the high word of the operand is provided in the first register of the pair (i.e. `rd1`, `rs1`, and `rs2`) and the low word is provided in the second register of the pair (i.e. `rd2`, `rs2`, and `rt2`). On 64-bit architectures, the first register argument in a pair is treated as a 64-bit register and the second register is simply ignored³.

The RT code provides a level of abstraction between the native architecture and the frontend. The baseline compiler does not construct any data structures containing RT instructions, instead, the frontend simply invokes the RT methods of the backend to generate the code. With the addition of an optimizing compiler, an intermediate representation composed of RT instruction nodes could be built to perform optimizations before emitting the code.

8.2.1 Instruction Types

Table 8.2 enumerates the types that RT instructions operate on. Most operations are performed on types: `int`, `long`, `float`, or `double`. Types `byte`, `short`, and `char` are mostly used in memory operations and type conversions. As opposed to VCODE where the types correspond to C types, the RT types are well defined and map directly to Java types. An `int` is always a 32-bit signed integer and a `long` a 64-bit signed integer no matter if the native platform is 32-bit or 64-bit. As a current limitation, SableJIT does not have a pointer type. Pointer values are stored into integer registers and instructions of type `int` are performed on them.

³SableJIT does not currently support any 64-bit architectures.

Name	Code	Description
int	I	32-bit signed integer
long	L	64-bit signed integer
float	F	single precision float
double	D	double precision float
byte	B	8-bit signed integer
short	S	16-bit signed integer
char	C	16-bit unsigned integer
void	V	void

Table 8.2: RT Code Instruction Types

8.2.2 Naming Convention

A convention is used to name RT instructions and their methods. Each name consists of a descriptive base name that can be followed by a single-letter type code (see table 8.2). Implementors wishing to provide additional instructions for modularity or flexibility are encouraged to append an architecture suffix `_arch` where `arch` identifies the architecture (for example, `_ppc`, `_x86`, ...). This suffix makes it clear that the method does not implement a defined RT instruction. Untyped instructions such as function calls do not have a type code. Finally, type conversion instructions do not have a type code appended as their types are made implicit in their base name. We provide some examples:

- `addI`, `addL`: *add integer*, *add long* (*I* for `int`, *L* for `long`).
- `addiI`: *add integer immediate*.
- `mtlr_ppc`: *move to link register* (a PowerPC-specific instruction).
- `i2f`, `i2d`: *integer to float*, *integer to double* (*x2y* converts type *x* to type *y*).

8.2.3 Default Implementation

Some RT instructions inherit a default implementation from the `Architecture` class. On some architectures it is possible that the default implementation happens to be the

best. On others, overriding the default implementation could improve performance. In that respect, RT code is somewhat similar to VCODE that has a core instruction set and extension layers implemented in terms of the core instructions. However, since most RT instructions are quite primitive, no clear delimitation between core and non-core instructions are made. Most, if not all, instructions should be implemented. Default implementations allow a quick port that could be improved later. Examples of instructions with a default implementation are:

- Several instructions with operands of `long` type expressed in terms of 32-bit instructions. Ex: `andL` is implemented as two `andI`.
- Some immediate variants expressed in terms of a *load immediate* followed by the register-only variant. Ex: `muliI` implemented as a `liI` followed by a `mulI`.
- Instructions expressed in terms of some other instruction through modification of the operands. Ex: `subiI(rd,rs,imm)` as a `addiI(rd,rs,-imm)`
- Complex instructions including `remL`, `remF`, `remD` and several type conversions expressed in terms of an internal C function call.

8.2.4 RT Instruction Set

Table 8.3 and table 8.4 show the RT instruction set. RT instructions are classified into three categories: simple instructions, ABI instructions, and jump table primitive instructions. Note that there are two *load* instructions operating on the byte type: `loadUB` and `loadSB`. The first one zero-extends the byte into the destination register and the second one sign-extends it.

Simple Instructions

Instructions that fall into this category map trivially to a single or relatively few (2 or 3) native instructions on a RISC architecture. For example, the *add integer* instruction, `addI`, performing addition on register operands is implemented in a single instruction on a typical RISC architecture. A second example is the *add immediate*

Simple Instructions

Name	Type	Arguments	Description
li	I, L	rd, imm	load immediate into register (rd = imm)
mr	I, L, F, D	rd, rs	move register (rd = rs)
load	UB, SB, S, C, I, L, F, D	rd, offset, rb	load from memory into register (rd = mem[rb + offset])
store	B, S, I, L, F, D	rs,offset,rb	store from register into memory (mem[rb + offset] = rs)
add, sub, div, rem	I, L, F, D	rd, rs, rt	binary arithmetic operations (rd = rs op rt)
neg	I, L, F, D	rd, rs	negation (rd = - rs)
shlX_rrn, shrX_rrn, ushrX_rrn	I, L	rd, rs, n	shift by n (rd = rs sh_op n)
shlX_rrr, shrX_rrr, ushrX_rrr	I, L	rd, rs, rt	shift by rt (rd = rs sh_op rt)
and, or, xor	I, L	rd, rs, rt	logical operations (rd = rs op rt)
addi, subi, muli, divi, remi, andi, ori, xori	I	rd, rs, imm	immediate operations (rd = rs op imm)
cmpL	L	rd, rs1, rs2, rt1, rt2	compare long (rd = cmp(rs1,rs2,rt1,rt2))
cmplF, cmpgF, cmplD, cmpgD	F, D	rd, f1, f2	floating point comparisons (rd = cmp(f1,f2))
ifxxI	I	rs, label/addr	compare against 0 and branch if condition xx is true
ifcmpxxI	I	rs, rt, la- bel/addr	compare registers and branch if condition xx is true

Table 8.3: Selected RT Instructions - Part 1

Simple Instructions

Name	Type	Arguments	Description
jumpReg, jumpImm, jumpLabel	N/A	rs/imm/label	unconditional branches
rawWord, rawLabel	I	imm/label	write argument verbatim in code array
i2l,i2f,i2d, l2i,l2f,l2d, f2i,f2l,f2d, d2i,d2l,d2f, i2b,i2c,i2s	–	rd, rs	type conversion ($rd = x2y(rs)$)
nop	N/A	none	no operation
trapWordEqZero	I, L	rs	trap on zero
trapWord- ArrayBounds	I	rs, sizeReg	trap on array index out of bounds

ABI Instructions

Name	Type	Arguments	Description
prologue	N/A	–	function prologue
epilogue	N/A	–	function epilogue
moveArg	I	rd, n	move function argument n to register rd
pushRegArg	I, L, F, D	rs	push register argument
pushImmArg	I, L, F, D	imm	push immediate argument
callReg	N/A	rs	indirect function call
callImm	N/A	imm	function call
moveReturnValue	I, L, F, D	rd	move return value in rd
return	I, V	rs/none	return value in rs or void
returni	I	imm	return immediate value

Jump Table Primitives

Name	Type	Arguments	Description
jumpTable, jumpRelative	N/A	rs	jump primitives
computePC, computePCImm	N/A	rd, label/imm	computes the PC identified by rd

Table 8.4: Selected RT Instructions - Part 2

integer instruction (`addiI`). On most RISC architectures it requires one or two native instructions depending on whether the immediate value fits into the immediate field of the instruction. RT code consists mostly of simple instructions.

ABI Instructions

Instructions that fall into this category are related to the ABI. They provide an abstract view of the native ABI, hiding the specifics from the frontend. The implementation of these instructions is located in ABI classes. Some instructions require several native instructions while others are quite simple. We briefly summarize each one:

- `prologue` and `epilogue`: Generate the prologue and the epilogue respectively.
- `moveArgI`: Loads an argument passed by the caller in a register.
- `pushRegArgX` and `pushImmArgX`: Pushes a function argument on the native stack or moves the function argument in a register.
- `callReg` and `callImm`: Call a function. The function address is provided either in a register or as a immediate value.
- `returnI` and `returniI`: Set the return register or stack location with the provided return value and then jump to the epilogue. The value can be specified by a register or immediate operand.
- `moveReturnValueX`: Retrieves the return value of the callee and stores it in a register. This instruction, when used, must follow immediately a `callReg` or `callImm` instruction. Otherwise, the value is undefined.

To perform a function call, a series of `pushXXArg` instructions are done consecutively before the actual call with a `callReg` or `callImm` instruction. Arguments are “pushed” from left to right in the order they appear in the function declaration. Contrary to what the name could imply, the arguments do not actually need to be

pushed on the stack; they can be passed in registers depending on the native architecture. Furthermore, the RT architecture does not dictate when the arguments are actually moved. It is possible to delay all code generation to the `callReg/callImm` instruction and to have `pushXXArg` implementations simply collecting information on the call. This flexibility is particularly useful for architectures such as x86 where arguments need to be pushed in reverse order.

The organization of the native stack is also left to the implementor. There are no requirements other than respecting the ABI of the native architecture. Figure 8.4 illustrates the native stack layout for compiled code on the PowerPC platform. Space in the stack frame is reserved for a parameter area and a linkage area to satisfy the native ABI. The *stack pointer* (SP) always points to the previously saved SP forming a back chain. The SP must also be aligned on a 16-byte boundary by adding necessary padding. For simplicity, a *general purpose register* (GPR) is used as a *frame pointer* (FP). Space is reserved to store any callee-saved GPRs that might be used. As no callee-saved *floating point registers* (FPR) are used by the baseline compiler, no space is allocated for them. Also, no space for variables is required as the baseline compiler uses the Java operand stack.

Jump Table Primitives

Instructions in this category are used in the implementation of jump tables and *jump subroutine* (`jsr`). Their specification is kept simple to minimize the amount of work required to port to a new architecture. Most functionality to generate jump tables is located in the frontend. However, the implementation of these instructions is less trivial than the first category and they usually require several native instructions. This category includes the following instructions:

- `computePCImm` and `computePCReg`: Two basic primitives that are sufficient to implement the two types of jump tables and the `jsr` instruction.
- `jumpTable`: This instruction is used to implement a table switch. It loads the table entry corresponding to the index argument and then jumps to the target loaded.

