A PORTABLE RESEARCH FRAMEWORK FOR THE EXECUTION OF JAVA BYTECODE

by

Etienne Gagnon

School of Computer Science
McGill University, Montreal

December 2002

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

Compilation to bytecode paired with interpretation is often used as a technique to easily build prototypes for new programming languages. Some languages, including Java, push this further and use the bytecode layer to isolate programs from the underlying platform. Current state-of-the-art commercial and research Java virtual machines implement advanced just-in-time and adaptive compilation techniques to deliver high-performance execution of Java bytecode. Yet, experimenting with new features such as adding new bytecodes or redesigning the type system can be a daunting task within these complex systems, when new features invalidate assumptions on which the internal dynamic optimizing compiler depends. On the other hand, simpler existing Java bytecode interpreters, written purely in high-level languages, deliver poor performance. The main motivation behind this thesis was to answer the question: *How fast can a portable, easily modifiable Java bytecode interpreter be?* In order to address this question, we have designed and developed the *SableVM* research framework, a portable interpreter-based Java virtual machine written in portable C.

In this thesis we introduce innovative techniques for implementing an efficient, yet portable Java bytecode interpreter. These techniques address three areas: instruction dispatch, memory management, and synchronization. Specifically, we show how to implement an inline-threaded engine in the presence of lazy code preparation, without incurring a high synchronization penalty. We then introduce a logical partitioning of runtime system memory that simplifies memory management, and a related sparse interface virtual table design for fast interface-method invocation. We show how to efficiently compute space-efficient garbage collection maps for *verifiable bytecode.* We also present a bidirectional object layout that simplifies garbage collection. Finally, we introduce an improvement to thin locks, eliminating busy-wait in case of contention.

Our experiments within the *SableVM* framework show that *inline-threading* [PR98] Java delivers significant performance improvement over switch and direct-threading, that sparse interface tables cause no memory loss, and that our map computation algorithm delivers a very small number of distinct garbage collection maps. Our overall performance measurements show that, using our techniques, a portable interpreter can deliver competitive interpretation performance, and even surpass that of a less-portable state-of-the-art *interpreter* on some benchmarks.

Résumé

La compilation en code-octet combinée avec l'interprétation est une technique souvent utilisée pour bâtir des prototypes de nouveaux langages. Certains langages, dont Java, vont plus loin et utilisent la couche de code-octet pour isoler les programmes de la plate-forme sous-jacente. Les machines virtuelles de pointe récentes, pour Java, incluent des compilateurs juste-à-temps avancés et usent de techniques de compilation adaptables pour offrir une haute performance d'exécution du code-octet Java. Toutefois, l'expérimentation de nouvelles caractéristiques, telles l'ajout de nouveaux codes-octets ou la modification du système de types, peut être une tâche colossale lorsque ces nouvelles caractéristiques invalident des hypothèses sur lesquelles le compilateur optimiseur interne dépend. D'autre part, les interpréteurs de code-octet Java plus simples existants, écrits avec des langages de haut niveau, offrent une faible performance. La motivation principale de cette thèse est de répondre à la question : *Jusqu'à quel point un interpréteur de code-octet portable et facilement modifiable peut-il être rapide ?* Pour répondre à cette question, nous avons conçu et développé *SableVM*, une machine virtuelle de Java portable, basée sur un interpréteur et écrite en C portable.

Dans cette thèse nous introduisons de nouvelles techniques pour implémenter un interpréteur de code-octet efficace et portable. Ces techniques couvrent trois sujets : l'envoi des instructions, la gestion de mémoire et la synchronisation. Plus spécifiquement, nous montrons comment implémenter un engin linéaire inclusif en présence d'une préparation paresseuse du code, sans payer un coût élevé de synchronisation. Puis nous introduisons une division logique de la mémoire du système d'exécution qui simplifie la gestion de mémoire. Nous présentons une conception de table virtuelle d'interface clairsemée permettant une invocation rapide des méthodes.

Nous montrons comment calculer efficacement des cartes de ramassage de miettes peu spacieuses pour du *code-octet vérifiable*. Nous présentons également une disposition bidirectionnelle des objets qui simplifie le ramassage des miettes. Finalement, nous introduisons une amélioration aux verrous légers qui élimine l'attente active en cas de litige.

Nos expérimentations au sein du cadre *SableVM* montrent qu'une interprétation *linéaire inclusive* [PR98] de Java offre une amélioration significative de la performance par rapport à une interprétation linéaire aiguillée ou directe ; que les tables d'interface clairsemées ne causent pas de pertes de mémoire et que notre calcul de cartes de ramassages de miettes livre un très petit nombre de cartes distinctes. Nos mesures globales de performance démontrent qu'un interpréteur portable utilisant nos techniques peut fournir une performance compétitive, surpassant celle d'un interpréteur de pointe moins portable pour certains programmes témoins.

Acknowledgments

This thesis would not be without the support of many other people. First, and most importantly, I would like to thank my supervisor, Professor Laurie Hendren, for her unfailing enthusiasm and her indispensable guidance throughout the course of this doctoral research. Thanks, Laurie, for infecting me with your passion for compilers. Thanks, also for all your help and support, and for having an open-door policy for your graduate students!

I would like to thank all the faculty and student members of the McGill Sable Research Group for their invaluable help discussing and commenting this work. I would like to thank, in particular, Professor Karel Driesen for providing pointers to important related work, and for his help in better defining the scope of this project. I would also like to thank Marc Berndl for his work on investigating inline-threading problems with *gcc*. I would like to thank Bruno Dufour for building the scripts for collecting empirical results and transforming the output into various formats, including HTML. I would also like to thank John Jorgensen for his invaluable help with the installation of libraries and other system administration tasks.

I would like to thank the various contributors and users of *SableVM* for their help and feedback. I would like to specially thank Grzegorz Prokopski for porting *SableVM* to the Debian/Alpha platform, and for his continuing help porting *SableVM* other Debian supported platforms. I would also like to thank Archie Cobbs for porting *SableVM* to the FreeBSD platform, and contributing bug fixes. I would like to thank Chak Wai So for helping to discover and fix little discrepancies in the garbage collector, and Brent Fulgham for contributing code for parsing configuration files.

This research wouldn't have been possible without financial support. I would like

to thank my supervisor Laurie Hendren, again, for her financial support. I would also like to thank FCAR for their graduate studies fellowship. I would like to thank Hydro-Québec and McGill for their reserved Ph.D. fellowship. I would like to thank UQÀM for providing me with a dream faculty position even before the end of my studies.

Finally, sometimes words are not enough. How could I thank enough my wife Gladys Iodio for all her love, moral support, help, patience, understanding, and so much more she provided me for all these years? Thanks, thanks, thanks, and more thanks!

Contents

Appendices

List of Figures

List of Tables

Chapter 1
Introduction and Contributions

1.1 Introduction

1.1.1 The Java Virtual Machine

Over the last few years, Java [GJSB00] has rapidly become one of the most popular general purpose object-oriented programming languages. The Java language was designed, from the ground up, to provide platform independence and security. This is achieved by compiling Java programs into *class files* which include type information and platform independent bytecode instructions. On a specific platform, a runtime system (or *virtual machine* [LY99]) loads and links class files, then executes bytecode instructions.

The idea of compiling a language to bytecode instructions and interpreting the result is not new; it is in fact a relatively common practice used in undergraduate compiler courses to limit the scope of term projects, and it is used in language research projects to rapidly prototype systems. Bytecode is also often used as a means to isolate compiled programs from the underlying platforms, as illustrated by the P-CODE system for PASCAL [Wir71], and in Caml [Cam] implementations. The Java programming language pushed this a little further by specifying the bytecode language as a non-optional core component of the system.

The virtual machine collaborates with a rich standard class library to provide key services to Java programs, including threads and synchronization, automatic memory management (garbage collection), safety features (array bound checks, null pointer detection, code verification), reflection, dynamic class loading, and more.

We should note that there exist static compilers that directly compile Java programs to machine code (e.g. [GCJ, Har, Tob]). Yet, the constraints of static and dynamic Java environments are quite different. Our research focuses solely on dynamic Java execution environments.

1.1.2 The Quest for High Performance

Early Java virtual machines were simple bytecode interpreters. Soon, the quest for performance led to the addition of *Just-In-Time compilers* (JIT) to virtual machines, an idea formerly developed for other object-oriented runtime systems like Smalltalk-80 [DS84] and Self-91 [CUL89]. JITs range from the very naive, that use templates to replace each bytecode with a fixed sequence of native code instructions (early versions of Kaffe [Kaf] did this), to the very sophisticated that perform register allocation, instruction scheduling and other scalar optimizations (e.g. [ATCL+98,Kra98,SOT+00, YMP+99]).

JITs face two major problems. First, they strive to generate good code in very little time, as compilation time is lost to the running application. Second, the code of compiled methods resides in memory; this increases the pressure on the memory manager and garbage collector. Recent virtual machines mostly overcome these problems. The main trend is to use dynamic strategies to find *hot* execution paths, and only optimize these areas (e.g. [AAB+00, CLS00, Hot]). HotSpot [Hot, PVC01], for example, is a mixed interpreter and compiler environment. It achieves high performance by dynamically profiling interpreted code to identify *hot spots*, then compiling and optimizing them. Jikes RVM [AAB+00, AAC+99], on the other hand, always compiles methods (naively at first), then uses adaptive online feedback to recompile and optimize hot methods. These techniques are particularly suited to virtual machines executing long running programs in server environments. The optimizer can be relatively slow and consist of a full-fledged optimizing compiler using intermediate representations and performing *costly* aggressive optimizations, as compile time will be amortized on the long overall execution time.

1.1.3 Portability

While *most* Java programs enjoy platform independence, the underlying virtual machine that provides this independence is itself a program that must interact with low-level system-specific routines of the host platform. As we have seen in Section 1.1.2, current state-of-the-art virtual machines include sophisticated full-fledged optimizing compilers. It is important to design such high-performance systems in a relatively portable way, as it would be impractical and too costly to completely rewrite such optimizing compilers for each platform to which the system is ported.

It is thus interesting to note that both the HotSpot and the Jikes RVM virtual machines implement their optimizing compilers in the Java programming language. In fact, the Jikes RVM virtual machine goes a step further and is completely implemented in Java, using a relatively complex mechanism to write a precomputed bootstrapping image to disk.

But, this does not mean that the HotSpot and the Jikes RVM internal optimizing compilers are then automatically platform independent: Even though these compilers can theoretically run on any system that provides a Java virtual machine, the generated optimized code targets a specific platform. So, these compilers are useless on a platform unless a compiler back-end is developed for that specific platform. In the case of Jikes RVM, in particular, porting to a new platform requires the porter to learn about the executable file format of the target platform for generating the bootstrap image.

So, in summary, even though the most complex parts of modern virtual machines are usually written in Java, porting to a new platform requires a significant development effort.

1.1.4 Java Virtual Machine Overview

Before getting into the details of our research, we present a short overview of the internal organization of a Java virtual machine. As illustrated in Figure 1.1, the main components of a Java virtual machine are:

1. Class Loaders: Class loaders are used to dynamically load application and library classes from a variety of sources such as the local file system and the network.

2. Native Interface[1]: The native interface allows the virtual machine to call non-Java routines in applications and class libraries.

3. Execution Engine: The execution engine is the heart of the virtual machine. It executes bytecode instructions loaded through class loaders. There exist various types of execution engines such as interpreters and just-in-time compilers.

4. Memory Manager: The memory manager provides a garbage collected heap for object instances and manages the memory used to store other internal virtual machine data structures.

5. Services: This component consists of a collection of sub-components providing the necessary internal virtual machine support for *standard* class library features such as threads and reflection.

1.2 Research Motivation and Objectives

1.2.1 Research Framework

Many academic research projects have limited resources. Sometimes, the human resources dedicated to a project are limited to a single graduate student, or a very small team of researchers. The development effort required to experiment with some

[1] The *Java Native Interface (JNI)* is a standard for dynamically linking Java and non-Java code.