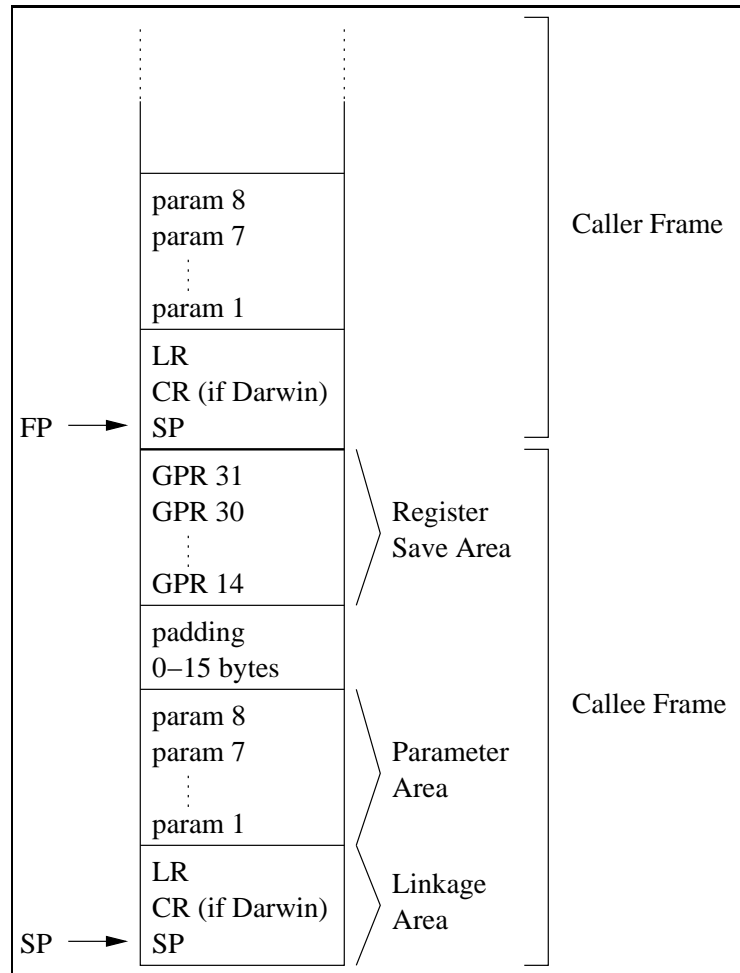


Figure 8.4: Native Stack Layout Used by Compiled Code on PowerPC

- `jumpRelative`: This instruction is used to implement a `jsr`. The jump is relative to the beginning of the compiled code array.

A RT instruction similar to the `jumpTable` could be added for the *lookup switch* bytecode implementation. However, the lookup switch implementation is complex as a binary search is used for efficiency. Instead of requiring complex code to be implemented for each architecture, the code implementing the lookup switch was implemented once in the frontend using RT code. The two primitives `computePCImm` and `computePCReg` are used in the implementation. They have the following prototypes:

```
public void computePCImm(int rd, int address)
public void computePCReg(int rd, int addressReg)
```

They differ only in their second operand. The first one takes an immediate value (known at compile time) for the address whereas the second one takes a register argument. The code generated by these methods computes an absolute memory address and stores it in the destination register `rd`. The computed address corresponds to `&code[address]`, that is, it identifies some location in the code array. The `tableswitch` and `jsr` bytecodes are, by default, expressed in terms of these two primitives. Computing absolute addresses for jump tables could be avoided by allocating the tables in fixed memory during code generation and providing the compiler with the address. Currently, they are allocated within the code array.

8.2.5 Example

We review the example introduced in section 5.1.2. It illustrates how the frontend uses the retargetable backend. We consider the Java bytecode instruction `iaload`. This instruction pops an *array reference* and an *index* from the Java operand stack. The element at the given index is retrieved and pushed on the operand stack. Figure 8.5 shows the memory layout of arrays used by the virtual machine. An array header containing information such as the array size is located before the array elements. The array reference points to the header. The steps required to execute the `iaload` instruction are as follows:

1. Pop the array reference.
2. Pop the index.
3. Check if the reference is `null`.
4. Check if the index is within bounds.
5. Compute the element address.
6. Load the element value.

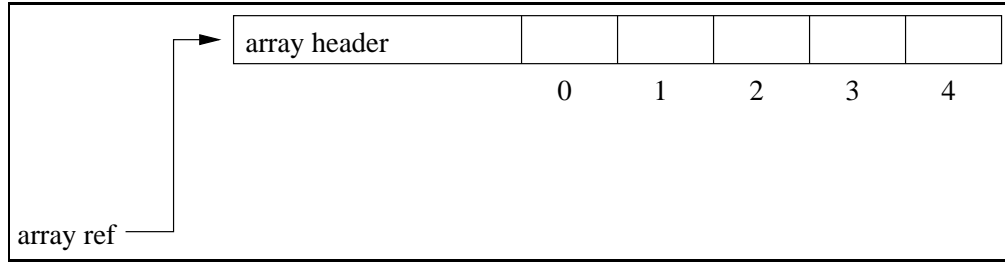


Figure 8.5: Memory Layout of Arrays

7. Push the value.

These steps are general and common to all architectures, only the native instructions implementing them differ. Figure 8.6 shows the implementation found in the baseline compiler. In line 4 and 5, `m4sj_popI` is a macro that expands to a `loadI` RT instruction. Using this macro, the array index and the array reference are popped from the operand stack and stored in registers `tmp1` and `tmp2`. At lines 7 and 8, RT code is generated for a null pointer check and an array bounds check. In line 11, `shlI_rrn` generates code to shift the register `tmp2` by 2 in order to multiply by 4. In line 13, `addI` adds the content of register `tmp2` to register `tmp1`. In line 15, `loadI` loads the array element. The offset from the base in the `loadI` instruction is used to skip the header. Finally, in line 18, the content of register `tmp1` is pushed on the operand stack.

It is important to note that this method is shared between all architectures. The native code generated depends on the type of the `arch` instance. For example, on the PowerPC architecture, `arch` is an instance of class `PPC`. The methods of the `PPC` class are invoked by the frontend and the PowerPC implementation of the RT instruction set is used.

A second thing to observe is that if the memory layout of arrays is changed, it is not necessary to modify any architecture specific code. Simply modifying the frontend is sufficient. The same statement applies to object layouts. Furthermore, simply porting to a new platform does not require knowledge of the virtual machine internals.


```
1 // arrayref index --> value
2 public void build_iaload(int currentPC) {
3
4     m4sj_popI(tmp2); // index
5     m4sj_popI(tmp1); // ref
6
7     checkNullPointerException(currentPC, tmp1);
8     checkArrayIndexOutOfBoundsException(currentPC, tmp1, tmp2);
9
10    // multiply index by 4
11    arch.shlI_rrn(tmp2, tmp2, 2);
12    // find element
13    arch.addI(tmp1, tmp1, tmp2);
14    // load element
15    arch.loadI(tmp1,
16               SableVMConstants.ALIGNED_ARRAY_INSTANCE_SIZE,
17               tmp1);
18    // push it on stack
19    m4sj_pushI(tmp1);
20 }
```

Figure 8.6: Implementation of `build_iaload` in RT Code

8.3 Register Allocation

The backend provides a `RegisterAllocator` interface. Classes implementing the `RegisterAllocator` interface do not perform register allocation per se, in that they do not automatically handle register spills and register loads. They simply provide architecture-specific register numbers to the frontend. They contain methods that are used by the frontend to allocate (request) and to free (release) registers.

There are currently four categories of registers recognized by the allocator. *Temporary registers* are integer registers that are not preserved across function calls. *Saved registers* are integer registers that are preserved across function calls. Finally, *float registers* and *double registers* are registers for single precision and double precision floating point numbers respectively. Floating point registers (FPRs) do not need to be preserved across function calls. This register classification is sufficient for the baseline compiler.

8.3.1 Register Requirements and Usage

For the baseline compiler, 2 float registers, 2 double registers and several temporary and saved registers are required. On some architectures such as SPARC, 2 single precision FPRs are used to store a double precision number. In that case, the first register of the pair is used to identify the pair and this is transparent to the frontend.

By collecting register usage statistics experimentally, we have found that 9 temporary registers and 9 saved registers seemed to be sufficient in the current PowerPC implementation. The number of integer registers required depends on the implementation of individual RT instructions. Computing the exact number would have consisted of tracing the program flow in the frontend and backend to compute the number of registers required for each bytecode. Note that if the allocator runs out of temporary registers, saved registers are used.

A fixed number of registers are preallocated and they refer to the same quantities at all time. These include registers for global⁴ values often used such as the Java locals pointer and the Java operand stack pointer. Several temporary registers used frequently by the frontend are also preallocated to a specific register, even though the meaning of their value changes. By preallocating them, we avoid frequently invoking methods to get and release them.

The frontend can specify which registers are *important*. On architectures with a large register set, this hint can simply be ignored. On architectures with a limited number of registers such as the x86 this hint is used to map important values to real native registers permanently rather than using locations on the native stack. In the next section we discuss issues encountered with the non-RISC x86 architecture.

8.4 x86: Problems and Solutions

RT code is based on a RISC-like model. Implementing RT code for a RISC architecture is simple and the implementation is efficient. However, implementing RT code for a CISC architecture brings up several issues. We summarize problems encountered

⁴Global to the compiled method.

on the x86 architecture and then we describe their solutions.

- **Limited number of registers:** The x86 has only 8 GPRs where one is the stack pointer. Our implementation uses an additional register as a frame pointer. This reduces to 6 the number of GPRs for storing variables and performing computations.
- **2-operand instructions:** Several x86 instructions have two operands rather than three, the destination operand serving also as a source operand. For example, the addition on the x86 has the format `add rd,rt` ($rd = rd + rt$) rather than the 3-operand format: `add rd,rs,rt` ($rd = rs + rt$).
- **Register specific usage:** On the x86 architecture some register operands are implicit, that is, they are not specified by the instruction encoding. A typical example is the integer division where the 64-bit dividend must be located in the register pair EDX:EAX (the 32-bit higher word in EDX and the 32-bit lower word in EAX). The divisor is specified by an explicit register (or memory) operand. The quotient and remainder are put in register EAX and EDX respectively.
- **Floating point register stack:** The x86 architecture has an x87 *floating point unit* (FPU) with 8 FPRs. The FPRs are addressed by floating point instructions in a stack-like manner. A top pointer is kept in a status register. Load instructions push values on the register stack and store instructions pop values from the stack. Floating point instructions operate on the top element and elements relative to the top.

We delay the presentation of the solution of the limited number of registers after the discussion of the other three solutions.

8.4.1 2-operand Instructions

3-operand RT instructions are expressed in terms of 2-operand native instructions by reorganizing the order of the operands and by adding necessary register moves.

In the implementation of the baseline compiler, it is likely that both `rs` (first source operand) and `rd` (destination operand) are the same. In that case, the code emitted is efficient, as the 3-operand instruction actually has a 2-operand form.

8.4.2 Register Specific Usage

The baseline compiler frontend currently has no knowledge on instructions with implicit register operands or on instructions with register usage restrictions. This problem is solved in a simplistic way: register moves are added as necessary to bring the operands in the correct registers. If necessary, values in target registers are saved on the native stack before the operation and are restored after.

8.4.3 Floating Point Register Stack

The register stack in the x87 FPU does not offer a big challenge for the baseline compiler. For floating point operations, values are read from the Java operand stack and the result is written back to the operand stack. At most two FPRs are required and only the top two elements of the register stack are used. The values are loaded on the register stack, the computation is performed and the result is stored back in memory.

8.4.4 Limited Number of Registers

The limited number of GPRs is the most difficult problem to solve. The limitation can be accommodated by introducing *virtual registers*. To the frontend, virtual registers appear as any other registers, however, their data is stored on the native stack rather than in real (or native) registers. The use of virtual registers was proposed as a solution by Engler [Eng96], but not implemented.

Figure 8.7 illustrates the native stack layout used for compiled code. In addition to the space reserved for the 3 callee-saved real registers, sufficient space is allocated for a predefined fixed number of virtual registers. These registers are accessible by

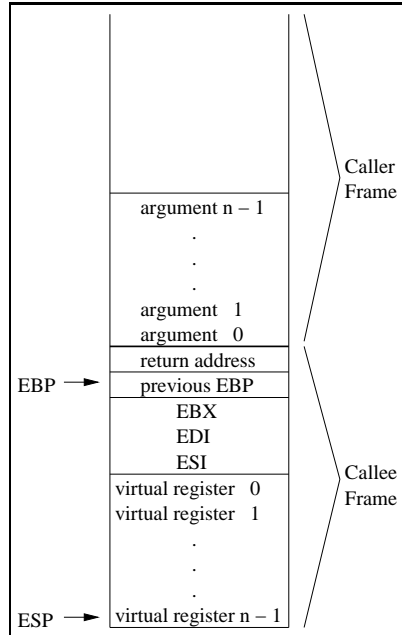


Figure 8.7: Native Stack Layout on x86

displacement addressing from the frame pointer (register EBP). Most x86 instructions have variants with memory operands. It is possible, with a clever selection of instructions, to achieve performance comparable to a RISC platform.

The use of virtual registers is completely transparent to the frontend. The implementation of the RT instructions distinguishes between real and virtual registers by their assigned register number. Real registers are numbered from 0 to 7 matching the native architecture numbering whereas larger numbers are used for virtual registers. Any combination of real and virtual register operands can be used with RT instructions.

Register Allocation Scheme

For most x86 instructions not all operands can be memory operands, some of them must be in registers. To achieve good performance, it is important to dedicate real registers to some specific variables to avoid excessive register loads and stores. Table 8.5 summarizes the use of real registers in compiled code. ESP and EBP are the

Register	Usage
ESP	native stack pointer
EBP	native frame pointer
EAX	scratch register, return value
ESI	Java local pointer
EDI	Java operand stack pointer
ECX	tmp1
EDX	tmp2
EBX	tmp3

Table 8.5: Register Usage on x86

stack pointer and the frame pointer respectively. The frame pointer is mostly used to access efficiently the virtual registers. EAX is used as scratch register and for return values. ESI and EDI are used for the Java local pointer and Java operand stack pointer respectively. Since the baseline compiler mimics the Java stack operations, these two pointers are used extensively. Finally, the three mostly used registers for computations (known to the frontend as `tmp1`, `tmp2` and `tmp3`) are assigned to registers ECX, EDX and EBX respectively.

8.5 Architecture Independent Functionality Provided

The backend provides some architecture independent functionality to handle common tasks for all architectures and to facilitate porting. These tasks include maintaining code arrays, performing branch patching and collecting information such as data structure offsets and other architecture-specific information. We now discuss each of these.

8.5.1 Maintaining Code Arrays

Maintaining code arrays consists of creating and resizing as necessary Java arrays that will contain the code emitted. This functionality is implemented in the `Architecture` class and it is made available through inheritance. Currently there is support for

`byte` arrays and `int` (or word) arrays. Architectures that encode instructions as 32-bit words such as PowerPC and SPARC use the word array. Architectures with a variable-size encoding such as x86 use the byte array.

In debugging mode, the backend can maintain string arrays containing information such as the equivalent assembly code. It is up to the implementor of a platform to provide most of the information. Some high-level information is also added by the frontend.

All maintained arrays are dynamically resized. No explicit check for overflow is done. Instead, the implementation relies on the Java array bounds exception mechanism to know when they should be resized. This is particularly useful for the one-pass baseline compiler as the code array size required does not need to be precomputed. In VCODE, it was up to the client code (that would correspond roughly to a compiler frontend) to allocate an array large enough to hold the emitted code.

8.5.2 Architecture Independent Branch Patching

The backend provides an architecture-independent branch patching mechanism. The same framework is also used to patch addresses in jump tables. Each architecture has to implement the abstract method: `int reserveBranchSpaceAmount(int branchType)`. This method returns amount of space, in terms of array elements, to set aside for forward branches. There is an extra cost for architectures where the needed space varies depending on the target. For a branch of a particular type, the largest amount of space ever required needs to be reserved. If it is not all used at patching time, nops are added to fill the unused portion. In the compiler, branch targets are abstracted by the use of `Label` objects. A label is set at the current point in the generated code by invoking the `setLabel` method located in `Architecture`. This label can be referred to in branch RT instructions. Branch RT instructions are available in immediate value form and label form. The immediate form takes an offset relative to the beginning of the code array as target. This form must be implemented for each architecture. On the other hand, the label variant is part of the framework. For backward branches, it invokes the immediate form immediately and for forward

```
1 Label label = new Label("else");
2 arch.ifeqI(tmp1, label);
3 arch.returniI(3);
4 arch.setLabel(label);
5 arch.returniI(4);
```

Figure 8.8: RT Code Segment Using a Label

branches it records information for patching the code later.

Figure 8.8 shows a small RT code fragment that makes use of labels. In line 1 the label object is created. A descriptive name (“else” in this case) can be given to help debugging. In line 2, the RT instruction compares `tmp1` against zero. If it is found to be equal, a jump to the label set in line 4 is done, otherwise execution continues at the next instruction. Since it is a forward branch, the target offset of the branch is computed during the patching phase.

Patching entries of jump tables are done in a similar way as branches. Two methods are available to add entries: `rawWord(int word)` and `rawLabel(Label label)`. The former writes its argument verbatim in the code array and the latter writes the address corresponding to the specified label either immediately if known, or later otherwise.

8.5.3 Platform Independence of the Memory Layout of Data Structures

The memory layout of data structures (`struct` and `union` types) in C code is architecture dependent. The compiled code needs to access and update some data structures used in the virtual machine. The memory offset of each member from the start of the structure varies from architecture to architecture, though it is constant within a single architecture. For convenience, maintenance and portability, these offsets are not hard coded in the compiler. A data file specifies relevant information. When the virtual machine is built, this file is preprocessed and used to generate source code that is then compiled with the virtual machine. This code prints out all the required information when SableVM is run with the `-C dump` command line option. This information is then used to generate a Java class containing all the field offset constants. These

constants can then be used in the compiler code.

In addition to member offsets, information on the SableVM configuration such as the used object layout (traditional or bidirectional) is provided to the compiler in a similar way. Some information such as addresses of functions changes with the slightest change in the source code. To avoid rebuilding the compiler after small modifications in the virtual machine, that information is passed to the compiler at runtime during its initialization. This process is also fully automated; adding a new hook function consists of adding a new entry to the data file and rebuilding the software.

Note that although the SableJIT compiler is written in Java, the compiler built is dependent on a specific platform and a specific SableVM configuration. The compiler jar file cannot be shared between different platforms or configurations. SableJIT does however have an option to provide all the configuration information at runtime in order to remove the dependency on configurations.

8.6 Summary

In this chapter, we have presented the compiler retargetable backend. We have described the RT code model and how it is implemented. We have seen how registers are allocated, their usage, and their requirements. We have studied the case of the non-RISC architecture x86 and how we were able to make it fit well with our RISC-based model. We have concluded the chapter by presenting various functionality provided by the backend to reduce the amount of work required to port and maintain the compiler code.

Chapter 9

Porting Strategy and Experience

In this chapter, we present a strategy for porting SableJIT to new platforms. We start by describing the suggested porting approach. Then we discuss our experience gained in retargeting SableJIT to the Solaris/sparc platform. Finally, we estimate the amount of effort required to add a new platform.

9.1 Suggested Porting Strategy

Since the compiler works in conjunction with the interpreter, the first obvious step to support a new platform is to port the various interpreters of SableVM to that new platform. The switch interpreter is quite portable whereas the inlined interpreter requires some additional work. If the target platform does not support the inlined interpreter or does not support signal-based exceptions, then it is still possible to port SableJIT without these features. Luckily, SableVM already supports a variety of platforms and it is likely that most work will be in SableJIT. For porting SableJIT to a new architecture, knowledge of the new architecture should be sufficient. It should not be necessary to know about bytecode instructions, object layouts or other virtual machine internals. Note that this property also makes the retargetable backend quite independent from some design decisions in SableVM. For example, changing the object or array layout does not require architecture specific changes, as all changes are located in the compiler frontend. Porting SableJIT requires minimal

9.1. Suggested Porting Strategy

Primitive Function	Description
<code>void *</code> <code>get_native_frame_pointer</code> (<code>_svmt_JNIEnv *env</code>)	Returns some pointer (not necessarily a frame pointer) that allows the native PC to be retrieved from stack frames by calling repetitively <code>_sjf_get_next_native_pc</code> .
<code>void *</code> <code>get_next_native_pc</code> (<code>void **frame_pointer</code>)	Returns the native PC (i.e. the return address) stored in the frame identified by <code>frame_pointer</code> . Also updates the pointer argument for the next call.
<code>void *</code> <code>get_fault_address</code> (<code>int signo,</code> <code>siginfo_t * info,</code> <code>void *context</code>)	Returns the address of the instruction that caused a SIGSEGV, SIGFPE or SIGTRAP.
<code>void get_exactitude</code> (<code>jint signal_code,</code> <code>jboolean *is_exact,</code> <code>jint *exactitude_offset</code>)	Provides some information on how close the fault address can be from the addresses stored in the map.

Table 9.1: Architecture Dependent Primitive Functions in the Runtime

work in the runtime and more extensive work in the compiler. We now present the work required and a strategy to accomplish it.