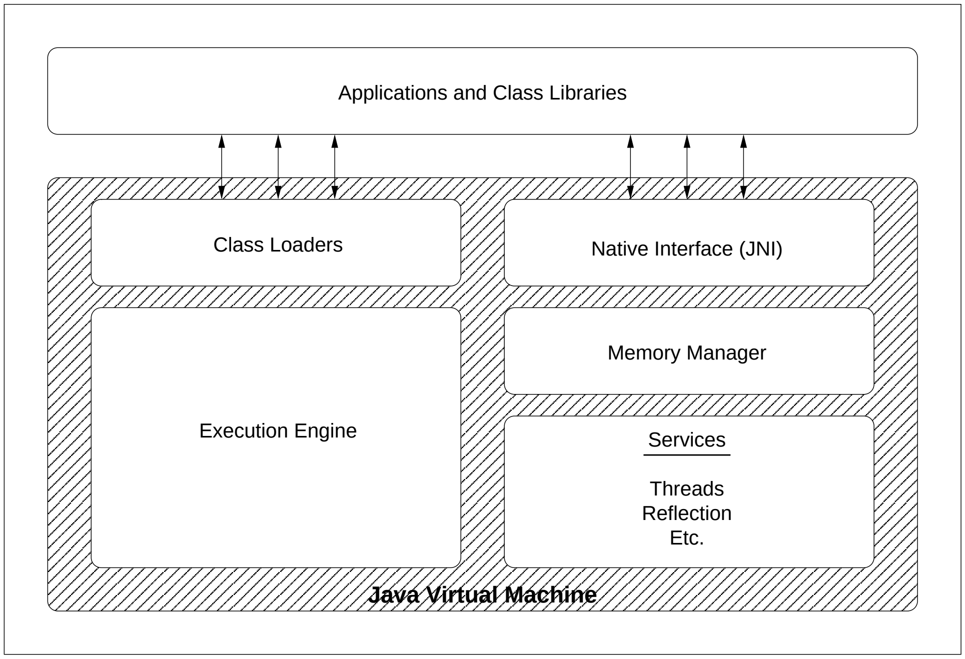


Figure 1.1: Java Virtual Machine Overview

language extensions, in state-of-the-art adaptive Java virtual machine systems, might involve rewriting key parts of highly sophisticated code optimizers. Such development effort can easily be out of reach of a small research team. Even using a simpler template-based just-in-time compiler might require more development work than a research team would like to invest, due to the requirement of writing assembly language for each target platform.

One of the main objectives of this research is the development of an openly available virtual machine suitable for performing research experiments with minimal development efforts. In order to achieve this goal, this virtual machine must be easily extensible, allowing experiments with language modifications and extensions such as redefining the semantics of arrays, or adding new bytecode instructions for better supporting functional languages. This *research framework* must also be easily portable to new platforms with minimal effort, to allow performing experiments on a variety of systems.

Finally, the virtual machine must also deliver acceptable performance, so that experiments can be done running real-world applications, not only toy benchmarks.

1.2.2 Bytecode Interpreter

An interpreter-based virtual machine would seem to meet our portability and easy modification goals. Interpreters written in high-level programming languages have the following advantages:

- Understanding their internal structure requires a very short learning curve.
- They have easily modifiable source code.
- It is usually possible to trace their internal execution with a debugger. This helps with learning the system and modifying it.
- High-level languages help increase the portability to other systems, by hiding low-level details such as the processor instruction set.

The main drawback of interpreter-based systems is that they often deliver poor performance, due to high instruction dispatch overhead.

Recently, a new technique has been introduced, called *inlined-threading* [PR98], that partly eliminates dispatch overhead for a subset of instructions. This technique has not been tested for Java before. As it looked promising in helping to achieve our *acceptable performance* goal, we decided to further investigate the use of this technique in the context of an interpreter-based Java virtual machine.

1.2.3 Specific Research Objectives

The specific objectives of this research are to:

- design and implement a portable and easily modifiable interpreter-based virtual machine,
- evaluate the relative performance achievable by a portable interpreter implementing modern and innovative techniques,

- research new memory management techniques,
- research new techniques for improving the performance of Java virtual machines, regardless of their engine type (interpreter, just-in-time compiler, adaptive optimizing system), and
- measure the performance of the proposed techniques.

A less formal objective is to keep the framework as simple as possible. As the development of a standards compliant Java virtual machine implementing the *Java Native Interface* (JNI) and the *Invocation Interface* requires a significant amount of work, it is important to keep a simple virtual machine design. For example, we have chosen to implement a simple semi-space copying collector, leaving the development of more advanced generational techniques to future interested users of the framework.

1.3 Contributions

In this section, we list the contributions of this thesis.

One contribution of this research is of a technical nature. It consists of the research framework itself. In the course of this research, we have developed the *SableVM* [Sabb] research framework, a freely available, portable, flexible, and efficient interpreter-based Java virtual machine. We think, that the relatively small source-code size of *SableVM* (approximately 55,000 lines before macro expansions) and the clarity and simplicity of its internal design makes it an ideal tool for conducting small to moderately sized research projects on the Java virtual machine.

SableVM implements 2 kinds of object layout, 3 flavors of threaded interpretation, provides a choice of using or not using signals to detect some exceptions, has many embedded debugging features, and can be easily stepped through, at execution time, using a traditional debugger.

Thus, the contributions of this thesis are:

- The development and public release of the *SableVM* research framework.

- The introduction of innovative techniques to allow inline-threaded interpretation of Java bytecode, without race conditions or high synchronization costs in a multi-threaded environment. Our experiments show that inline-threading Java bytecode offers significant speed improvement over that of traditional bytecode interpretation.

- The introduction of a logical partitioning of runtime memory that simplifies memory management. This memory partitioning allows *SableVM* to use a very simple semi-space copying collector to manage the Java heap, and to use very simple partition-specific memory managers for the rest.

- The introduction of a sparse interface method virtual table design that reduces the cost of interface method invocation to that of a normal virtual method call. Appropriate for a dynamic loading environment, this design uses a simple, yet very effective strategy to recycle memory holes in the sparse tables. Our experimental results show that, on all tested benchmarks and applications, including an interface-intensive application, no memory loss resulted from the sparse design.

- The introduction of a simple and fast algorithm to compute space-efficient garbage collection maps for verifiable bytecode. Our experimental results show that at most 74 distinct garbage collection maps of a total size of 1,776 bytes (less than 2Kb) were computed on tested benchmarks, with most benchmarks requiring between approximately 30 to 40 maps each. The biggest application, requiring 74 maps, had 39,653 garbage collection checkpoints.

- The introduction of a bidirectional object layout that groups together all reference fields for simpler garbage collection tracing.

- The introduction of an improvement to thin locks [BKMS98] that eliminates busy-wait in case of contention, without causing any overhead into the object layout.

- One of our most significant experimental results, that counts as a contribution, is that a carefully designed, yet simple and portable Java bytecode interpreter can achieve competitive performance with a commercial state-of-the-art less-portable interpreter. More specifically, the inlined-threaded interpreter of *SableVM* does deliver competitive performance to the official Java Development Kit 1.4.0 HotSpot client interpreter, being sometimes faster, sometimes slower.

1.4 Thesis Organization

The remainder of this thesis is structured as follows.

In Chapter 2, we describe the problem of inline-threading Java in the presence of lazy preparation and multi-threading, and we introduce our *preparation sequence* technique to circumvent the problem and increase the length of inlined instruction implementation sequences. In Chapter 3, we motivate and describe a logical partitioning of runtime memory among various partition-specific memory managers. We explain how this partitioning simplifies memory management in *SableVM*. In Chapter 4, we introduce a sparse interface virtual table design for fast interface-method invocation, taking advantage of a partition-specific memory manager to recycle memory holes using a simple and fast algorithm. In Chapter 5, we discuss the traditional layout of objects in Java and introduce a bidirectional layout for simplifying garbage collection tracing. In Chapter 6 we describe the difficulty of computing precise garbage collection maps in Java, then we introduce a simple and fast, yet effective algorithm for computing space-efficient garbage collection maps. In Chapter 7, we introduce an improvement to thin locks that eliminates busy-wait in case of contention. In Chapter 8, we discuss how portability and extensibility are achieved in *SableVM*. In Chapter 9 we describe our experimentation setting, and present our overall performance measurements, with comparisons to various other virtual machines. Finally, in Chapter 10, we discuss possible future work and present our conclusions.

Chapter 2
Fast Instruction Dispatch

In this chapter we discuss the core instruction dispatch mechanism of *SableVM*. In fact, the *SableVM* framework offers a choice of 3 different flavors of *threaded interpretation* with distinct performance-portability tradeoffs[1], but we will mainly focus on the fastest flavor: *inline-threading*.

In particular, we will introduce the necessary techniques to implement an efficient *inline-threaded* interpreter engine, in the presence of lazy code preparation and multi-threading.

This chapter is structured as follows. In Section 2.1 we discuss how *SableVM* differs from pure bytecode interpreters by preparing and aligning bytecodes prior to execution. Then, in Section 2.2 we describe three existing instruction dispatch techniques and discuss their efficiency-portability tradeoffs. Next, in Section 2.3 we discuss the difficulty of applying threaded interpretation techniques in a Java interpreter without paying a high synchronization penalty, and introduce techniques to solve the problem and increase performance. In Section 2.4, we present our experimental results. In Section 2.5 we discuss related work. Finally, in Section 2.6, we present our conclusions.

[1] In Chapter 8, we will explain how *SableVM* avoids source code duplication while permitting easy debugging of instructions, by implementing abstraction levels using the M4 macro processor.

2.1 Preparation: Reducing Work at Runtime

2.1.1 Pure Bytecode Interpretation

Simplicity is usually the main motive behind the usage of a bytecode instruction set by students and programming language researchers. Targeting stack-based bytecode instructions greatly simplifies compilation by eliminating the need for performing register allocation and isolating the compiler developer from low-level system-specific implementation details such as object-code format. In addition, writing a pure bytecode interpreter can often be done in a few hours of work (given a simple instruction set and a knowledgeable programmer).

A typical bytecode interpreter loads a bytecode program from disk using standard file operations, and stores instructions into an array. It then dispatches instructions using a simple loop-embedded *switch* statement, as shown in Figure 2.1.

```
char code[CODESIZE];
char *pc = code;
int stack[STACKSIZE];
int *sp = stack;


/* load bytecodes from file and store them in code[] */

...

/* dispatch instructions */

while(true) {
  switch(*pc++) {
  case ICONST_1: *sp++ = 1; break;
  case ICONST_2: *sp++ = 2; break;
  case IADD: --sp; sp[-1] += *sp; break;
  ...
  case END: exit(0);
}}
```

Figure 2.1: Pure Switch-Based Bytecode Interpreter

2.1.2 Precomputing and Aligning Data

The Java class file format is relatively complex. It includes, among other things, a *constant pool* used to store the most complex *operands* of bytecode instructions such as strings and class names. Bytecode instructions refer to class pool entries using a one or two byte immediate operand representing an index into the constant pool. Yet, Java bytecode instructions have no alignment requirement[2].

Modern processors usually have *word* sized registers, and most of memory hierarchy hardware is optimized for accessing aligned words (e.g.: single or multiple word cache entries, word aligned access to lower memory). Accessing byte-sized data often results in additional computation or hardware overhead for extracting the appropriate bits from the enclosing word.

In order to simplify computation, and to reduce run-time overhead, *SableVM* does not directly interpret bytecode instructions. Instead, it precomputes an aligned code array with word elements. Said differently, *SableVM* translates bytecodes into *wordcodes*. In the process, *SableVM* performs several optimizations such as translating big-endian multi-byte values into platform-specific words, and eliminating some *constant-pool* indirections by inlining values into the code array.

Translating bytecodes also involves making many small adjustments such as recomputing relative branch targets. *SableVM* takes this opportunity to precompute a variety of values such as absolute branch targets, to minimize run-time computation. *SableVM* also makes the necessary adjustments to exception and line number tables.

Finally, in order to preserve portability to 32 and 64-bit big and little-endian systems, *SableVM* does not assume any particular word size or byte ordering. It simply uses generic types such as `(void *)`, and the `_svmt_word` type (which is defined in a system-specific header file).

[2] Exception: The operands of the *lookupswitch* and the *tableswitch* instructions include padding to provide 32-bit aligned jump tables.

2.2 Dispatch Types

In this section, we describe three dispatch mechanisms generally used for implementing interpreters and discuss their efficiency-portability tradeoffs. The third mechanism is relatively new, and has been introduced by Piumarta and Riccardi [PR98].

2.2.1 Switching

As we have seen in Section 2.1, simple bytecode interpreters use a loop-embedded switch statement to dispatch instructions.

This approach has some benefits:

- It is very simple to implement.
- It is a very portable approach, as it requires no platform or compiler-specific support.
- It requires no special preparation of the bytecode array.

But this approach has performance drawbacks. Dispatching instructions is very expensive. On every iteration, the dispatch loop fetches the next bytecode, looks up the associated implementation address in a table, then transfers control to that address. A typical compilation of the dispatch loop requires a minimum of 3 control transfer machine instructions per iteration: one to jump from the previous bytecode implementation to the head of the loop, one to test whether the bytecode is within the bounds of handled switch-case values, and one to transfer control to the selected case statement. On modern processors, control transfer is one of the main obstacles to performance [HP96], so this dispatch mechanism causes significant overhead.