9.1.1 Porting the SableJIT Runtime

The SableJIT runtime is written in portable C code. Most of the code is architecture independent. The few architecture-dependent code segments have been isolated in two well-commented files: `jit_system.h` (about 55 lines) and `jit_system.c` (about 190 lines). In file `jit_system.h`, the code array width is specified: either words or bytes corresponding to the Java `int []` type and `byte []` type respectively. In `jit_system.c`, four small primitive functions need to be implemented for each architecture. Table 9.1 lists their prototypes with a brief description. They require a few lines of assembly or C code. Functions `_sjf_get_native_frame_pointer` and `_sjf_get_next_native_pc` are used to walk up the native stack during garbage code collection (see chapter 7). Functions `_sjf_get_fault_address` and `_sjf_get_exactitude` are used in signal-based exception handling (see chapter 6).

9.1.2 Porting the SableJIT Compiler

Porting the compiler demands more effort than the runtime. The amount of effort greatly varies depending on whether the compiler is being retargeted to a new operating system on an already supported processor architecture or if we are targeting a new processor architecture.

Retargeting a New OS on a Supported Processor Architecture

Retargeting SableJIT to a new operating systems on a supported processor architecture requires little work. The already implemented RT instruction set for that architecture is reused. However, the ABI could differ from the currently supported ones. Two options are available to port the backend to a new ABI:

1. Implementing the full ABI for the target OS as a new class.
2. Modifying an existing ABI implementation to add support for your OS.

As mentioned in earlier chapters, all the ABI-specific code is located in one of the classes implementing the ABI interface. The backend is designed to easily support different ABIs for a particular processor architecture. The choice between a full implementation of a new ABI or a modification of an existing ABI class depends on the number of differences between the target ABI and an existing one. It is important to note that only a subset of the ABI is used by the compiled code. This could further reduce the amount of differences. Several simplifications are made:

- Compiled code does not call functions with more than eight arguments.
- Compiled code does not receive or pass structures by value. A pointer to the structure is always used.
- Compiled code does not call functions with mixed floating point and integer arguments. All arguments of a function call are of the same type: either of integer type (32-bit or 64-bit) or of one of the floating point types.

- Compiled code does not call variable arguments (a.k.a varargs) functions.

By taking into account these simplifications, operating systems with mostly similar ABIs could become identical for our purposes. This is indeed the case between the System V ABI and the Mach-O ABI on the PowerPC platform. The set of differences is small enough that creating a new ABI class was not justified. Both implementations were merged in the same class.

Retargeting a New Processor Architecture

Retargeting to a new processor architecture involves more work as the full RT instruction set must be implemented. Several means are available to help the development process. Some instructions have a default implementation in terms of others. This helps incremental development as the full RT code does not need to be implemented at once then tested as a whole. Isolating bugs is made easier. Also, the fact that the compiler can recover from compilation errors by falling back to interpretation helps running full series of test cases without interruption due to bugs. In particular, some methods or functionality could be temporarily implemented in a dummy way by adding a statement that throws a `NotImplementedException` exception. This delaying of implementation further helps incremental development and testing. Incremental development is useful as the code in development can be kept relatively stable and new code can be tested as it is added.

The robustness of the compiler helps in that test cases provided with SableJIT can be used early in the development process. Tests failing to compile are simply interpreted (and hence they are skipped) instead of terminating the virtual machine. Progress made can be observed by noting the ratio of successful test cases, as implementation and debugging proceed. This strategy was used in the initial SableJIT development as well as the port to the SPARC architecture, and it proved to work well.

Finally, when SableJIT is run in testing mode, it is possible to explicitly specify which methods to include for compilation. The options used to do so are described in section A.3 in the appendix. This control provides further helps in isolating bugs

9.1. Suggested Porting Strategy

to a single or small number of methods.

Retargeting to a new architecture can be divided into the following steps:

1. Implement basic integer and branch instructions.
2. Start testing and debugging the backend using the provided test cases.
3. Implement the ABI.
4. Start testing and debugging full method compilation.
5. Implement, test and debug remaining instructions such as floating point operations, type conversions and jump table primitives.
6. Implement, test and debug advanced functionality such as signal-based exceptions.

At the end of step 1, a basic code generation system is built. It is sufficient to compile simple RT code sequences (as opposed to full Java methods) to native code. In step 2, the test cases provided in class `sablejit.arch.ArchitectureTest` are used to test if the implemented RT instructions follow the specification. The testing code creates small sequences of RT instructions and executes them. This happens entirely in the backend. After step 3 is done, a sufficient part of the RT instruction set has been implemented to compile full, simple, Java methods. In step 4, testing the complete compilation process begins. These test cases which are located in the class `sablejit.CompilerTest` consist of compiling the bytecode of small Java methods to native code. The complete compilation system is tested: the runtime, the frontend, and the backend. In step 5 and 6, the implementation is completed with the remaining RT instructions and with the addition of signal-based exception handling.

This strategy was followed to retarget SableJIT to the Solaris/sparc platform and it worked well. Once the implementation was completed and debugged using the provided test cases, the compiler was robust enough to survive compilation of most Java code including the compiler and the Java class libraries, with the exception of very few library methods.

9.2 Evaluation of Porting Effort

In this section, we evaluate the required amount of effort to port SableJIT to a new platform. The platform is Solaris/sparc and the main developer for this task was Christian Arcand, an undergraduate student in computer science at UQAM. Arcand implemented most of the SPARC backend. The author helped Arcand in debugging, testing, and completing the implementation of advanced features such as signal-based exceptions.

It is important to take into account two factors while evaluating this effort. First, neither Christian nor the author had previous knowledge of the SPARC architecture. Second, SableJIT as well as SableVM were still under development during that period of time. Regularly incorporating new changes into the SPARC branch required some effort from time to time. Also, at the beginning of the project, SableVM had to be ported to Solaris. Small issues needed to be solved and several GNU software tools needed to be installed.

The first thing we noted from that experience is that the architecture-dependent code is well isolated. No changes other than adding some configuration information were required in the compiler frontend. The work mainly consisted of adding two classes: the `Sparc` class as a subclass of `Architecture` and the `SVR4SparcABI` class implementing the ABI interface.

Adding and testing support for signal-based exceptions mainly consisted of implementing two small primitive functions in the runtime as well as RT instructions for traps. A small refactoring of the C code in the runtime was required, though, as a previously made assumption was not holding on SPARC¹. The runtime code was refactored and the varying code was relocated as a primitive function in file `jit_system.c`. Signal-based exceptions were implemented in about two hours, including learning about hardware traps on the Sparc platform, coding, testing and debugging.

¹It was assumed that the first instruction of the trapping code generates the trap, however, on SPARC, the first instruction is a *compare* and it is actually the second instruction that generates the trap based on the result of the comparison.

9.2.1 Evaluation in Man Hours

We estimated that it took a total of 5 to 6 weeks (200-240 man hours) to learn the architecture, install the necessary software tools, implement, test and debug the SPARC backend. The actual work was done part-time, and spread over several months.

9.2.2 Evaluation in Lines of Code

We present a count of *lines of code* (LOC) for different components of SableJIT to provide a general idea of the relative proportions of architecture-dependent code and architecture-independent code. The LOC metric is a count of all lines including blank lines and comments. The reason to include these resides in the usage of GNU M4 macros. Most of these macros are defined inside comments. If we had used tools that exclude comments from the counts, these definitions would not have been reflected in the counts. Files with M4 macros are expanded to either plain C or plain Java source code that is then passed to the compiler. We provide numbers for files before and after M4 expansion.

Table 9.2 shows the LOC value for various components of SableJIT. The first and second column show the LOC before and after macro expansion respectively. The automatically generated file `SableVMConstant.java` is excluded from the counts. The files containing test cases are also excluded. The percentage values are in terms of the total line count (the *All* row). The *All* row consists of all the source code of SableJIT. The second row, *Compiler*, is the compiler only, that is, the Java code. The third row, *Runtime*, is the runtime written in C and compiled with virtual machine code. The fourth row, *Base*, is the compiler base system, that is, all the compiler code shared among all architectures. The next three rows correspond to the architecture dependent backend code for the x86, PowerPC, and SPARC architectures respectively. The last two rows sum up the architecture-dependent and architecture-independent code respectively. The architecture dependent code for all three architectures represent 25.0% and 18.5% before and after macros expansion, respectively.

Source	LOC before expansion	LOC after expansion
All	33657 (100.0%)	56019 (100.0%)
Runtime	11401 (33.9%)	20873 (37.3%)
Compiler	22256 (66.1%)	35146 (62.7%)
Base	13856 (41.2%)	24776 (44.2%)
x86	3935 (11.7%)	5462 (9.6%)
ppc	2365 (7.0%)	2931 (5.2%)
sparc	2100 (6.2%)	1977 (3.5%)
dependent	8400 (25.0%)	10370 (18.5%)
independent	25257 (75.0%)	45649 (81.5%)

Table 9.2: Lines of Code for SableJIT

9.3 Summary

In this chapter, we have presented the porting process and experience. We have started by describing a porting strategy and the work necessary to port to a new platform. Then, we have described our experience in porting SableJIT to the Solaris/sparc platform. Finally, we have evaluated the required amount of work.

Chapter 10

Experimental Results

In this chapter we present experimental results. We start by describing the testing environments and the benchmarks used. Then, we present various performance results.

10.1 Test Platforms

The experiments were conducted on three platforms that we designate as Linux/x86, Linux/ppc, and Solaris/sparc. The Linux/x86 platform is an AMD Athlon 1250MHz with 512MB of RAM running Debian GNU/Linux 3.0 with kernel 2.4.20. The Linux/ppc platform is a PowerPC G4 533MHz with 1152MB of RAM also running Debian GNU/Linux 3.0 with kernel 2.4.20. Finally, the Solaris/sparc platform is a Sun Enterprise 3500 with 6 CPUs clocked at 400MHz and 3840MB of RAM running Solaris 8.

For these experiments, we used SableVM version 1.1.6 (Classpath 0.10) built with SableJIT support (subversion revision 2830). Unless indicated otherwise, SableVM and SableJIT are built with default options. In particular, signal-based exception handling is used and methods are considered for compilation on their second invocation.

10.2 Benchmarks

We used the SPECjvm98 [SPE] benchmarks that are well-known in the literature for measuring performance. Results obtained should not be treated as official SPEC results as the SPEC methodology was not used to run the benchmark programs. Instead, custom-written wrapper scripts were used to run them and to collect the results.

We also used SableCC [Saba] as an additional benchmark for its high use of object-oriented features. SableCC is a *compiler compiler* developed by the Sable Research Group at McGill. It is used in the construction of compilers and interpreters in Java. SableCC takes as input a grammar and then generates a lexer, a parser, and a set of classes for building and traversing abstract syntax trees. We used version 2.18.2 and we fed as input the grammar of *Simple C*¹ available from the SableCC website.

10.3 Results

10.3.1 Interpreters and Compilers

We ran our benchmarks to compare the various combinations of interpreters and compilers. It is important to remember that our baseline compiler is very naive and, as such, does not do much more than the inline-threaded interpreter other than removing additional instruction dispatch overhead.