The main drawbacks of this switch-based bytecode dispatch can be summarized as:

- High dispatch overhead.
- Bytecodes are not aligned, causing additional computation or hardware overhead.

2.2.2 Direct-Threading

An effective technique to reduce dispatch overhead was popularized by the Forth programming language. This technique has the name of *threaded code.* Note that the word *thread*, in this context, has nothing to do with the concurrent programming technique (e.g. *Java threads*, *POSIX threads*). Among the traditional *threaded code* techniques, the most efficient is *direct-threading* [Ert][3].

Direct-threading improves on switch-based dispatch by eliminating central dispatch. This works as follows. In the executable code stream, each bytecode is replaced by the address of its associated implementation. Also, at the end of each bytecode implementation, the code required to dispatch the next opcode is added. This is illustrated in Figure 2.2.

```
/* code */

void *code[] = {
  &&ICONST_2, &&ICONST_2,
  &&ICONST_1, &&IADD, ...
}
void **pc = code;

/* dispatch first instruction */

goto **(pc++);

/* implementations */

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

Execution proceeds as follows. The `code` array is initialized. Then the `pc` program counter variable is initialized, pointing to the first element of `code`. Then, dispatch proceeds by jumping to the address stored in `*pc` by executing the `goto **(pc++);` instruction. The target instruction `ICONST_2` is executed, then the next instruction is

[3] *Inline-threading*, which we discuss in Section 2.2.3, is a recent technique, and thus does not count as traditional.

again dispatched by a single indirect jump `goto **(pc++);`, and so on. This effectively eliminates the table lookup and the central dispatch loop. A typical compilation of this code yields a single control transfer instruction per dispatch.

Direct-threading requires preparation of the code array, as the storage size of an implementation address (`sizeof(void *)`) is larger than that of the bytecode it replaces.

Figure 2.2 uses the *label-as-value* GNU C extension, but direct-threading can also be implemented using a couple of macros containing inline assembly.

Advantages of direct-threading:

- It is relatively simple to implement.
- It is directly supported by the widely ported the GNU C Compiler [GCC], yet also implementable using other compilers.
- It operates on aligned data.

The drawbacks of this approach are:

- Porting to a new platform or compiler might require a little system-specific assembly programming, if the target compiler is not the GNU C compiler.
- It requires preparation of the code array, prior to executing the code, to convert bytecodes into wordcodes.

2.2.3 Inline-Threading

The last dispatch mechanism we survey is that of *inline-threading* [PR98]. This technique improves on direct-threading by eliminating dispatch overhead for instructions within a *basic block* [ASU86].

The general idea is to identify instruction sequences forming basic blocks, within the code array, then to dynamically create a new implementation for the whole sequence by sequentially copying the body of each implementation into a new buffer, then copying the dispatch code at the end. Finally, a pointer to this sequence implementation is stored into the code array, replacing the original bytecode of the first instruction in the sequence.

Figure 2.3 displays a simplified example of creation of an instruction sequence implementation. Figure 2.4 shows the resulting instruction sequence implementation. Note that Figure 2.4 is only an abstract *source code* representation of the actual inlined instruction sequence implementation.

```
/* Instructions */

ICONST_1_START: *sp++ = 1;
ICONST_1_END: goto **(pc++);

INEG_START: sp[-1] = -sp[-1];
INEG_END: goto **(pc++);

DISPATCH_START: goto **(pc++);
DISPATCH_END: ;
```

```
/* Implement the sequence ICONST_1 INEG */

size_t iconst_size = (&&ICONST_1_END - &&ICONST_1_START);
size_t ineg_size = (&&INEG_END - &&INEG_START);
size_t dispatch_size = (&&DISPATCH_END - &&DISPATCH_START);

void *buf = malloc(iconst_size + ineg_size + dispatch_size);
void *current = buf;

memcpy(current, &&ICONST_START, iconst_size);
current += iconst_size;
memcpy(current, &&INEG_START, ineg_size);
current += ineg_size;
memcpy(current, &&DISPATCH_START, dispatch_size);
...
/* Now, it is possible to execute the sequence using: */
goto **buf;
```

Figure 2.3: Inlining a Sequence

Inline-threading improves performance by reducing the overhead due to dispatch. This is specially effective for sequences of simple instructions such as `ICONST_1` and `IADD`, which have a high *dispatch-to-real-work* ratio.

In [PR98], Piumarta and Riccardi experimented with inline-threading on toy

```
ICONST_1 body:  *sp++ = 1;
INEG body    :  sp[-1] = -sp[-1];
DISPATCH body:  goto **(pc++);
```

Figure 2.4: Inlined Instruction Sequence

benchmarks using a simple bytecode language, and achieved, in one case, 70% of the speed of an equivalent optimized C program. Experiments within an *Objective Caml* bytecode interpreter showed significant speed improvement in some conditions (depending on benchmarks and platforms tested).

Processor Specific Concerns

Many modern processors have distinct data and instruction caches. On such systems, an inline assembly function is required in order to ensure that the instruction cache sees the dynamically created sequence implementations. This instruction is simply an architecture-specific *cache-flush* machine instruction, which cannot be expressed in portable C.

The problem is that newly created sequence implementations are written back by the processor to its *data cache*. This data needs to be written back to *main memory* before it can be seen by the *instruction cache*. So, by using this function, we prevent the disastrous execution of potentially random memory content.

Limitations

Unlike direct-threading, which applies uniformly to all instructions, inline-threading presents some limitations. Not all instructions can be inlined. These limitations are mainly caused by relative jumps. As inlined implementations are copied elsewhere in memory, the target of a relative jump within an implementation might become invalid.

The following list of instructions cannot be inlined:

- Instructions that contain C function calls, if the C compiler implements the call as a relative displacement to the processor's program counter (PC).

- Any instruction that causes the C compiler to emit a hidden internal function call, if this call is implemented as a relative displacement to the PC (e.g. this happens for long division, on the x86 platform using *gcc*).

- Any instruction that contains a jump to other than an absolute target address, or a PC-relative one within the `START` and `END` labels of the instruction implementation.

The most obscure is the third class of instructions. This happens, for example, on the x86 platform using *gcc* version 3.1 with optimization on (`gcc -O2`) for any instruction that contains conditionals such as: `if (condition)`. It is an inconsistent behavior that only shows when using specific compiler options. For example, this limitation does not show for conditionals when using `gcc -O0` on the same platform[4].

We discovered this third limitation in our experiments. The initial paper by Piumarta and Riccardi [PR98] did not identify this limitation[5] nor the hidden calls limitation.

In summary, *inlinability* of an instruction implementation is dependent on the compiler, platform, and compiler-options used. Thus inline-threading requires a careful testing of each instruction, to discover whether it is inlinable or not, and under which conditions.

Advantages and Drawbacks

Advantages of inline-threading:

- It completely eliminates dispatch overhead for all but the last instruction of inlined sequences, and can yield significant performance improvement over direct-threading.

- It operates on aligned data.

[4] Further investigation revealed that this problem is caused by reordering of basic blocks by the *gcc* 3.1 optimizer.

[5] Our tests indicate that this limitation does not show in *gcc* 2.95, which they used for their experiments.

Drawbacks:

- Preparation of the code array requires more work, including basic block identification.
- It can require one *memory cache* related inline assembly function on some platforms in addition to having all the requirements of direct-threading.
- Inlined instruction sequence implementations cannot be traced normally using a debugger (other than at the machine-code level).
- Porting to a new system or compiler requires careful testing of instructions to assess their inlinability.

2.3 Inline-Threading Java

In the previous section, we described the idea of inline-threading. To our knowledge, this technique has not been applied to Java interpreters before. In this section, we first explain the difficulty of applying inline-threading to Java bytecode, then introduce new techniques that make it possible.

Even though some of the problems and new techniques discussed in this section also apply to switch and direct-threaded interpreters, we will only focus on inline-threading to simplify the text.

2.3.1 Conflict: Laziness and Multi-Threading

Lazy Loading and Preparation

In Java, classes are dynamically loaded. The Java Virtual Machine Specification [LY99] allows a virtual machine to *eagerly* or *lazily* load classes (or anything in between). But this flexibility does not extend to *class initialization*[6]. Class initialization must occur at specific execution points, such as the first invocation of a static method

[6] Class initialization consists of initializing static fields and executing static class initializers.

or the first access to a static field of a class. Lazily loading classes has many advantages: it saves memory, reduces network traffic, and reduces startup overhead.

As we have seen, inline-threading requires analyzing a bytecode array to determine *basic blocks*, allocating and preparing implementation sequences, and lastly preparing a code array. As this preparation is time and space consuming, it is advisable to only prepare methods that will actually be executed. This can be achieved through lazy method preparation.

Performance Issue

Lazy preparation (and loading), which aims at improving performance, can pose a performance problem within a multi-threaded[7] environment. The problem is that, in order to prevent corruption of the internal data structure of the virtual machine, concurrent preparation of the same method (or class) on distinct Java threads should not be allowed.

The natural approach, for preventing concurrent preparation, is to use synchronization primitives such as *pthread mutexes*[8]. But, this approach can have a very high performance penalty; in a naive implementation, it adds synchronization overhead to every method call throughout a program's execution, which is clearly unacceptable, specially for multi-threaded Java applications.

One-Word Replacement

A clever trick to avoid synchronization on every method call is to put a pointer to a special preparation method in place of a pointer to the real method to be executed, in code arrays and virtual tables. The special preparation method uses synchronization primitives and performs preparation, if it hasn't been done yet, then finally stores a pointer to the real method into the calling code array or virtual table[9].

[7] Note that *multi-threading* is a concurrent programming technique which is inherently supported in Java, whereas *inline-threading* is an instruction dispatch technique.

[8] POSIX Threads mutual exclusive locks.

[9] As we will explain later, this replacement trick only works if there is a single word to change; if two or more words are changed, a race condition occurs in absence of explicit synchronization.

Broken Sequences

In the case of inline-threading the laziness problem is amplified. An important performance factor of inline-threading is the length of inlined instruction sequences. Longer sequences reduce the dispatch-to-real-work ratio and lead to improved performance. Lazy class initialization mandates that the first call to a static method (or access to a static field) must cause initialization of a class. This implies (in a naive Java virtual machine implementation) that instructions such as `GETSTATIC` must use a conditional to test whether the target class must be initialized prior to performing the static field access. If initialization is required, a call to the initialization function must be made. The conditional and the C function call are, in light of the limitations identified in Section 2.2.3, potential reasons that can prevent inlining of the `GETSTATIC` instruction.

What we would like, is to use the same replacement trick as discussed earlier, using two versions of the `GETSTATIC` instruction, as shown in Figure 2.5. But, unfortunately this does not completely solve our performance problem[10].

Even though this technique eliminates synchronization overhead from most function calls, it inhibits the removal of dispatch code in an instruction which has very little *real work* to do. In fact, the cost can be as high as the execution of two additional dispatches. To measure this, we compare the cost two instruction inline-threaded instruction sequences that only differ in their respective use of `ILOAD` and `GETSTATIC` in the middle of the sequence.

Broken Sequence Cost

So, if we had the sequence of instructions `ICONST2-ILOAD-IADD`, we could build a single inlined sequence for these three instructions, adding a single dispatch at the end of this sequence. Cost: $3 \times realwork + 1 \times dispatch$.

If, instead, we had the sequence of instructions `ICONST2-GETSTATIC-IADD`, we would not be allowed to create a single inlined sequence for the three instructions.

[10] Note that, for simplicity, Figure 2.5 implements the *integer static field* access instruction `GETSTATIC_INT` variant of `GETSTATIC`.

Synchronized GETSTATIC	Unsynchronized GETSTATIC

/* Pseudo-code */
 	
GETSTATIC_INIT:
 	
pthread_mutex_lock(...);
 	
/* lazily load class */
...
 	
/* conditional */
if (must_initialize)
{
  /* function call */
  initialize_class(...);
}
 	
/* do the real work */
*sp++ = class.static_field;
 	
/* replace by fast version */
code[pc -1] =
  &&GETSTATIC_NO_INIT;
 	
pthread_mutex_unlock(...);
 	
/* dispatch */
goto **(pc++);
	/* pseudo-code */
	
	GETSTATIC_NO_INIT:
	
	/* do the real work */
	*sp++ = class.static_field;
	
	/* dispatch */
	goto **(pc++);

Figure 2.5: GETSTATIC With and Without Initialization

This is because, in the prepared code array, we would need to put 3 distinct instructions: ICONST2, GETSTATIC_INIT, and IADD, where the middle instruction cannot be inlined. Even though the GETSTATIC_INIT will eventually be replaced by the more efficient GETSTATIC_NO_INIT, the performance cost will remain: $3 \times realwork + 3 \times dispatch$.

So, the overhead of a broken sequence can get as high as two additional dispatches.

Two-Values Replacement

In reality, the problem is even a little deeper. The *pseudo-code* of Figure 2.5 hides the fact that GETSTATIC_INIT needs to replace two values, in the code array: the instruction opcode and its operand. The idea is that we want the address of the static variable as an operand (not an indirect pointer) to achieve maximum efficiency, as shown in Figure 2.6. But this pointer is unavailable at the time of preparation of the code array, as lazy class loading only takes place later, within the implementation of the GETSTATIC_INIT instruction.

Fast Instruction	Code Array

GETSTATIC_NO_INIT:
{
  int *pvalue = (pc++)->pvalue;
  *sp++ = *pvalue;
}
 	
/* dispatch */
goto **(pc++);
	/* Initially */
	...
	[GETSTATIC_INIT]
	[POINTER_TO_FIELD_INFO]
	...
	
	/* After first execution */
	...
	[GETSTATIC_NO_INIT]
	[POINTER_TO_FIELD]
	...

Figure 2.6: Two-Values Replacement in Code Array

Replacing two values without synchronization creates a race condition. Here is a short illustration of the problem. A first Java thread reads both initial values, does the instruction work, then replaces the first of the two values. At this exact point of time (before the second value is replaced), a second Java thread reads the two values (instruction and operand) from memory. The second Java thread will thus get the fast instruction opcode and the old field info pointer. This can of course lead to random execution problems.

2.3.2 Getting Longer Inlined Sequences

Before attacking the problem of two-values replacement, we introduce some techniques to eliminate non-inlinable features from instruction implementations. In other words, using these techniques, we can eliminate conditionals and function calls from the body of many instructions. This will increasing the number of inlinable instructions, leading to the computation of longer inlined sequences, in inline-threaded code.

Type-Specific Instructions

The first technique is to split some bytecode instructions such as `GETSTATIC` into multiple type-specific versions. In Java bytecode, there is a single `GETSTATIC` instruction to access static fields, yet there are eight primitive field types (boolean, byte, short, char, int, long, float, and double), and reference types. As reference types are created dynamically, we consider all reference types as a single type: reference. We call instructions such as `GETSTATIC`: *overloaded instructions*.

When *SableVM* prepares the code array of a method, it replaces every overloaded bytecode by the appropriate type-specific versions such as `GETSTATIC_INT` and `GETSTATIC_REFERENCE`.

Here is the list of the most important overloaded Java bytecode instructions: `GETSTATIC`, `PUTSTATIC`, `GETFIELD`, `PUTFIELD`, `NEWARRAY`, and `ASTORE`. Note that we have included `ASTORE` in this list as it can operate on both *reference* and *address*-type stack values.

Stop-The-World or Not

A Java virtual machine must provide a garbage collector (GC). *SableVM* implements a precise copying *stop-the-world* garbage collector. A commonly used technique to stop the world, is for each Java thread to regularly check a flag. This flag is raised whenever garbage collection is needed.

To ensure that no thread gets into an arbitrarily long loop without checking for GC requests, GC checks are usually inserted in backward branch instructions. These instructions are usually said to be *garbage-collection safe*.

As *SableVM* already has to analyze the code to detect *basic blocks* for inline-threading, it also takes note of basic blocks which contain bytecode instructions that include compulsory GC checks. The following bytecode instructions have compulsory checks: `NEW`, `NEWARRAY`, `ANEWARRAY`, `MULTIANEWARRAY`, `INVOKESTATIC`, `INVOKEVIRTUAL`, `INVOKESPECIAL`, and `INVOKEINTERFACE`. Only backward branches to basic blocks which do not contain such instructions are considered GC check points.

Our technique is thus to provide two implementations for branch instructions: one with GC check, and one without GC check. This allows us to get an inlinable version (with no GC check), as shown in Figure 2.7.

`GOTO_CHECK`	`GOTO` (No Check)

`GOTO_CHECK:`
 	
`if (gc_requested)`
`{`
`  ...`
`}`
 	
`pc = (*pc)->addr;`
 	
`/* dispatch */`
`goto **(pc++);`
	
	`/* Inlinable */`
	`GOTO_START:`
	
	`pc = (*pc)->addr;`
	
	`GOTO_END:`
	
	`/* dispatch */`
	`goto **(pc++);`

Figure 2.7: Branch Instruction With and Without GC Check

Note that a branch instruction determines the end of a basic block, and is thus always followed by a dispatch. Inlining a branch instruction helps eliminating the dispatch at the end of the *previous* instruction.

A nice secondary side effect of only adding checks to a subset of backward branches is a reduction in the number of GC points, and possibly in the number of GC maps[11].

[11] We discuss garbage collection maps in Chapter 6.

Load, Link, and Initialize or Not

As we have discussed in Section 2.3.1 and illustrated in Figure 2.5, instruction splitting can also be applied to instructions that can cause class loading, linking, and initialization on their first execution. These instructions include[12]: `LDC_STRING`, `GETSTATIC_*`, `PUTSTATIC_*`, `GETFIELD_*`, `PUTFIELD_*`, `CHECKCAST`, `INSTANCEOF`, `INVOKESTATIC`, `INVOKEVIRTUAL`, `INVOKESPECIAL`, `INVOKEINTERFACE`, `NEW`, `ANEWARRAY`, and `MULTIANEWARRAY`.

Note that we have not yet addressed the two-values replacement problem that results from this splitting.

Using Signals

An additional technique to increase the length of inlinable instruction sequences is to eliminate explicit checks for NULL values.

This can be done in a portable manner using POSIX signals and ISO C *long jumps*. This NULL check technique is relatively well known, and in used in other virtual machines such as Kaffe [Kaf]. The idea is to setup a signal handler to trap segmentation faults, then to remove explicit NULL checks from the code. NULL pointers cause segmentation faults which are trapped by the signal handler, which in turns resumes normal execution using a `siglongjmp()` call.

The advantage of signal-based NULL checks is that, in absence of NULL pointers, a check costs 0 machine instructions. The drawback is that signals can be very expensive, as they seldom are the most optimized part of Operating Systems.

In the context of an inline-threaded interpreter, signal-based NULL checks carry the additional advantage of eliminating a conditional.

This is useful for instructions such as `GETFIELD`, as shown in Figure 2.8.

[12] Overloaded instructions are first split into type-specific versions.

Without Signals

```
GETFIELD_NO_INIT:
{
  int *instance =
    (pc++)->instance;
  int offset = (pc++)->offset;

  if (instance == NULL)
  {
    /* throw exception */
    ...
  }

  *sp++ = instance[offset];
}

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

With Signals

```
/* inlinable! */
GETFIELD_NO_INIT_START:
{
  int *instance =
    (pc++)->instance;
  int offset = (pc++)->offset;

  *sp++ = instance[offset];
}

GETFIELD_NO_INIT_END:

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

Figure 2.8: Using Signals

2.3.3 Preparation Sequences

Problems and Incomplete Solution

Our two most important problems left, at this point, are *two-values replacement*, and *shorter sequences* caused by the slow *preparation version*[13] of instructions such as `GETSTATIC`, as explained in Section 2.3.1.

Of course, there is a simple solution to two-values replacement that consists of using indirection in the *fast* version of instructions, as shown in Figure 2.9. Note how this implementation differs from Figure 2.6; in particular the additional `fieldinfo` indirection. This simple solutions comes at a price, though: that of an additional indirection in a very simple instruction. Furthermore, this solution does not solve the shorter sequences problem.

[13] We mean: the version which does all necessary first execution *preparation* work, such as class loading, linking and initialization.

Fast Instruction with Indirection	Code Array

GETSTATIC_NO_INIT:
{
  int *pvalue =
    (pc++)->fieldinfo->pvalue;
  *sp++ = *pvalue;
}
 	
/* dispatch */
goto **(pc++);
	/* Initially */
	...
	[GETSTATIC_INIT]
	[POINTER_TO_FIELD_INFO]
	...
	
	/* After first execution */
	...
	[GETSTATIC_NO_INIT]
	[POINTER_TO_FIELD_INFO]
	...

Figure 2.9: Single-Value Replacement of GETSTATIC

The Basic Idea

Instead, we propose a solution that solves both problems. This solution consists of adding *preparation sequences* in the code array.

The basic idea of preparation sequences is to duplicate certain portions of the code array, leaving fast inlined-sequences in the main copy, and using slower, synchronized, non-inlined preparation version of instructions in the copy. Single-value replacement is then used to direct control flow appropriately.

Single-Instruction Preparation Sequence

Preparation sequences are best explained using a simple illustrative example. We continue with our simplified GETSTATIC example[14]. We assume, for the moment, that the GETSTATIC is preceded and followed by non-inlinable instructions, in the code array. An appropriate instruction sequence would be MONITORENTER-GETSTATIC-MONITOR-EXIT, as neither monitor instruction is inlinable.

Figure 2.10 illustrates the initial content of a prepared code array containing the above 3-instructions sequence. The GETSTATIC *preparation sequence* appears at the end of the code array.

The initial content of the code array is as follows. After the MONITORENTER, we

[14] We assume the reader has noticed that in reality, our GETSTATIC example is implementing the type-specific GETSTATIC_INT overloaded version.

Original Bytecode	Initial Content of Code Array
... ... MONITORENTER GETSTATIC INDEXBYTE1 INDEXBYTE2 MONITOREXIT	... ... [MONITORENTER]* OPCODE_1: [GOTO]* [@ SEQUENCE_1] OPERAND_1: [NULL_POINTER] NEXT_1: [MONITOREXIT]* ... ... SEQUENCE_1: [GETSTATIC_INIT]* [POINTER_TO_FIELDINFO] [@ OPERAND_1] [REPLACE]* [GETSTATIC_NO_INIT] [@ OPCODE_1] [GOTO]* [@ NEXT_1] Opcodes followed by a * are instructions.

Figure 2.10: Single GETSTATIC Preparation Sequence

insert a GOTO instruction followed by two operands: (a) the address of the GETSTATIC *preparation sequence*, and (b) an additional word (initially NULL) which will eventually hold a pointer to the static field. At the end of the code array, we add a preparation sequence, which consists of 3 instructions (identified by a *) along with their operands.

Figure 2.11 shows the implementation of four instructions: GOTO, REPLACE, GETSTATIC_INIT, and GETSTATIC_NO_INIT. Notice that in the preparation sequence, the GETSTATIC_NO_INIT opcode is used as an operand to the REPLACE instruction.

We used labels (e.g. SEQUENCE_1:) to represent the address of specific opcodes. In the real code array, absolute addresses are stored in opcodes such as [@ SEQUENCE_1].

Here is how execution proceeds. On the first execution of this portion of the code, the MONITORENTER instruction is executed. Then, the GOTO instruction is executed, reading its destination in the following word. The destination is the SEQUENCE_1 label, or more accurately, the GETSTATIC_INIT opcode, at the head of the *preparation sequence*.

GETSTATIC_INIT

```
GETSTATIC_INIT:
{
  fieldinfo_t *fieldinfo =
    (pc++)->fieldinfo;
  int **destination =
    (pc++)->ppint;

  pthread_mutex_lock(...);

  /* lazily load and initialize
     class, and resolve field
     if not already done */
  ...