Overall Performance

Table 10.1, 10.2, and 10.3 show the results obtained on the Linux/x86, Linux/ppc, and Solaris/sparc platforms respectively. All times shown are in seconds. Times are computed as the average of the sum of user and system time over three runs. The first three columns of numbers show the results for the switch, direct-threaded, and inline-threaded interpreters respectively. These interpreters are compiled without the SableJIT runtime. The last three columns show the results when just-in-time

¹Command used: `java org.sablecc.sablecc.SableCC -d work simplec.sablecc`

10.3. Results

Benchmark	switch	direct	inlined	switch/jit	direct/jit	inlined/jit
compress	361.49	273.25	185.28	91.69 (3.94)	88.45 (3.09)	85.57 (2.17)
db	148.70	120.30	100.85	69.70 (2.13)	69.49 (1.73)	68.53 (1.47)
jack	43.15	36.97	34.44	38.29 (1.13)	37.85 (0.98)	37.72 (0.91)
javac	105.85	85.98	79.01	92.44 (1.15)	90.69 (0.95)	89.78 (0.88)
jess	77.90	63.76	57.30	44.77 (1.74)	45.40 (1.40)	46.09 (1.24)
mpegaudio	317.08	217.86	166.79	114.86 (2.76)	114.35 (1.91)	113.35 (1.47)
mtrt	96.45	77.70	72.57	93.57 (1.03)	92.95 (0.84)	94.74 (0.77)
raytrace	93.15	75.44	69.66	92.30 (1.01)	91.76 (0.82)	92.15 (0.76)
sablecc	91.47	73.66	67.53	70.52 (1.30)	64.56 (1.14)	61.80 (1.09)

Table 10.1: Performance Results for Linux/x86

Benchmark	switch	direct	inlined	switch/jit	direct/jit	inlined/jit
compress	393.64	287.56	220.05	172.20 (2.29)	168.01 (1.71)	165.54 (1.33)
db	160.42	126.19	111.86	104.05 (1.54)	104.40 (1.21)	101.73 (1.10)
jack	55.43	46.60	48.54	53.79 (1.03)	53.51 (0.87)	54.19 (0.90)
javac	131.97	108.59	103.19	118.71 (1.11)	116.20 (0.93)	112.93 (0.91)
jess	93.45	75.97	70.60	67.78 (1.38)	67.05 (1.13)	67.29 (1.05)
mpegaudio	350.92	253.35	240.95	142.09 (2.47)	141.87 (1.79)	142.18 (1.69)
mtrt	116.77	92.41	89.19	115.03 (1.02)	115.27 (0.80)	113.72 (0.78)
raytrace	111.54	89.18	84.81	112.25 (0.99)	111.69 (0.80)	110.79 (0.77)
sablecc	115.32	95.68	88.95	103.55 (1.11)	98.03 (0.98)	93.45 (0.95)

Table 10.2: Performance Results for Linux/ppc

Benchmark	switch	direct	inlined	switch/jit	direct/jit	inlined/jit
compress	912.36	662.90	476.00	389.10 (2.34)	373.17 (1.78)	380.82 (1.25)
db	343.27	252.90	222.83	204.21 (1.68)	201.15 (1.26)	220.47 (1.01)
jack	117.13	94.21	92.44	102.99 (1.14)	101.58 (0.93)	102.67 (0.90)
javac	277.13	216.45	198.69	232.77 (1.19)	224.67 (0.96)	219.55 (0.90)
jess	194.26	150.68	140.09	142.82 (1.36)	145.02 (1.04)	142.87 (0.98)
mpegaudio	820.28	538.35	400.94	391.44 (2.10)	389.28 (1.38)	390.31 (1.03)
mtrt	225.04	171.82	159.96	193.81 (1.16)	192.04 (0.89)	195.06 (0.82)
raytrace	217.82	169.18	162.72	188.71 (1.15)	187.12 (0.90)	188.30 (0.86)
sablecc	235.23	182.58	167.71	206.62 (1.14)	190.22 (0.96)	179.77 (0.93)

Table 10.3: Performance Results for Solaris/sparc

compilation is used. Numbers in parentheses are speedups achieved by the interpreter/compiler combination over the same interpreter without compilation support.

For some benchmarks, the compiler improves the performance while on others it worsens it. Adding compilation support to the switch interpreter always leads to improvements in execution time with the exception of *raytrace* on Linux/ppc which is slowed down by 1%. The best result was obtained for *compress* on Linux/x86 with a speedup of 3.94. For the direct-threaded and inline-threaded interpreter, adding compilation support improves the execution time of about half the benchmarks. We see improvements in *compress*, *db*, *jess* (except for inline-threading on SPARC), *mpe-gaudio*, and *sablecc* (on x86 only). Benchmarks where performance decreases are *jack*, *javac*, *mtrt*, *raytrace*, *jess* (inline-threading on SPARC), and *sablecc* (on PowerPC and SPARC).

Similar performance patterns are obtained on all three platforms. Note also that when compilation is enabled, the interpreter that is used in combination with the compiler does not seem to affect much the performance. For some benchmarks, the direct/jit combination performs better than the inlined/jit combination. This could be explained in part by the compilation of inline-threaded code being more expensive than the compilation of direct-threaded code.

It is interesting to note that the non-RISC x86 architecture performs better, on average, than both RISC architectures. This result is quite surprising considering that our model is RISC-based and makes heavily use of registers. We could conclude that the usage of the native stack as register space does not have a major impact on performance on the x86. The better performance results obtained on PowerPC compared to SPARC could be explained by the fact that less development effort was invested in the SPARC backend implementation. Features such as branch delay slots are not yet fully exploited.

Compilation Times

Table 10.4 shows the compilation times on the Linux/x86 platform for each interpreter type. All times shown are in seconds. Columns identified by *Comp* and *Total* are the

Benchmark	switch/jit		direct/jit		inlined/jit	
	Comp	Total (%)	Comp	Total (%)	Comp	Total (%)
compress	2.15	91.05 (2.36)	2.02	88.32 (2.29)	2.07	84.87 (2.44)
db	2.20	70.64 (3.11)	2.07	69.44 (2.98)	2.07	68.05 (3.04)
jack	6.04	39.30 (15.37)	5.96	38.37 (15.53)	6.01	39.62 (15.17)
javac	12.01	93.04 (12.91)	12.21	91.43 (13.35)	12.47	91.27 (13.66)
jess	3.94	45.08 (8.74)	3.85	46.50 (8.28)	3.88	44.91 (8.64)
mpegaudio	2.45	115.70 (2.12)	2.29	114.20 (2.01)	2.30	113.48 (2.03)
mtrt	3.77	93.64 (4.03)	3.75	92.89 (4.04)	3.75	93.21 (4.02)
raytrace	3.00	92.59 (3.24)	2.86	90.63 (3.16)	2.88	90.44 (3.18)
sablecc	5.42	71.53 (7.58)	5.39	67.01 (8.04)	5.46	64.58 (8.45)

Table 10.4: Compilation Times for Linux/x86

compilation times and total execution times, respectively. Numbers in parentheses are the percentages of the total execution time spent in compilation, that is, $\frac{Comp}{Total} \times 100$.

Compilation times for the direct/jit combination are smaller than the switch/jit combination for all benchmarks except *javac*. Compilation times for the inlined/jit combination are smaller than the switch/jit combination for all benchmarks except *javac* and *sablecc*. Since our compiler is written in Java, its performance is usually improved if the interpreter used performs better. However, the compilation times for the inlined/jit are slightly higher than the direct/jit except for two benchmarks where they are equal. The inline-threaded to switch code conversion cost is likely to be more expensive than the direct-threaded to switch code conversion and this would explain the higher compilation times for the inline-threaded.

Some trends can be seen between the proportion of time spent compiling and the performance of SableJIT. In benchmarks where a large proportion of time is spent compiling, the performance is poor. This can be observed for *jack* and *javac* where the compilation accounts for up to 15.53% and 13.66% of the execution time, respectively. In particular, for these benchmarks, the direct-threaded and inlined-threaded interpreter outperform our compiler. Benchmarks where our compiler performs very well, such as *compress* and *mpegaudio*, have low compilation times. The compilation time accounts for up to 2.44% for *compress* (inlined/jit) and up to 2.12% for *mpegaudio* (switch/jit). Note that the two benchmarks where our compiler performs

the poorest, *mtrt* and *raytrace*, are major exceptions to this pattern. Compilation times for these account only for at most 4.04% (*mtrt*, *direct/jit*) and 3.24% (*raytrace*, *switch/jit*) of the total execution time.

General Discussion

Profiling in SableJIT is still quite primitive. A simple counter is used at method entry. By compiling a method on its second invocation, we might compile too much, that is, we might compile methods not contributing significantly to the execution time. On the other hand, if the compilation threshold is too high, we might miss the opportunity of compiling infrequently invoked methods that contribute significantly to the execution time. A typical example of such method is a method containing loops that do many iterations. Better profiling as well as the addition of compilation entry points within a method body are currently under development.

Other factors than the compilation time could explain the lower performance obtained on some benchmarks. First, SableJIT does not have an optimizing compiler. Most improvements in execution time are gained from the removal of instruction dispatch overhead. Since the direct-threaded and inline-threaded interpreters remove some of that overhead, the performance gains obtained from compilation are much lower for these interpreters. For some benchmarks, the removal of the remaining overhead might have little effect and any speedup gained could be outweighed by other factors such as the compilation time. Also, it might be possible that the GCC generated code outperforms our hand-coded architecture-independent RT implementation. This would especially be the case for method invocation bytecodes as their implementation consists of large blocks of code. Finally, some overhead could be added by compiled preparation sequences. The interpreter patches the slower instruction variants after their first execution. The compiled code, however, does not patch them. Instead, it profiles them with the hope of recompiling them later. Therefore, the compiled code does not take immediate advantage of a faster implementation for such bytecodes. This summarizes the areas where performance could be studied and improved in future development.