  /* store field information
   in code array */
  *destination =
    fieldinfo->pvalue;

  /* do the real work */
  *sp++ = *(fieldinfo->pvalue);

  pthread_mutex_unlock(...);
}

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

GETSTATIC_NO_INIT

```
GETSTATIC_NO_INIT:

/* skip address */
pc++;

{
  int *pvalue =
    (pc++)->pvalue;

  /* do the real work */
  *sp++ = *pvalue;
}

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

GOTO

```
GOTO:

{
  void *address =
    (pc++)->address;

  pc = address;
}

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

REPLACE

```
REPLACE:

{
  void *instruction =
    (pc++)->instruction;
  void **destination =
    (pc++)->ppvoid;

  *destination =
    instruction;
}

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

Figure 2.11: Instruction Implementations

The GETSTATIC_INIT instruction then reads two operands: (a) a pointer to the field information structure, and (b) a destination pointer for storing a pointer to the resolved static field. It then proceeds normally, loading and initializing the class, and resolving the field, if it hasn't yet been done[15]. Then, it stores the address of the resolved field in the destination location. Notice that, in the present case, this means that the pointer-to-field will *overwrite* the NULL value at label OPERAND_1. Finally, it executes the *real work* portion of the instruction, and dispatches to the next instruction.

The next instruction is a special one, called REPLACE, which simply stores the value of its first operand into the address pointed-to by its second operand. In this particular case, a pointer to the GETSTATIC_NO_INIT *instruction* will be stored at label OPCODE_1, overwriting the former GOTO instruction pointer. This constitutes, in fact, our *single-value* replacement.

The next instruction is simply a GOTO used to exit the *preparation sequence*. It jumps to the instruction following the original GETSTATIC bytecode, which in our specific case is the MONITOREXIT instruction.

Future executions of the same portion of the code array will see a GETSTATIC_NO_INIT instruction (at label OPCODE_1), instead of a GOTO to the *preparation sequence*. Two-values replacement is avoided by leaving the GOTO operand address in place. Notice how the implementation of GETSTATIC_NO_INIT in Figure 2.10 differs from the implementation in Figure 2.6, by an additional pc++ to skip the address operand.

Some Explanations

Our single-instruction preparation sequence has avoided two-values replacement by using an extra word to permanently store a *preparation sequence address* operand, even though this address is useless after initial execution.

This approach adds some overhead in the fast version of the *overloaded* instruction; that of a program-counter increment, to skip the preparation sequence address. One

[15] Each field is only resolved once, yet there can be many GETSTATIC instructions accessing this field. The same holds for class loading and initialization.

could easily question whether this gains any performance improvement over that of using an indirection as in Figure 2.9. This will be answered by looking at longer preparation sequences.

The strangest looking thing is the usage of 3 distinct instructions in the preparation sequence. Why not use a single instruction with more operands? Again, the answer lies in the implementation of longer *preparation sequences*.

Full Preparation Sequences

We now proceed with the full implementation of preparation sequences. Our objective is two fold: (a) we want to avoid two-values replacement, and (b) we want to build longer inlined instruction sequences for our inlined-threaded interpreter, for reducing dispatch overhead as much as possible.

To demonstrate our technique, we use the three instruction sequence: `ICONST2-GETSTATIC-ILOAD`.

Figure 2.12 shows the initial state of the code array. The content of the dynamically constructed `ICONST2-GETSTATIC-ILOAD` inlined instruction sequence, as well as some related instruction implementations are shown in Figure 2.13. Finally, the content of the code array after first execution is shown in Figure 2.14.

This works similarly to the single-instruction preparation sequence, with two major differences: (a) the jump to the *preparation sequence* initially replaces the `ICONST_2` instruction, instead of the `GETSTATIC` instruction, and (b) the `REPLACE` instruction stores a pointer to an *inlined instruction sequence*, overwriting the `GOTO` instruction.

Here is how execution proceeds in detail. On the first execution of this portion of the code, the `GOTO` instruction is executed. Its destination is the `ICONST_2` opcode, at the head of the *preparation sequence*.

Next, the `ICONST_2` instruction is executed. Next, the `GETSTATIC_INIT` instruction reads two operands: (a) a pointer to the field information structure, and (b) a destination pointer for storing a pointer to the resolved static field. It then proceeds normally, loading and initializing the class, and resolving the field, if it hasn't yet

Original Bytecode	Initial Content of Code Array
	...
	...
	OPCODE_1: [GOTO]*
	[@ SEQUENCE_1]
	OPERAND_1: [NULL_POINTER]
	[INDEX]
...	NEXT_1: ...
...	...
ICONST_2	...
GETSTATIC	SEQUENCE_1: [ICONST_2]*
INDEXBYTE1	[GETSTATIC_INIT]*
INDEXBYTE2	[POINTER_TO_FIELDINFO]
ILOAD	[@ OPERAND_1]
INDEX	[REPLACE]*
...	[ICONST2-GETSTATIC-ILOAD]
...	[@ OPCODE_1]
	[ILOAD]*
	[INDEX]
	[GOTO]*
	[@ NEXT_1]
	Opcodes followed by a * are instructions.

Figure 2.12: Full Preparation Sequence

ICONST2-GETSTATIC-ILOAD Inlined Instruction Sequence
SKIP body : pc++;
ICONST_2 body : *sp++ = 2;

GETSTATIC_NO_INIT body: {int *pvalue = (pc++)->pvalue;
*sp++ = *pvalue;}

ILOAD body : {int index = (pc++)->index;
*sp++ = locals[index];}

DISPATCH body : goto **(pc++);

SKIP	GETSTATIC_NO_INIT
	GETSTATIC_NO_INIT_START:
SKIP_START:	{
	int *pvalue = (pc++)->pvalue;
*pc++;	*sp++ = *pvalue;
	}
SKIP_END:	
	GETSTATIC_NO_INIT_END:
/* dispatch */	
goto **(pc++);	/* dispatch */
	goto **(pc++);

Figure 2.13: Inlined Instruction Sequence

```
              ...
              ...
OPCODE_1:     [ICONST2-GETSTATIC-ILOAD]*
              [@ SEQUENCE_1] (skipped)
OPERAND_1:    [POINTER_TO_FIELD] (for GETSTATIC)
              [INDEX] (for ILOAD)
NEXT_1:       ...
              ...
              ...
SEQUENCE_1: [ICONST_2]*
              [GETSTATIC_INIT]*
              [POINTER_TO_FIELDINFO]
              [@ OPERAND_1]
              [REPLACE]*
              [ICONST2-GETSTATIC-ILOAD]
              [@ OPCODE_1]
              [ILOAD]*
              [INDEX]
              [GOTO]*
              [@ NEXT_1]

Opcodes followed by a * are instructions.
```

Figure 2.14: Code Array After First Execution

been done. Then, it stores the address of the resolved field in the destination location. Finally, it executes the *real work* portion of the instruction, and dispatches to the next instruction.

The next instruction is a `REPLACE`, which simply stores a pointer to the dynamically *inlined instruction sequence* `ICONST2-GETSTATIC-ILOAD` at label `OPCODE_1`, overwriting the former `GOTO` instruction, and performing a *single-value* replacement.

Next, the `ILOAD` instruction is executed. Finally, the tail `GOTO` exits the *preparation sequence*.

Future executions of the same portion of the code array will see the `ICONST2-GETSTATIC-ILOAD` instruction sequence (at label `OPCODE_1`), as shown in Figure 2.14. Notice that the *inlined implementation* of `GETSTATIC_NO_INIT` in Figure 2.13 does not add any overhead to the fast implementation shown in Figure 2.6.

Thus, we have achieved our goals. In particular, we have succeeded at inlining an instruction sequence, even though it had a complex two-modes (preparation / fast) instruction in the middle, while avoiding two-values replacement. All of this with minimum overhead in post-first execution of the code array.

Detailed Preparation Procedure

Preparation of a code array, in anticipation of inline-threading, proceeds as follows:

1. Instructions are divided in three groups: inlinable, two-modes-inlinable (such as `GETSTATIC`), and non-inlinable.
2. Basic blocks (determined by control-flow and non-inlinable instructions) are identified.
3. Basic blocks of inlinable instructions, without two-modes-inlinable instructions, are inlined as explained in Section 2.3.
4. Every basic block containing two-modes-inlinable instructions causes the generation of an additional *preparation sequence* at the end of the code array, and the construction of a related *inlined instruction sequence*.

The construction of a *preparation sequence* proceeds as follows:

1. Instructions are copied sequentially into the preparation sequence.
    - Inlinable instructions and their operands are simply copied as-is.
    - The *preparation* version of two-modes-inlinable instructions is copied into the preparation sequence, along with the destination address for resolved operands.
2. A `REPLACE` instruction with appropriate operands is inserted just after the last two-modes-inlinable instruction.
3. A final `GOTO` instruction with appropriate operand is added at the end of the preparation sequence.

The motivation for adding the replace instruction just after the the last two-modes-inlinable instruction, is that it is the earliest safe place to do so. Replacing sooner could cause the execution (on another Java thread) of the fast version of an upcoming two-modes instruction before it is actually prepared. Replacing later can also be a problem, specially if some upcoming inlinable instruction is a conditional (or unconditional) branch instruction. This is because, if the branch is taken, then single-value replacement will not take place, forcing the next execution to take the slow path[16].

The construction of an *inlined instruction sequence* containing two-modes-inlinable instructions proceeds as follows:

1. The body of the `SKIP` instruction is copied at the beginning of the sequence implementation.

2. Then, all instruction bodies are sequentially copied.

3. Finally, the body of the `DISPATCH` instruction is copied at the end of the sequence implementation.

Note that a single preparation sequence can contain multiple two-modes instructions. Yet, on the fast execution path, there is a single program-counter increment (i.e. `SKIP` body) per inlined instruction sequence.

Adjusting Exception and Line Number Tables

Implementing *preparation sequences* involves many little details related to computation of absolute addresses. We will only discuss briefly of the trickiest issue, that of exception handling within preparation sequences.

Java class files include *exception tables* which determine the flow of control when exceptions are thrown. Each entry in an exception table specifies: (a) an instruction

[16] Multiple executions of the same *preparation sequence* is allowed, but suffers from high dispatch overhead. It *can* happen in the normal operation of the inline-threaded interpreter as the result of an exception thrown before single-value replacement, while executing a *preparation sequence*.

range using a start and end offset in the bytecode array, (b) a catch reference type (or *any*), and (c) an exception handler address (an offset in the bytecode array).

When an exception happens, the exception table of the currently executing method is searched *sequentially* for a matching entry. Order does matter, as there might be more than one matching entry in the table, yet the first one *must* be chosen. If a matching entry is found, execution resumes at the specified exception handler address. If none is found, the current stack frame is popped, and the process is repeated recursively.

The consequence is that, in the presence of preparation sequences, exception tables require some additional preparation work. An entry in the exception table might specify a range of instructions which contains *two-modes* instructions. As preparation sequences are added to the end of the code array, we have modified the structure of an exception table entry to include two ranges: one range in the lower part of the code array, and one range in the preparation part of the code array. To process exceptions, the modified exception table is simply searched *sequentially* for a matching entry, as usual, with the difference that the *program counter* must be within one of the two ranges for an entry to match.

An identical modification is applied to line number table entries, which are used to report line numbers in exception stack traces.

2.4 Experimental Results

We have implemented 3 flavors of threaded interpretation in the *SableVM* framework: switch-threading, direct-threading and inline-threading. *Switch-threading* differs from simple switch-based bytecode interpretation in that it is applied on a *prepared* code array of word-size elements. All of the techniques introduced in this chapter are in use within the inline-threaded interpreter engine. Some of the techniques are also in use within the switch-threaded and direct-threaded engines, including *single-instruction preparation sequences*, to avoid the problem of two-values replacement.

The test environment and choice of benchmarks is discussed in Chapter 9. In summary, we have performed our experiments on a Pentium IV based workstation, running SPECjvm98 benchmarks and two object-oriented applications: Soot version 1.2.3[17] and SableCC version 2.17.3[18].

2.4.1 Inlined Instruction Sequences Characteristics

Table 2.1 shows a first set of measurements. The main objective of these measurements was to quantify the proportion of inlinable instructions, within bytecode arrays, and to measure the average length of inlined sequences.

benchmark	methods	instr. (bc)	seq.	inl. instr.	ins./s.
compress	411 (41)	16,886 (30,901)	4,238	12,263 (73%)	2.9
db	461 (40)	18,283 (34,255)	4,804	13,083 (72%)	2.7
jack	689 (49)	33,695 (62,791)	9,095	23,678 (70%)	2.6
javac	1,238 (38)	47,484 (101,027)	12,169	33,415 (70%)	2.7
jess	892 (31)	27,220 (52,985)	7,331	19,025 (70%)	2.6
mpegaudio	581 (81)	47,145 (76,509)	11,449	34,405 (73%)	3.0
mtrt	588 (38)	22,628 (42,037)	5,896	15,833 (70%)	2.7
raytrace	583 (39)	22,531 (41,863)	5,876	15,758 (70%)	2.7
soot	3,475 (30)	104,781 (223,557)	29,259	67,030 (64%)	2.3
sablecc	1,701 (29)	49,923 (93,335)	13,602	32,723 (66%)	2.4

Table 2.1: Inlined Sequences (1)

Our measurements were made only on methods that were actually executed. As *SableVM* prepares methods lazily (on first execution of a method), we collected our measurements at runtime, just after the preparation of methods. Thus the shown numbers are *preparation time* numbers.

Columns of Table 2.1 contain respectively: (a) the name of the executed benchmark, (b) the number of prepared methods, and the average number of instructions per method in parentheses, (c) the total number of instructions of prepared methods, and the total number of bytecodes of these instructions in parentheses, (d) the total number of inlined sequences of all prepared methods, (e) the total number of

[17] `http://www.sable.mcgill.ca/soot/`

[18] `http://www.sablecc.org/`

inlined instructions, and the percentage of inlined instructions over the total number of instructions, and (f) the average number of instruction per inlined sequence.

Note that the number of instructions and bytecodes are different quantities; a single instruction can consist of multiple bytecodes representing the instruction opcode and operands.

In our measurements, the average length of inlined instruction sequences lies between 2.3 and 3.0. The ratio of inlined instructions is between 66% and 73% of all instructions.

Table 2.2 shows additional characteristics of inlined sequences. The main objective of these measurements was to quantify the memory requirement for storing inlined sequences, and identify the longest sequence in terms of number of instructions and inlined implementation size.

benchmark	**seq.**	**distinct seq.**	**max.ins./s.**	**max.size/s.**
compress	4,238	850 (156K)	41	1,686
db	4,804	900 (155K)	22	999
jack	9,095	1,174 (213K)	22	1,090
javac	12,169	2,200 (426K)	25	999
jess	7,331	1,312 (234K)	22	999
mpegaudio	11,449	1,293 (251K)	85	3,469
mtrt	5,896	1,175 (198K)	32	1,362
raytrace	5,876	1,165 (196K)	32	1362
soot	29,259	2,906 (574K)	54	2,076
sablecc	13,602	1,484 (277K)	22	999

Table 2.2: Inlined Sequences (2)

Columns of Table 2.2 contain respectively: (a) the name of the executed benchmark, (b) the total number of inlined instruction sequences of all prepared methods, (c) the number of distinct inlined instruction sequences, (d) the highest number of instructions within a single inlined sequence, and (e) the biggest implementation size (in bytes) of a single inlined sequence.

SableVM saves space by allocating a single copy for each distinct *inlined instruction sequence*. This proves very effective, specially for bigger benchmarks. For the *Soot* benchmark, in particular, *SableVM* does not use additional storage space for over 90% of inlined instruction sequences. The total size of genuine inlined instruction sequences is 574K. If we divide this storage size by the total number of inlined instructions (67,030), found in Table 2.1, we find that, on average, less than 9 bytes of implementation code is required per instruction in the *Soot* benchmark.

2.4.2 Performance Measurements

We have performed execution time measurements with *SableVM* (within the test environment described in Chapter 9), to measure the efficiency of inline-threading Java, using our techniques.

In a first set of experiments, we have measured the relative performance of the switch-threaded, direct-threaded and inline-threaded engines. Results are shown in Table 2.3. To do these experiments, three separate versions of *SableVM* were compiled with identical configuration options, except for the interpreter engine type. In particular, the usage of signals to trap NULL pointer exceptions was turned on in all three versions.

benchmark	**switch-threaded**	**direct-threaded**		**inline-threaded**		
	(sec.)	(sec.)		(sec.)		
compress	317.72	281.78	(1.13)	131.64	(2.41)	(2.14)
db	132.15	119.17	(1.11)	87.64	(1.51)	(1.36)
jack	45.65	46.78	(0.98)	38.16	(1.20)	(1.23)
javac	110.10	105.24	(1.05)	89.37	(1.23)	(1.17)
jess	74.79	68.12	(1.10)	53.57	(1.40)	(1.27)
mpegaudio	285.77	242.90	(1.18)	136.97	(2.09)	(1.77)
mtrt	142.87	115.34	(1.24)	100.39	(1.42)	(1.15)
raytrace	166.19	134.06	(1.24)	113.55	(1.46)	(1.18)
soot	676.06	641.96	(1.05)	548.13	(1.23)	(1.17)
sablecc	40.12	36.95	(1.09)	26.09	(1.54)	(1.41)

Table 2.3: Inline-Threading Performance Measurements (1)

Columns of Table 2.3 contain respectively: (a) the name of the executed benchmark, (b) the execution time in seconds using the switch-threaded engine, (c) the execution time in seconds using the direct-threaded engine, and the speedup over the switch-threaded engine in parentheses, and (d) the execution time in seconds using the inline-threaded engine, and the speedup over both switch-threaded and direct-threaded engines respectively in parentheses.

The *Inline-threaded* engine does deliver significant performance improvement. It achieves a speedup of up to 2.41 over the switch-threaded engine. The smallest measured speedup, over the *fastest* of the two other engines on a benchmark, is 1.15 on the *mtrt* benchmark, where it also delivers a speedup of 1.42 over the slower engine.

It is important to note that the switch-threaded engine already has some advantages over a pure switch-based bytecode interpreter. It benefits from word alignment and other performance improving features of the *SableVM* framework. So, it is likely that the performance gains of inline-threading over pure bytecode interpretation are even bigger than those measured against switch-threading. In Chapter 9, we measure the relative performance of *SableVM* against a naively implemented bytecode interpreter.

In a second set of tests, we measured the speed of the inlined-threaded engine when using signal-based NULL pointer detection and when using explicit NULL checks. Results are shown in Table 2.4.

benchmark	**explicit NULL checks** (sec.)	**signal-based NULL checks** (sec.)	**speedup**
compress	166.41	131.64	1.26
db	92.10	87.64	1.05
jack	39.85	38.16	1.04
javac	91.29	89.37	1.02
jess	56.51	53.57	1.05
mpegaudio	145.31	136.97	1.06
mtrt	96.02	100.39	0.96
raytrace	109.23	113.55	0.96
soot	606.86	548.13	1.11
sablecc	28.04	26.09	1.07

Table 2.4: Inline-Threading Performance Measurements (2)

Columns of Table 2.4 contain respectively: (a) the name of the executed benchmark, (b) the execution time in seconds using the inline-threaded engine and explicit NULL checks, (c) the execution time in seconds using the inline-threaded engine and signal based NULL pointer detection, and (d) the speedup achieved by signal-based detection over explicit checks.