Benchmark	switch	switch/jit	direct	direct/jit	inlined	inlined/jit
compress	393.13	397.90 (0.99)	287.18	290.84 (0.99)	219.55	220.68 (0.99)
db	160.77	168.35 (0.95)	126.05	130.12 (0.97)	112.14	112.98 (0.99)
jack	55.40	56.31 (0.98)	46.66	47.47 (0.98)	48.57	49.89 (0.97)
javac	131.98	134.33 (0.98)	108.50	109.98 (0.99)	103.06	104.50 (0.99)
jess	93.41	95.10 (0.98)	75.99	77.73 (0.98)	70.45	71.69 (0.98)
mpegaudio	350.90	352.49 (1.00)	253.30	251.82 (1.01)	241.07	242.33 (0.99)
mtrt	119.44	118.06 (1.01)	93.70	94.31 (0.99)	88.78	90.14 (0.98)
raytrace	111.51	114.46 (0.97)	89.15	90.93 (0.98)	84.83	87.36 (0.97)
sablecc	115.35	118.02 (0.98)	95.93	97.31 (0.99)	88.92	91.53 (0.97)

Table 10.5: Runtime Overhead in Interpreter-Only Mode

	Without JIT	With JIT	Difference
Switch	545176	661828	116652 (21.3%)
Direct	565476	686628	121152 (21.4%)
Inlined	602048	735780	133732 (22.2%)

Table 10.6: Size of Executables

Conclusion

We can conclude that we have good results, though there is a lot of room for improvements. In particular, we think that with the addition of an optimizer, better profiling, and better control over compilation; our retargetable JIT would improve significantly performance-wise.

10.3.2 Interpreter-Only Mode

In this section, we measure the overhead of the interpreter-only mode over SableVM compiled without SableJIT support. In the interpreter-only mode, the compiler is disabled at all time. It is not even bootstrapped.

All experiments in this section were conducted on Linux/ppc. Table 10.5 shows the overhead of the interpreter-only mode. The overhead ranges from 0% (unnoticeable) to 5%. For most benchmarks, it is in the range 1-2%. The 5% overhead is rather exceptional and it occurs only with the *db* benchmark on the switch interpreter. The overhead in interpreter-only mode comes mostly from the selection of the appropriate

INVOKE instruction during method preparation (see section 4.2). For each INVOKE instruction, a check is done to see if the compiler is enabled in order to choose between the regular interpreter INVOKE instruction or an INVOKE instruction with a compilation entry point.

Table 10.6 shows the total size (in bytes) of the binaries of SableVM. The second and third columns are without and with SableJIT runtime, respectively. The fourth column is the difference in size. The SableJIT runtime increases the total size of the binaries from 21.3% for the switch interpreter up to 22.2% for the inline-threaded interpreter.

10.4 Summary

We have seen that our baseline compiler improves over the interpreters for most benchmarks. We have described several areas of future improvements. We have also studied the overhead of the runtime when using the interpreter-only mode and we have concluded that the runtime does not add much cost to the interpreter when the JIT is disabled.

Chapter 11

Related Work

In this chapter we present related work. The chapter is separated in two parts. In the first part, we cover work on retargetable code generators. In the second part, we study the retargetability aspect of various Java virtual machines and JITs.

11.1 Retargetable Code Generators

11.1.1 Code-Generator Generators

Code-generator generators are tools that help developers in the construction of code generators. Most of these tools are *Bottom-Up Rewrite Systems* (BURS [PLG88]). They take as input a grammar consisting of a set of rules that describe how code should be generated for each expression subtree. Examples of such tools are `iburg` [FHP92,ibu] and `JBurg` [JBu]. `iburg` is a *Bottom-Up Rewrite Machine Generator* (BURG) used by LCC [LCC], a retargetable (static) C compiler. A modified version of `iburg` is also used in Jikes RVM [AAB⁺00], a Java virtual machine described in section 11.2.3. We limit our description to `JBurg` as other systems essentially work the same.

`JBurg` takes as input a grammar consisting of rules. Each rule matches an expression subtree and has an associated number of actions. For code generators, these

actions are simply statements emitting the native code corresponding to the subtree node. Basically, each rule tells how code should be generated for a subtree node.

The grammar is allowed to be ambiguous, that is, two or more native instruction sequences could be derived (or generated) from the same expression tree. This ambiguity is useful in that there is usually more than one way to generate native code for some expression. A cost is assigned to each rule. When the code generator is built, dynamic programming techniques are used to precompute data about derivations and their cost.

Code generation is performed by traversing the expression tree twice. The tree is first traversed bottom-up to compute a derivation of minimal cost. In the second pass, actions associated with each rule used in the derivation are executed. For code generators, these actions consist of emitting the code.

Porting the code generator to a new architecture consists of writing a new grammar with a new set of actions. This new grammar describes how expression trees should be converted to code for that new architecture.

SableJIT currently does not use any code-generator tools. We would like to consider the use of such tools in the future. These could be used to generate code generators that convert a high-level *intermediate representation* (IR) to low-level IRs such as RT code or native code.

11.1.2 VCODE

Our work is mostly based on the work of Engler on VCODE [Eng96]. VCODE is designed to be a retargetable, very fast dynamic code generator. It generates native code in-place, that is, directly in its final destination in memory, without building any IR data structure. The goal of VCODE is to generate good code *fast* rather than taking more time to generate optimized code.

VCODE defines a virtual RISC-based architecture. The client code¹ generates

¹The client code is the program that makes use of the VCODE backend. For compiler systems, it corresponds to the compiler frontend.

native code by using C macros and functions that correspond to VCODE instructions. Adding a new architecture backend to VCODE consists of implementing each VCODE instruction in terms of native instructions for the target architecture. On RISC architectures, most VCODE instructions map trivially to one or two RISC instructions. It is therefore easy to add new RISC architectures. CISC architectures, however, require more work.

It is important to note that VCODE is not a full compiler system but could be used in a compiler backend. Tasks such as register allocation are mostly left to the client code. VCODE defines two classes of registers: *temporary* (caller saved) and *persistent* (callee saved). The client can request and release registers, however, if VCODE runs out of registers, it is up to the client to handle this situation, by performing register spills for example.

VCODE has some limitations. First, it is up to the client to allocate sufficient memory to hold the generated code. As the generated code is not relocatable, the framework cannot reallocate a larger code array. Second, at most one function at a time can be generated, therefore making it unsuitable for multithreaded environments. It is mentioned that this limitation could be removed in a future release. A third limitation is that types in VCODE are C types. This makes the size of some types, such as `long`, dependent on the underlying native architecture.

Our compiler backend does not suffer from these limitations. Our backend framework maintains the code array, resizing them as necessary. This is done transparently to both the client code (our compiler frontend) and to the architecture-dependent backend components. In addition, our generated code is fully relocatable. Although such code might not perform as well as non-relocatable code on some architectures, it provides extra flexibility. Our entire compilation system is fully reentrant. Finally, our backend uses well-defined Java types rather than C types. Types are the same on all architectures.

VCODE currently supports the MIPS, SPARC, and Alpha architectures. All three are RISC architectures. Engler estimates that between 1 to 4 days of work are required to add support for a new RISC architecture. VCODE does not support the ubiquitous x86 CISC architecture. Several challenges need to be solved for that

architecture. In SableJIT, we use virtual registers, as suggested by Engler, to solve the small register set problem of the x86. Virtual registers consist of space allocated on the native stack as register space.

To allow for a quick port, VCODE defines a core instruction set that each architecture is required to implement. Several extension layers are then defined in terms of this core set. These instructions can later be implemented in terms of native instructions for efficiency. In SableJIT, we do not explicitly distinguish between core instructions and extended instructions, although we do provide a default implementation to several instructions.

11.1.3 GNU Lightning

GNU lightning [GNU] is a library for dynamic code generation. It borrows ideas from VCODE and is very similar to VCODE in several respects. It currently supports the x86, SPARC, and PowerPC architectures. In GNU lightning, only 6 registers in addition to the stack pointer (SP) are made available to the client code. Of these 6 registers, 3 registers are guaranteed to be preserved across function calls (callee-saved) while the other 3 are not. By limiting the number of registers to 6, GNU lightning solves the limited register set challenge of the x86 without resorting to implementation tricks such as virtual registers.

GNU lightning, like VCODE, suffers from the problem that a client must allocate sufficient memory to hold the generated code. GNU lightning, unlike VCODE, does have provisions for reentrant and multithreaded code. Several code generations can be in progress at the same time provided that the client code for each code generation in progress resides in different object files. The reason for this is that the static variable containing the compilation state is defined in a GNU lightning header file. It is also possible for the client code to specify its own state variable. This can be used to implement a reentrant compiler, but in this case extra work is put on the client side as it needs to manage state variables.

The main motivation behind the design of GNU lightning is to provide a JIT for the GNU Smalltalk interpreter. It has however been designed to be easily used in

other projects as well.

Since GNU lightning is similar in both design and implementation to VCODE, we will leave out a comparison of GNU lightning with SableJIT.

11.1.4 LIL

LIL [GTC⁺04] is a low-level domain-specific language designed to address specific problems in the Open Runtime Platform (ORP [CEG⁺03]). ORP is a research platform that defines JIT and garbage collector (GC) interfaces. Its goal is to allow JITs and GCs to be independently develop and used as interchangeable modules.

Virtual machine stubs are small functions implementing runtime support for tasks such as object allocation and JNI native method calls. These stubs need access to registers and to the native stack. Therefore, they cannot be written in a high-level language. For this reason, they were originally implemented in assembly language. However, writing stubs in assembly language was tedious, non-portable and difficult to maintain. Since ORP supports both the x86 and ia64 processor architectures and both the Linux and Windows operating systems, portability and maintainability became important issues. LIL was designed to address these problems. LIL is a very low-level architecture-independent language designed to implement stubs efficiently. Stubs written in LIL can be shared across all architectures. LIL consists of both low-level assembly-like instructions and high-level VM-specific instructions. The compilation of LIL code is done at runtime as some stubs are specialized with runtime values. No IR is built and no optimizations are performed. The code is generated directly after computing space requirements.

Whereas VCODE and GNU lightning were designed as general purpose retargetable code generators, LIL was conceived to solve a specific problem: the implementation of stubs in an efficient and portable way.

11.1.5 The Virtual Processor

The Virtual Processor (VPU [Piu04]) by Piumarta is a general-purpose high-level code generator. Complex tasks such as register allocation that were usually left to

the client code on previous retargetable code generators are entirely handled by the VPU. The VPU aims to be an intermediate between a full language specific dynamic compiler and a virtual assembly language code generator. It should be easier to write client code for the VPU than other lower-level code generators. The VPU model is not register-based but stack-based. The complete set of C operations are supported as well as operations for stack manipulation. Its implementation consists of constructing an IR, optimizing the code, performing register allocation, and emitting code. Only a limited number of optimizations are performed. They are chosen for their effectiveness and low-cost. Code generation is done with a BURS. The VPU currently supports three architectures: PowerPC, SPARC and Pentium.

The VPU and SableJIT differ in their main objective. The VPU goal is to minimize the amount of work required for the client code (or frontend) whereas in SableJIT the goal is to minimize the amount of code required to port the backend to a new architecture. As the source code of the VPU does not seem to have been made available, we could not evaluate the amount of work that would be required to add support for a new architecture.

11.2 Portability in Virtual Machines and JITs

11.2.1 OpenJIT

OpenJIT [Ope,MOKSH98] is a reflective just-in-time open compiler framework written in Java. OpenJIT can self-compile as is the case with SableJIT. Some C code is used to act as glue code by providing an interface allowing the compiler to plug into any virtual machine supporting the Java Compatibility Kit (JCK).

It supports the SPARC (Solaris) and x86 (Linux, FreeBSD, and Solaris) architectures. The authors report that porting the SPARC implementation to x86 was done without much difficulty. They also state that it would be relatively easy to port the backend to a new RISC architecture. By looking at the source, the porting strategy seems to be about “translating” several files to the target architecture. We do not see a clear separation between architecture-dependent and architecture-independent

components as it is the case with other JITs including SableJIT. There is no clear limited set of functions or instructions to implement. Retargeting OpenJIT would consist of searching throughout the code and rewriting classes. One of the goals in the design of our compiler is to isolate as much as possible architecture-independent functionality and to share it as much as possible between the architectures in order to minimize the amount of effort required for porting.

Finally, although it is written in Java, the code does not seem to take advantage of the Java language features. In particular, inheritance is more used to separate the code into several files rather than into logical components sharing common functionality through inheritance.

11.2.2 Kaffe

Kaffe [Kaf] is a virtual machine with both an interpreter and a JIT. Kaffe is entirely written in C with some assembly. For short-running applications, this might provide a performance advantage over a JIT written in Java such as SableJIT. In SableJIT, the compiler code is first interpreted, leading to a higher overhead in compilation early in execution time. For long-running applications, however, the overall impact becomes less significant since the compiler code gets compiled to native code as execution progresses.

Kaffe uses C preprocessor macros extensively. We found that this makes the source difficult to follow from time to time. The SableJIT compiler is written to take advantage of features of the Java programming language. In particular, common functionality is shared through inheritance. In addition, some bugs in our compiler are immediately detected such as dereferencing null pointers or indexing arrays out of bounds. Exceptions caused by such bugs are gracefully handled by our compiler. Recovery can be performed for most of them, thus avoiding the termination of the virtual machine. We think that these features makes the design of SableJIT clearer and more robust.

Like SableJIT, Kaffe was designed with portability in mind. In particular, all architecture-dependent code is isolated in a separate directory tree with subdirectories

for each supported processors and operating systems. A VCODE-like approach, called *icode*, is used. *Icode* instructions are implemented with C macros. Unlike SableJIT, the native code is not emitted in a single pass. Kaffe uses two stages: *bytecode analysis* and *translation*. The bytecode analysis stage consists of performing a pass through the bytecode in order to compute information such as bytecode attributes and usage of locals. In the translation phase, each basic block is translated to native code in two passes. First, the JIT constructs an IR data structure containing pointers to code emitting functions. Optimizations and register allocation are performed only within basic blocks. In the second pass, the data structure is traversed and the functions are called to emit code.

SableJIT does have a stage corresponding to bytecode analysis since SableVM already computes a large and sufficient amount of information about bytecodes. Unlike Kaffe, SableJIT emits the native code directly without constructing a data structure. Finally, SableJIT processes bytecodes individually for code generation rather than basic blocks. This very local view of the bytecode provides fewer opportunities for optimizations than Kaffe.

The interpreter of Kaffe runs on several processors including x86, ia64, arm, MIPS, PowerPC, SPARC, and Alpha. Several operating systems are supported, including: Linux, *BSD, other Unixes, Windows/Cygwin, several real time OSes, and several embedded OSes. A JIT is not available though for all architectures supported by the interpreter. The Kaffe JIT currently supports more architecture than SableJIT. The last generation JIT, known as JIT3, runs on the following processor architectures: arm, x86, m68k, and MIPS. On Alpha and SPARC, only the previous generation JIT is available. The wide range of platforms supported by Kaffe confirms that it is a very portable virtual machine and JIT.

11.2.3 Jikes RVM

Jikes RVM (Research Virtual Machine) [AAB⁺00] is a virtual machine consisting only of compilers. The Java bytecode is always compiled, never interpreted. It is first compiled quickly but naively with a baseline compiler. Then, to improve performance,

frequently executed code is recompiled later with an optimizing compiler. Jikes RVM uses several IRs and has several levels of optimizations. One interesting aspect is that the virtual machine is entirely written in Java with the exceptions of a boot loader and various small routines that are written in C.

Jikes RVM was originally designed as a high performance virtual machine for the AIX/PowerPC platform. It was later ported to Linux on both the x86 and the PowerPC architectures. More recently, it has been ported to Mac OS X on PowerPC. The PowerPC and x86 implementation are located in separate directories. Most of the compiler infrastructure is shared between all architectures. The naive compiler of Jikes RVM, called the baseline compiler, is similar to SableJIT in that the compiler mimics the Java stack operations without much optimizations. However, the strategy used to retarget it is different. In Jikes RVM, it is required to implement all bytecode instructions for the target platform whereas in SableJIT, only the implementation of simpler elementary RISC-based instructions is required. In other words, the architecture-specific implementation happens at a lower level in SableJIT. The implementation of individual bytecodes in SableJIT can therefore be shared among all architectures. Doing so might lead to a less efficient baseline compiler as the sequences of native instructions generated for a bytecode instruction in our compiler might not turn out to be as optimal as in Jikes RVM. However, porting SableJIT to a new platform should be simpler as it is not required to have knowledge of the virtual machine internals or the semantics of bytecode instructions. Simply knowing the target architecture should be sufficient.

The optimizing compiler in Jikes RVM converts higher-level IRs to a machine-dependent IR (MIR) that is then used to generate native code. A low-level IR (LIR) to MIR converter needs to be implemented for each architectures. It is also required to implement a MIR to machine code converter. Part of the tedious work involved has been automated with the use of specification files and of a BURS.

11.2.4 Sun's HotSpot

The Java HotSpot VM [Hot,PVC01] is a virtual machine using an adaptive optimizer. Java bytecode is first interpreted and then, frequently executed portions, called *hot spots*, are compiled to native code. Optimizations are based on information collected from the program execution. The system can therefore adapt to the program being executed.

The HotSpot VM has two compilers: the Java HotSpot Client and the Java HotSpot Server. The Client is more suitable for interactive applications requiring a fast startup whereas the Server is more suitable for server applications where throughput rather than startup time matters.

Sun claims that HotSpot is highly portable however we could not verify this claim². A machine description file is used to describe all architecture-specific aspects. The Java HotSpot VM is available for Solaris (SPARC 32-bit/64-bit, x86), Linux (x86) and Windows (x86). HP and Apple also uses the HotSpot technology on their respective platforms: PA-RISC and Mac OS X (PowerPC) respectively.

11.3 Summary

In this chapter, we have presented a survey of various retargetable code generators. We have also studied the retargetability aspects of various existing JVMs and JITs.

²We cannot have access to the source code of HotSpot for the purpose of this thesis. Our description is based on the information made publicly available.

Chapter 12

Conclusion and Future Work

12.1 Summary and Conclusions

JITs have become an essential component of high-performance virtual machines. However, JITs, by their nature, tend to be very architecture-specific. Porting them to new platforms usually involve some large amount of tedious work. Large portions of the compiler might have to be rewritten to the target platform.

In this thesis, we introduced SableJIT, a retargetable JIT. We have explained how SableJIT is a natural extension of SableVM by integrating well with any of the three available interpreters and by taking advantage of existing features in SableVM such as signal-based exception handling.

We have studied the relationship between the inline-threaded interpreter and a naive JIT by developing a baseline compiler that mimics the Java operations on the stack. We could say that our compiler takes inline-threading one level further by removing the remaining dispatch overhead of interpretation. Our results have shown that our naive JIT has difficulty competing with the advanced interpreters of SableVM.

We have designed a retargetable VCODE-like backend for SableJIT. Our backend was designed to work well within a Java JIT context through the use of inheritance and the use of Java semantics in our retargetable virtual instruction set. Virtual register support was implemented for the x86 architecture and our results demonstrate

that virtual registers, with a clever implementation, are a viable option for simple retargetable code generation engines.

We have minimized the required amount of work to port SableJIT to a new platform. We have made a serious attempt to isolate as much as possible architecture-independent code from architecture-dependent code. Several code portions such as the generation of jump tables and signal-based exceptions have been made mostly architecture-independent by isolating the architecture-dependent code in small primitive functions. The retargetable backend has been designed such that no knowledge of virtual machine internals should be required to simply port SableJIT to a new platform, knowledge of the target platform should be sufficient.

We have designed a good development and testing environment. A testing framework accompanied by a series of test cases is provided to assist developers in testing their implementation of new target architectures. Furthermore, the compilation system is robust enough to survive compilation failures. Methods failing to compile are marked as uncompileable and are simply interpreted. Our development environment favours incremental development and testing.

Finally, our retargetable backend framework was applied to several platforms: x86 (Linux, FreeBSD), PowerPC (Linux, Mac OS X), and SPARC (Solaris). In particular, the portability of SableJIT was demonstrated by porting SableJIT to Solaris/SPARC without making any major changes in the architecture-independent components.

12.2 Future Work

Although, SableJIT already supports three processor architectures, it is still work in progress and it can be improved in several ways. Improvements can be grouped in two categories: performance-related and retargetability-related. Performance improvements are important to make SableJIT useful for real-world applications. Retargetability improvements help adding support for new platforms.

We enumerate future improvements:

- SableJIT does not have an optimizing compiler. We would like to design an

optimizing compiler to improve performance.

- SableJIT does not have an advanced profiling system. In particular, we would like to complete our implementation for profiling loop back edges and for switching from interpreted code to compiled code on loop back edges.
- Compiled preparation sequences are not patched with a faster implementation. We would like to implement a patching mechanism that would be as retargetable as possible.
- The RT instruction set consists mostly of simple primitive instructions. We would like to consider adding higher-level instructions such as a field and array access instructions. These would have a default implementation in terms of simpler instructions, however, it would be possible to override them to provide a better implementation on some architectures.
- Retargeting the backend still involves tedious work. We would like to automate part of the process through the use of machine specification files. We hope to reduce errors if instruction encoding information could be provided in a table form.
- SableJIT does not use any code-generator generator. With the addition of an optimizer, we would like to consider the use of a BURS to help translating expression trees to RT code or to native code.
- SableJIT does not currently support any 64-bit architectures. Although some provisions have been made in the design concerning 64-bit architecture, SableJIT is currently not fully 64-bit compliant. It would be interesting to port SableJIT to 64-bit architectures.

Appendix A

SableJIT User Guide

A.1 Getting and Installing SableJIT

SableJIT is known to work on the following platforms:

- x86: GNU/Linux and FreeBSD
- PowerPC: GNU/Linux and Mac OS X
- SPARC: Solaris

SableJIT is available for download on the SableVM website:

<http://sablevm.org/>

For a specific release `x.y.z` of SableVM, a corresponding version with SableJIT support is packaged in `sablevm-sablejit-x.y.z.tar.gz`. This file contains the full source code of SableVM modified for SableJIT support as well as the source code of the SableJIT compiler. For convenience, a file `sablevm-sablejit-patch-x.y.z.tar.gz` is also made available. Rather than including the full sources of SableVM, this file includes a patch to add SableJIT support to SableVM. In both cases, it is also required to download the file `sablevm-classpath-x.y.z.tar.gz` that contains the Java class library for SableVM.

A.2. Customizing SableJIT

The following instructions assume that the file with the full SableVM and SableJIT source is used. To patch the source code, the documentation in the archive should be consulted.

1. Unpack the `sablevm-sablejit-x.y.z.tar.gz` archive:

```
tar -xzvf sablevm-sablejit-all-x.y.z.tar.gz
```

2. Consult the `INSTALL` files located in the `sablevm` and the `sablejit` directories for current build requirements and installation instructions.

3. Unpack and install SableVM Classpath:

```
tar -xzvf sablevm-classpath-x.y.z.tar.gz
cd sablevm-classpath-x.y.z
./configure
make
make install
```

4. Install SableVM and SableJIT. In the `sablevm` directory, do:

```
./configure --with-sablejit
make
make install
make -f Makefile.sablejit sablejit
make -f Makefile.sablejit sablejit_install
```

To test if the installation went well, do:

```
make -f Makefile.sablejit sablejit_version
```

This should print the SableJIT configuration information.

A.2 Customizing SableJIT

A.2.1 Source Organization

The `bin` directory contains various sample scripts to build SableVM and SableJIT with different configurations. The `sablejit` directory tree contains the Java classes

that are part of the SableJIT compiler. The `sablevm` directory tree contains the SableVM sources modified for SableJIT support. In particular, the SableJIT runtime files are located in `src/libstablevm/sablejit`.

A.2.2 Configure Options

Any SableVM configuration options can be passed to the `./configure` script. In addition, several SableJIT-specific options can be passed. A listing of all current options can be obtained by executing the following command:

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

We briefly describe some of them:

- **Files and programs locations:**

```
--with-sablejit-srcdir=DIR           SableJIT Compiler (Java) files
                                     location. Default: ../sablejit
--with-sablejit-junit-path=CLASSPATH Location of the JUnit testing
                                     framework.
--with-sablejit-java-compiler=PATH   Java compiler to use.
```

- **Untested platform builds:** If you are trying out SableJIT on an untested platform, you can explicitly specify the backend to use.

```
--with-sablejit-force-platform=PLATFORM
Current values for PLATFORM are:
x86-linux      - x86 using same calling convention as GNU/Linux.
                 Works on FreeBSD.
ppc-sysv       - ppc using the System V calling convention.
                 Used by GNU/Linux.
ppc-macosx     - ppc using the Mac OS X calling convention.
sparc-solaris - Used by Solaris on SPARC.
```

- **Signal-based exception handling:** If SableVM is configured to use signals for exceptions (option `--enable-signals-for-exceptions`), they will be enabled for compiled code as well. The best known configuration for the target platform is used by default. It is however possible to enable and disable them independently:

```
--disable-sablejit-signals-for-exceptions-null
--disable-sablejit-signals-for-exceptions-div
--enable-sablejit-trapping-null
--disable-sablejit-trapping-div-by-zero
--disable-sablejit-trapping-array-bounds
```

- **Testing Mode (or Control Mode):** It is required to enable the testing mode in order to run test cases. The testing mode also provides some control over the methods that are to be considered for compilation (see section A.3). The testing mode is enabled with:

```
--enable-sablejit-testing
```

A.3 Running SableVM with SableJIT

In non-testing mode, SableJIT considers all methods for compilation. To disable the compiler and run SableVM in interpreter-only mode, pass the `-C int` option to SableVM on the command line.

In testing mode, SableJIT does not compile any method by default. Methods to be considered for compilation must be explicitly specified by using one or more of the following Java properties:

```
sablevm.jit.compile.include=LIST
sablevm.jit.compile.exclude=LIST
sablevm.jit.recompile.include=LIST
sablevm.jit.recompile.exclude=LIST
```

Properties are specified on the command line by using the `-p` option. Each property value is a list of method specifications delimited by “:”. A method specification is a prefix of the *fully qualified name* of a method, that is, the fully qualified class name, followed by the method name and the method descriptor. A special method specification for the `sablevm.jit.compile.include` property is the ALL wildcard that matches all methods. We present a few examples:

- `"java/lang/"` (methods of classes in package `java.lang`)

- "java/lang/String" (methods of classes with prefix `java.lang.String` such as `java.lang.String` and `java.lang.StringBuffer`)
- "java/lang/String." (methods of the `java.lang.String` class)
- "java/lang/String.toString" (`toString()` method of `java.lang.String`)

The set of methods considered for compilation/recompilation is *INCLUDE_SET-EXCLUDE_SET*. If recompilation is enabled and the `sablevm.jit.recompile.include` property is not defined, the default is to consider all methods for recompilation.

A.4 -C Options

Some options are specified as `-C` options. These are used to control the compiler and to provide information on the compiler. The most useful `-C` options are:

- `-C int`: Runs SableVM in interpreter-only mode. The compiler is disabled.
- `-C intr-count=VALUE`: Specifies the number of times a method should be interpreted before being considered for compilation.
- `-C pool-size=VALUE`: Specifies the size of the compiler object pool. This imposes a limit on the number of compilations that can be in progress at any time. In particular, a value of 1 indicates a single compilation at any time; the compiling thread will not be allowed to re-enter the compiler code. Valid values range from 1 up to some constant value set at build time.
- `-C recompile-threshold`: Sets the threshold before a method is considered for recompilation. The threshold is compared against the number of times slower `PREPARE_inst` instructions have been executed.
- `-C version`: Prints the version of SableJIT and its configuration.
- `-C help`: Prints a list of available `-C` options.
- `-C info`: Prints more info on available options including the properties.

Bibliography

- [AAB⁺00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Sheperd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.
- [BD98] Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proceedings of the 1998 International Conference on Computer Languages*, page 240. IEEE Computer Society, 1998.
- [CEG⁺03] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open runtime platform: A flexible high-performance managed runtime environment. *Intel Technology Journal*, 7(1):5–18, February 2003.
- [Cla] Classpath.
URL: <<http://www.classpath.org/>>.
- [DVL98] Geert Deconinck, Johan Vounckx, Rudy Lauwereins, and Jean Peperstraete. Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. *International Journal of Modelling and Simulation*, 18(1):66–71, 1998.

- [EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: a language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 131–144. ACM Press, 1996.
- [Eng] Dawson R. Engler. A VCODE tutorial.
URL: <http://www.pdos.lcs.mit.edu/~engler/vcode-tutorial.ps>.
- [Eng96] Dawson R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN PLDI '96*, pages 160–170, 1996.
- [EP94] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of ASPLOS-VI*, pages 263–272, October 1994.
- [Ert] Anton M. Ertl. A portable Forth engine.
URL: <http://www.complang.tuwien.ac.at/forth/threaded-code.html>.
- [FHP92] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.
- [Gag02] Etienne M. Gagnon. *A Portable Research FrameWork for the Execution of Java Bytecode*. PhD thesis, McGill University, December 2002.
- [GCJ] The GNU Compiler for Java (GCJ).
URL: <http://sources.redhat.com/java/>.
- [GH01] Etienne M. Gagnon and Laurie J. Hendren. SableVM: A research framework for the efficient execution of java bytecode. In *Proceedings of the USENIX JVM '01*, pages 27–40, April 2001.

Bibliography

- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.
- [GNU] GNU lightning.
URL: <<http://www.gnu.org/software/lightning/>>.
- [GTC⁺04] Neal Glew, Spyridon Triantafyllis, Michal Cierniak, Marsha Eng, Brian Lewis, and James Stichnoth. LIL: An architecture-neutral language for virtual-machine stubs. In *Proceedings of the USENIX VM '04*, pages 111–125, May 2004.
- [HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 32–43. ACM Press, 1992.
- [Hot] White paper – the java hotspot virtual machine, v. 1.4.1.
URL: <http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_4.html>.
- [ibu] iburg.
URL: <<http://www.cs.princeton.edu/software/iburg/>>.
- [JBu] JBurg.
URL: <<http://jburg.sourceforge.net/>>.
- [Jik] Jikes.
URL: <<http://oss.software.ibm.com/developerworks/opensource/jikes/>>.
- [Kaf] Kaffe.
URL: <<http://www.kaffe.org/>>.

Bibliography

- [LCC] `lcc`, A Retargetable C Compiler.
URL: <<http://www.cs.princeton.edu/software/lcc/>>.
- [LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
- [M4] The GNU m4 Macro Processor.
URL: <<http://www.gnu.org/software/m4/m4.html>>.
- [MOKSH98] S. Matsuoka, H. Ogawa, Y. Kimura K. Shimura, and K. Hotta. OpenJIT – A reflective Java JIT compiler. In *Proceedings of the OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, pages 16–20, October 1998.
- [Ope] OpenJIT.
URL: <<http://www.openjit.org/>>.
- [PEK97] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. `tcc`: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN PLDI '97*, pages 109–121, 1997.
- [Piu04] Ian Piumarta. The virtual processor: Fast, architecture-neutral dynamic code generation. In *Proceedings of the USENIX VM '04*, pages 97–110, May 2004.
- [PLG88] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: an application of BURS theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 294–308. ACM Press, 1988.
- [PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300. ACM Press, June 1998.

- [PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM-01)*, pages 1–12, Berkley, USA, April 2001. USENIX Association.
- [Saba] SableCC, The Sable Research Group Compiler Generator.
URL: <<http://www.sablevm.org/>>.
- [Sabb] SableVM.
URL: <<http://www.sablevm.org/>>.
- [SPE] SPECjvm98 Benchmarks.
URL: <<http://www.spec.org/osg/jvm98>>.
- [SYN03] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 312–323. ACM Press, 2003.
- [THL02] Mustafa M. Tikir, Jeffrey K. Hollingsworth, and Guei-Yuan Lueh. Re-compilation for debugging support in a jit-compiler. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 10–17. ACM Press, 2002.
- [Wha01] John Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 166–179. ACM Press, 2001.