Note that using signal-based checks can help eliminate up to two dispatches within an instruction sequence, by making an instruction *inlinable.* The drawback is that signals are costly, so if NULL pointer exceptions effectively happen, the performance of an application can be negatively affected.

Our performance measurements show that using signal-based detection can yield significant speedup (up to 1.26 on *compress*), but can also reduce performance (a penalty of 4% on *mtrt* and *raytrace*).

2.5 Related Work

The most closely related work to the work of this chapter is the work of I. Piumarta and F. Riccardi in [PR98]. We have already discussed the inline-threading technique introduced in this paper in Section 2.3. Our work builds on top of this work, by introducing techniques to deal with *multi-threaded* execution environments, and inline *two-modes* instructions.

Inline-threading, in turn, is the result of combining the Forth-like *threaded interpretation* technique [Ert] (which we have already discussed in Section 2.2.2) with the idea of *template-based* dynamic compilation [APC$^+$96, NHCL98]. The main advantage of inline-threading over that of template based compilation is its simplicity and portability.

A related system for dynamic code generation is that of *vcode*, introduced by D. Engler [Eng96]. The *vcode* system is an architecture-neutral runtime assembler. It can be used for implementing *just-in-time* compilers. It is in our future plans to experiment with *vcode* for constructing an architecture-neutral *just-in-time* compiler for *SableVM*, offering an additional choice of performance-portability tradeoff.

Other closely related work is that of *dynamic patching.* The problem of potential high cost synchronization costs for concurrent modification of executed code is also faced by dynamically adaptive Java systems. In [CLS00], M. Cerniac *et al.* describe a technique for *dynamic inline patching* (a similar technique is also described in [IKY$^+$00]). The main idea is to store a self-jump (a jump instruction to itself) in the executable code stream before proceeding with further modifications of the executable code. This causes any concurrent thread executing the same instruction to spin-wait for the completion of the modification operation.

Our technique of using explicit synchronization in preparation sequences and single value replacement has the marked advantage of causing no spin-wait. Spinning can have, in some cases, a highly undesirable side effect, that of *almost* dead-locking the system when the spinning thread has much higher priority than the *code patching* thread. This is because, while it is spinning, the high priority thread does not make any progress in code execution and, depending on the thread scheduling policy of the host operating system, might be preventing the patching thread from making noticeable progress.

2.6 Conclusions

In this chapter we have explained the difficulty of using the *inline-threaded* interpretation technique in a Java interpreter. Then, we introduced new techniques that not only make it possible, but also effective. At the heart of our techniques is the idea of *preparation sequences*, which when combined with other techniques, help increase the length of *inlined instruction sequences* and thus reduce *dispatch* overhead.

We then presented our experimental results, showing that an inline-threaded interpreter engine, implementing our techniques, achieves significant performance improvements over that of switch-threaded and direct-threaded engines.

Chapter 3
Logical Partitioning of Memory

Memory management is a central issue in the design of a Java virtual machine. Many of the runtime services provided by the virtual machine have memory management related requirements: garbage-collected heap, Java stacks, dynamic class loading, dynamic linking, *JNI* native references, structures for dynamic dispatch (e.g. virtual tables), etc.

In this chapter we discuss the organization of memory in the *SableVM* runtime environment. In particular, we introduce a logical partitioning of the runtime system memory which allows the design of simple and flexible, yet effective, partition-specific memory managers.

This chapter is structured as follows. In Section 3.1, we motivate *logical memory partitioning* as a technique for simplifying the internal organization of a Java virtual machine. In Section 3.2, we introduce our partitioning of the Java runtime memory, and discuss the related partition-specific memory managers. In Section 3.3 we discuss related work, and finally in Section 3.4 we present our conclusions.

3.1 Simplifying Memory Management

3.1.1 The Price of Flexibility

One of the key features that must be provided by a Java virtual machine is a garbage-collected heap for object instances. Yet, memory management in the virtual machine is much broader than simply providing an object heap; the virtual machine must also *explicitly* or *implicitly* manage memory for local and global JNI references, invocation and native interface data structures, class and array information data structures, fat locks, inlined instruction implementations (for *inlined-threading*) or compiled code (for *just-in-time* compilation), virtual tables, stacks and many other features.

Among the objectives of the *SableVM* framework is to be easily modifiable and flexible, allowing the research on memory management techniques. It is thus important that *SableVM* be compatible with various garbage collection algorithms, including precise, mostly-accurate, and conservative *moving* and *non-moving* algorithms. Such flexibility comes at a price.

Single Garbage-Collected Heap

A memory organization that would intuitively seem simple is to allocate all (or most) memory in the garbage-collected heap. But, many allocated memory sections have special requirements. For example, virtual tables cannot be allocated in a movable heap, unless *pinning*[1] is supported by the garbage collector or a complex code patching process is applied every time a virtual table is moved. But either alternative adds complexity to the virtual machine. Similarly, allocating stack frames on the heap could increase the pressure on the garbage collector by causing frequent collections, unless some special care is taken to minimize the pressure (e.g. generational collection or special treatment for stack frame allocation). In summary, a single garbage-collected heap comes at the cost of increased complexity of the virtual machine.

[1] A *pinned* memory object is left in place by a moving garbage collector.

Using `malloc()` and `free()`

Another relatively intuitive memory organization is to manage a separate garbage-collected heap for object instances, and allocate all other memory using the `malloc()` C library function. This organization has the advantage of permitting the implementation of various garbage collection algorithms including the simplest ones such as copying collection without pinning. But such organization can be expensive in both space and time. On the *space* front, the cost is that every allocated memory block returned by `malloc()` requires some overhead memory space for storing the block size (and possibly additional information) in anticipation of future `free()` calls. As many allocated blocks are very small (e.g. JNI references are a few words long, and some *class information* related memory blocks can be as short as a single word), this overhead could be significant. On the *time* front, the costs are that `malloc()` and `free()` calls cause global synchronization[2], and that `free()` calls can get very expensive when many small blocks are freed successively and incremental aggregation of freed memory takes place to fight memory fragmentation.

Reducing Complexity and Costs

As we have seen, maintaining a separate garbage-collected heap for object instances and using `malloc()` and `free()` for the rest increases flexibility without adding noticeable complexity to the virtual machine architecture. The main problem related to that approach is the C library overhead when managing small memory blocks.

`malloc()` and `free()` were designed as general purpose memory allocation functions. By studying the typical memory usage of various virtual machine features, we discover patterns in memory usage. For example, a Java stack always allocates and frees its *top frame*. Stack frames are never accessed by other threads (except possibly by a garbage collection thread). It thus seems wasteful to pay the overhead of a general purpose memory manager for allocating and freeing stack frames. A similar analysis can be done of other features.

[2] Note that the virtual machine *must* be linked with the multi-threaded version of the C library

Therefore, our solution for reducing overhead without increasing system complexity is to partition the Java runtime memory according to usage patterns, and to design simple partition-specific memory managers that take advantage of usage patterns to minimize overhead.

3.2 Logical Memory Partitions

We have studied the memory usage behavior of the different virtual machine features, and identified distinct allocation and release patterns. These patterns offer a natural partitioning of the runtime memory that we present in this section.

We first draw a clear distinction between *physical* and *logical* partitions, then we introduce each identified partition and discuss its management strategy within the *SableVM* framework.

3.2.1 Physical and Logical Partitions

We define a *physical partition* as a single contiguous segment of *virtual memory* managed by a single memory manager. In contrast, we define a *logical partition* as the subset of virtual memory which is managed by a single memory manager. Memory in the subset need not be contiguous.

Figure 3.1 illustrates the difference between physical and logical partitions. In the left side of the figure, we see a partitioning of virtual memory into physical partitions, each of which is contiguous. In the right side of the figure, we see the physical layout of a *logical* partitioning of the same memory. On this side, each logical partition is not necessarily contiguous. For example, the single *Stack* logical partition is divided into three distinct memory segments.

Managing *logical* partitions, instead of *physical* ones, is very convenient. It allows for dynamically creating and deleting partitions, and for growing and shrinking them without worrying about low-level memory layout details. For increasing the portability of the virtual machine, the standard `malloc()` and `free()` library functions are used as low-level primitives by memory managers for allocating and releasing *aligned*

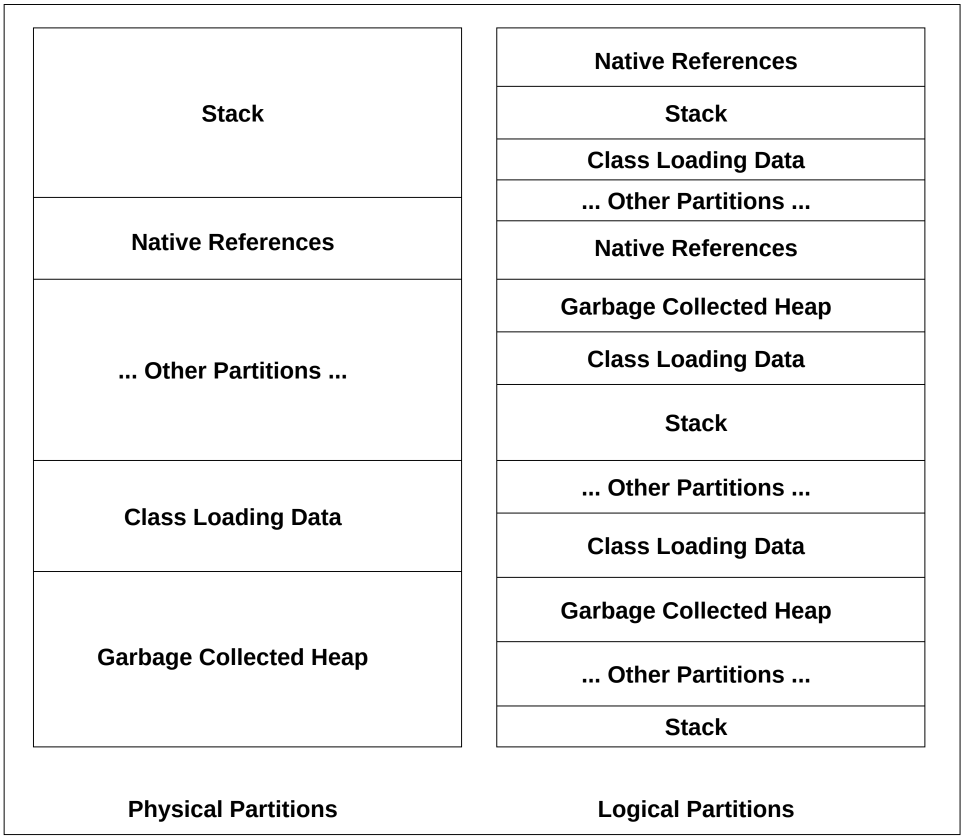


Figure 3.1: Layout of Physical and Logical Partitions

memory blocks[3].

3.2.2 Thread-Specific Memory

Thread-specific memory consists of all memory specifically allocated by the virtual machine for the internal management of each Java thread.

This memory consists primarily of Java stacks, JNI local reference frames, and internal structures storing thread-specific data like stack information, JNI virtual table, and exception status.

This memory exhibits precise allocation and release patterns. Thread-specific structures have a lifetime similar to their related thread. So, this memory can be allocated and freed (or recycled) at the time of respective creation and death of the underlying thread. Stacks do not conceptually suffer from fragmentation, as they grow and shrink on one side only. This property is shared by JNI local reference frames.

SableVM Implementation

SableVM allocates thread structures on thread creation but does not release them at thread death. Instead, it manages a free list to recycle this memory on future thread creation.

Java stack memory is managed differently. For each thread, an initial stack is allocated (using `malloc()`) at the time the thread is created. Then this stack is expanded (using `realloc()`) as required by the computation. *SableVM* never shrinks the size of a Java stack. It simply frees it on thread death. Initial allocation size and growth are controlled through runtime parameters. A maximum stack size can be optionally specified; by default a stack is allowed to grow until memory exhaustion. Note that all Java stack frame information, in *SableVM*, is stored relatively (e.g. offset to previous/next frame, instead of direct address), which enables stacks to be moved.

[3] The ANSI/ISO C standard states that `malloc()` must return aligned memory [SAI^{+}90].

The advantage of a stack whose maximum size can only grow, is simpler stack memory management at method call boundaries. It also simplifies the virtual machine usage, as the user needs not specify a maximum stack size, which can be difficult to do for each thread of a multi-threaded application. There's no need to manage the potential fragmentation of non-movable stacks. It would be possible to allocate Java stack frames on the shared garbage-collected Java heap but this would put unnecessary additional pressure on the garbage collector, and it would not take advantage of the highly regular allocation and release behavior of this memory.

Some Measurements

In order to determine compile-time default values for initial stack size and increment, we have measured the maximum stack depth reached while executing each of the benchmarks identified in Section 2.4.

Table 3.1 shows our measurements. The deepest measured stack depth is 9612 bytes, which is surprisingly small.

benchmark	**max. Java stack depth**
compress	2740 bytes
db	2740 bytes
jack	3356 bytes
javac	7572 bytes
jess	4756 bytes
mpegaudio	3684 bytes
mtrt	2896 bytes
raytrace	2896 bytes
soot	8408 bytes
sablecc	9612 bytes

Table 3.1: Stack Depth

As the stack size is highly application (and input data) specific, we have chosen to set both compile-time default initial size and increment to 64Kb. Of course, these values can be overridden using command-line options. *SableVM*'s stack-related command-line options are:

- `-p sablevm.stack.size.min=SIZE` : Minimum size in bytes.

- `-p sablevm.stack.size.max=SIZE` : Maximum size in bytes.
- `-p sablevm.stack.size.increment=SIZE` : Maximum increment size in bytes.

For example, to set the stack increment size to 32Kb, we would write the following command:

```
sablevm -p sablevm.stack.increment=32768 HelloWorld
```

3.2.3 Class-Loader-Specific Memory

Class-loader-specific memory consists of all memory specifically allocated by the virtual machine for the internal management of each class loader.

This memory consists primarily of the internal data structures used to store class loaders (except the related `java.lang.ClassLoader` instances, which are stored in the garbage-collected heap), classes and methods, and related information. This includes method bodies in their various forms like bytecode, direct threaded code, inlined threaded code, and compiled code (in the presence of a JIT). It also includes virtual tables for dynamic method dispatch.

This memory exhibits precise allocation and release patterns. It is allocated at class-loading time, and at various preparation, verification, and resolution execution points. The behavior of this memory differs significantly from other memory in that once it is allocated, it must stay at a fixed location, and it is unlikely to be released soon. The Java virtual machine specification allows for potential unloading of all classes of a class loader as a group, if no direct or indirect references to the class loader, its classes, and related object instances remain. In such a case, and only if a virtual machine supports class unloading, all memory used by a class loader and its classes can be released *at once*.

Isolating the management of this memory is a distinctive feature of the *SableVM* framework.

SableVM Implementation

SableVM manages this memory on a *class loader* basis. In other words, each class loader has its own related memory manager.

A *class-loader memory manager* uses `malloc()` to allocate chunks of memory, and provides its own allocator for distributing smaller memory fragments. This has many advantages.

It allows for the allocation of many small memory blocks without the usual cost of memory space overhead, discussed in Section 3.1.1. Also, memory chunks can be freed on class unloading without regard for internal sub-allocation, thus significantly reducing the number of `free()` calls and the related memory aggregation penalty.

It also allows class parsing and decoding in one pass without memory overhead, by allocating many small memory blocks. This is usually not possible, as it is not possible to estimate the memory requirement for storing internal class information before the end of the first pass. The usual alternatives (in absence of a class-loader specific memory manager) are to either pay a memory overhead for allocating small blocks using `malloc()`, or do 2 passes over the class file; one pass to compute memory requirements, and the second to extract the information and store it in the allocated memory. But even then, the second alternative does not solve the problem of many small allocations which are required to store threaded or compiled code, and other linking information computed lazily throughout execution.

Finally, and importantly, it allows for irregular memory management strategies: it makes it possible to return sub-areas of an allocated block to the allocator, if these areas are known not to be used. We take advantage of this to allocate sparse interface method lookup tables without losing memory[4].

A default chunk size and increment, as well as an optional maximum class-loader memory size, can be specified on the command-line[5]. The compile-time default chunk size and increment have both been *arbitrarily* set to 1Mb. Note that many applications only use two class loaders: the special bootstrap class loader[6] and the *application* (or system) class loader[7]. For running programs using many user-defined class loaders,

[4] This will be explained in Chapter 4.

[5] The options are: `-p sablevm.classloader.heap.size.[min|max|increment]=SIZE`

[6] The bootstrap class loader is used to load standard library classes such as `java.lang.*` and `java.io.*`.

[7] The application class loader is used to load classes found on the *classpath*.

users are encouraged to set default values appropriately.

3.2.4 Shared Memory

Shared memory consists of remaining memory which is explicitly managed by the virtual machine[8].

This memory consists primarily of the object instance heap (which is garbage collected), global JNI references, and global virtual machine information structures.

The allocation and release behavior of this memory exhibits no specific pattern, as it is highly application dependent.

This memory is potentially allocated and modified by different threads while executing methods of classes loaded by various class loaders.

SableVM Implementation

SableVM manages the object instance heap separately from other memory. This provides maximum flexibility for testing various garbage collection algorithms. In the current version of *SableVM*, a precise bare-bones copying garbage collector (with no pinning nor generations) is provided. Chapter 6 will discuss the algorithm to compute the necessary maps for precise garbage collection. The parameters of the copying collector provided by *SableVM* can be controlled using command-line options. In particular, minimum and maximum heap size, and increment can be specified. If no maximum is specified, *SableVM* will potentially grow the heap until memory exhaustion. *SableVM* does also shrink the heap. Its algorithm for determining heap size aims to keep the heap 1/3 full (as suggested in [JL96]).

Most of the remaining memory is simply managed using `malloc()` and `free()` calls. Exceptions to this include *global JNI references* (and similar small structures) which have a special memory manager. This manager allocates big memory blocks, divides them into small JNI reference structures, and manages free lists to avoid `free()` memory aggregation overhead.

[8] This excludes memory which is not explicitly managed by the virtual machine.

3.2.5 System Memory

System memory is the remaining memory, on which we have essentially no control. It consists of the memory used for virtual machine code (i.e. the executable itself), native stacks, ANSI/ISO C's heap manager, and dynamically-linked native libraries, as well as any other uncontrollable memory.

SableVM Implementation

As *SableVM* is written in portable C, it has no control on the management of this memory. Assembly and system specific virtual machines can (and should probably) manage this memory explicitly, in which case it could be classified among the previously identified partitions. For example, dynamically-linked native libraries would be classified as class-loader specific memory.

The only attempt to control this memory is the following. *SableVM* manages its own Java stacks (one per thread), and it *does not* use a C function call to implement Java method calls. Thus, it makes a minimal use of the native C stack. Recursive C function calls are only possible through native JNI calls[9].

3.3 Related Work

The traditional single-threaded runtime memory organization for procedural languages (C, Pascal, etc.) is as follows. Executable code and constant data segments are laid out consecutively at lower memory addresses. The remaining space is divided between the heap, growing bottom up, and the stack, growing top down, with free space in the middle [ASU86]. For multi-threaded applications, multiple stacks are required (one per thread). POSIX threads creation parameters include an optional stack size and assume otherwise an implementation specific default size [NBF96]. A typical implementation places stacks sequentially (top down) reserving enough space

[9] A native JNI method is allowed to invoke the virtual machine interpreter which may in turn call back the JNI native method.

for each stack, and optionally guarding against overflow using a protected page (for causing a segmentation fault in case of stack overflow).

The most naive virtual machines, like Kaffe [Kaf], implement the simplest approach to memory management in C by using `malloc()` for all allocation, and using a conservative garbage collector for freeing memory. This approach severely limits potential experimentation with garbage collection techniques. Kaffe provides no support for precise collection, and also assumes that objects cannot be moved throughout their life time. This latest assumption allows Kaffe to save some object storage by using object addresses as hashcode.

Other more elaborate virtual machines, such as Jikes RVM which entirely is written in Java, use a single heap to manage all memory [AAC+99], except *system memory* used to store the precomputed boot image. This approach would seem attractive, at first sight, but it has some drawbacks. Firstly, some memory areas, such as JIT compiled code, cannot be moved in memory, forcing the garbage collector to either be non-moving or to support pinning. Secondly, this approach does not allow for irregular memory management where a single allocated block is not contiguous (like *SableVM*'s sparse interface tables). Finally, even if a memory block is pinned within a generational heap, it might still cause some additional work at garbage collection time. This is unlike *SableVM*'s class-loader specific memory which is never traced at garbage collection time, regardless of what garbage collection algorithm is used.

3.4 Conclusions

In this chapter we have identified logical memory partitions exhibiting distinct allocation and release behavior. We have discussed each partition and explained how the *SableVM* framework manages memory within it.

We have in particular identified a class-loader specific memory partition. This memory may not be moved, and it is either never freed or freed *all at once* on class-loader destruction. We have explained how *SableVM* takes advantage of this to sub-allocate small memory blocks without space overhead. We have briefly mentioned that *SableVM* also permits irregular allocation strategies within this partition for sparse interface virtual tables. Using a specific memory management strategy for class-loader specific memory is a distinctive feature of the *SableVM* framework.

Using partition-specific memory managers allows *SableVM* to be extremely flexible (it is compatible with various garbage collection algorithms, ranging from the simplest to the most complex ones). These managers minimize overhead even though they use relatively simple operational strategies.

Chapter 4
Sparse Interface Virtual Tables

In this chapter we introduce a sparse virtual table design that eliminates the overhead of interface method invocation over that of normal virtual method invocation. This design takes advantage of class-loader specific memory management[1] to recycle holes in the virtual table using a very simple, yet effective, algorithm. Our experimental results show a 100% recycle rate for sparse virtual table holes, even in the most *interface intensive* applications tested.

This chapter is organized as follows. In Section 4.1, we discuss the traditional organization of virtual tables in Java virtual machines, and discuss related performance problems. In Section 4.2, we introduce our sparse virtual table organization. In Section 4.3, we present our experimental results. In Section 4.4 we discuss related work. Finally, in Section 4.5, we present our conclusions.

4.1 Traditional Virtual Table Organization

One of the distinctive features of object-oriented programming languages, relative to procedural programming languages, is virtual function calls (or polymorphic calls). Efficiently implementing polymorphic calls has been a very active research field. In [Dri01], K. Driesen surveys most of the implementation techniques that have been proposed and used in various object-oriented programming languages.

[1] See Chapter 3.

In this section, we present the traditional (and intuitive) organization of *virtual tables* for implementing both virtual and interface method calls in Java. In the context of this thesis, *virtual method calls* and *interface method calls* correspond respectively to the `INVOKEVIRTUAL` and `INVOKEINTERFACE` Java bytecode instructions.

4.1.1 Virtual Tables for Single Inheritance

Java is a statically typed language (i.e. all variables have compile-time types), but its classes are dynamically loaded and linked at execution time. Luckily, the Java virtual machine specification imposes constraints on loaded classes (and bytecode) to ensure proper runtime behavior.

Java supports single inheritance of classes. This enables the efficient implementation of *virtual* function calls using *virtual method tables* (or simply: virtual tables).

The virtual table of a class is an array of pointers to methods. Each virtual table entry points either to the implementation of a method declared in the class itself or to the implementation of an inherited method that has not been *overridden*.

The virtual table of a class C is constructed as follows. First, the virtual table of the parent class P is built[2], if it hasn't already been built[3]. Then, each virtual method of class C is assigned a unique offset into the virtual table. Each virtual method of C that overrides a virtual method of P or any ancestor of P is assigned the same offset as the overridden method. All other virtual methods are assigned an increasing offset starting at the maximum offset of inherited methods plus one. The highest offset determines the size of the virtual table. The virtual table of C is filled as follows. First it is initialized with the content of the virtual table of its parent P. Then, for each method of C, a pointer to its implementation is written in the virtual table at the method's offset.

Figure 4.1 illustrates the final result. Class `Parent` declares two methods `a()` and `b()`, which are assigned offsets 1 and 2 respectively. Class `Child` declares method `a()` which overrides method `a()` of class `Parent`, and method `c()` which is assigned

[2] Unless C is `java.lang.Object`.

[3] The rules for dynamic loading and linking in Java are relatively complex. A class can be loaded, yet not necessarily linked. See [LY99] for details.

offset 3.

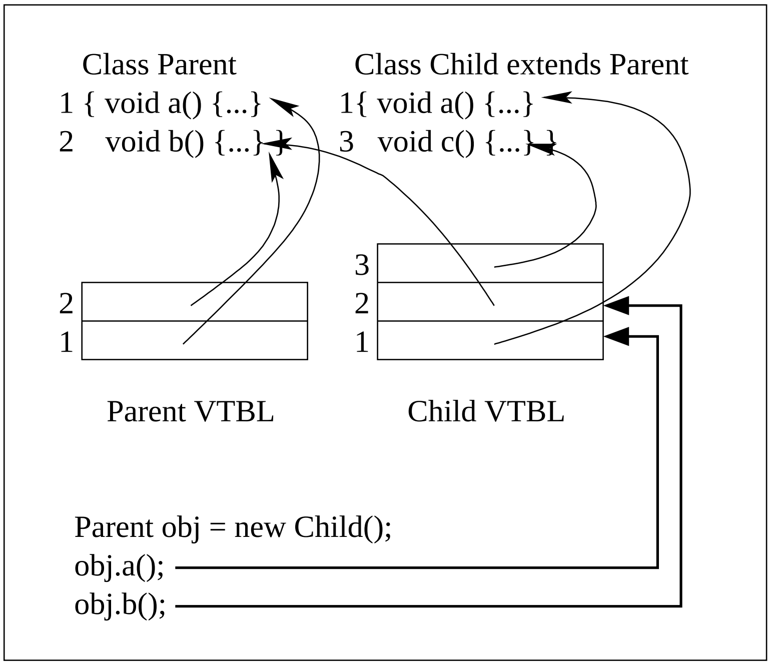


Figure 4.1: Single Inheritance Virtual Table (VTBL)

Virtual tables enable *virtual method invocation* in constant time, requiring a single indirection. Virtual method invocation proceeds as follows. The offset of the called method is used to retrieve the method implementation pointer from the virtual table. The pointer is then dereferenced and the target method executed. In Figure 4.1, the declaring type of variable `obj` is used to determine the method offset (`a() = 1, b() = 2`), but the actual method lookup is done using the virtual table of the instance type (which is `Child`). It is thus important that the offset remains the same for overridden methods (such as `a()`).

4.1.2 Virtual Tables for Interfaces

Java supports multiple inheritance of interfaces. The virtual table organization described in section 4.1.1 does not work for multiple-inheritance. The problem is that two distinct interfaces might assign conflicting offsets to method signatures, so when a class implements both interfaces, it can't determine a unique offset for its methods. This is illustrated in Figure 4.2. Method `a()` is assigned offset 1 in interface `Father` and method `b()` is assigned offset 1 in interface `Mother`. So, the system can't decide whether to put a pointer to the implementation of `a()` or `b()` at offset 1 in the virtual table.

Figure 4.2 also illustrates the usual solution to this problem. The idea is to reserve the normal *virtual table* for implementing virtual function calls only, and to build *interface virtual tables* for implementing interface method calls. A single class can have many interface virtual tables; one per directly or indirectly implemented interface. Interface method invocation proceeds as follows. First, the list of interface virtual tables is searched to find the appropriate interface virtual table, then an interface-specific method offset is used to lookup the implementation pointer in that table. For example, in Figure 4.2, to invoke method `b()` of interface `Mother` on an instance of class `Child`, the list of *IVTBL* pointers attached to the normal virtual table of class `Child` is searched. Then, the `Mother`-specific offset of method `b()`, which is 1, is used to lookup the implementation address of `b()`.

Many approaches are possible for representing the list of interface virtual tables of a class. One possibility is to use a plain linked list as was done in early versions of the Kaffe virtual machine. A superior approach is to build an array of pointers to interface virtual tables at a negative offset of the normal virtual table of a class as was done in Figure 4.2, then to use an efficient search technique (binary search / hashing) to find the appropriate interface virtual table.

Performance Issues

There are two performance issues related to using a list of interface virtual tables. The first and most important issue is that using this technique, the cost of interface method lookup is not *constant*. The cost of an interface method lookup grows with the number of directly and indirectly implemented interfaces of a class. The second problem is that the cost for searching for an appropriate IVTBL represents an overhead for interface method calls over normal virtual calls which do not need to perform any

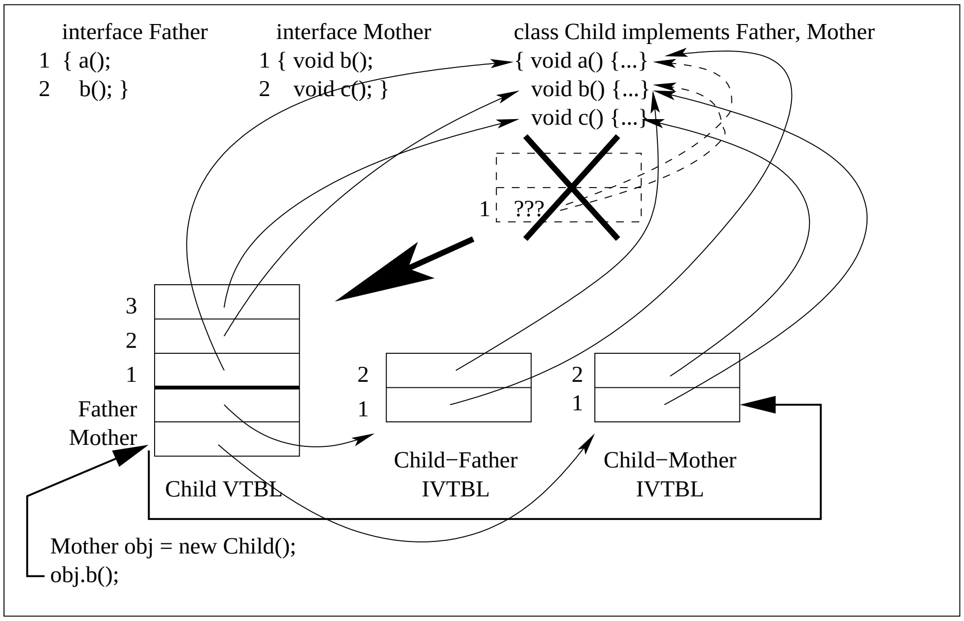


Figure 4.2: Interface Virtual Tables (IVTBL)

search to find a virtual table.

4.2 Sparse Interface Virtual Tables

In this section, we introduce a sparse interface virtual table layout that completely eliminates the usual overhead of interface method lookup over virtual method lookup.

The idea of maintaining multiple interface virtual tables in case of multiple inheritance is reminiscent of C++ implementations [ES90]. But, Java's multiple inheritance has a major semantic difference: it only applies to interfaces which may only declare method signatures without providing an implementation. Furthermore, if a Java class implements two distinct interfaces which declare the same method signature, this class satisfies both interfaces by providing a single implementation of this method. (Unlike Java, C++ allows the inheritance of distinct implementations of the same method signature).

We take advantage of this important difference to rethink the data structure needed for efficient interface method lookup. Our ideas are based on previous work on efficient method lookup in dynamically typed OO languages using of *selector-indexed dispatch tables* [Cox87, Dri93, VH94].

4.2.1 Basic Implementation

We assign a globally unique increasing index[4] to each method signature declared in an interface. A method signature declared in multiple interfaces is assigned a single index. When the virtual table of a class is created, we also create an *interface virtual table* that grows down from the normal virtual table. This interface virtual table has a size equal to the highest index of all methods declared in the direct and indirect super interfaces of the class. For every declared super interface method, the entry at its index is filled with the address of its implementation. The execution of *invokeinterface* can then proceed similarly and at the *exact same* cost as an *invokevirtual* instruction. The only difference is that interface method offsets are negative, while virtual method offsets are positive.

In the proposed organization, the interface virtual table is a *sparse* array of method pointers (unlike the normal virtual table which is *dense*). As more interfaces are loaded, with new interface method signatures (throughout program execution), the amount of free space in interface virtual tables grows. In fact, the total size of all interface tables is $O(i \times m)$, where i is the total number of interfaces, and m is the total number of distinct *interface method signatures*. Most of this space is empty, and could thus represent a significant loss of memory.

4.2.2 Filling the Holes

The traditional approach has been to use table compression techniques to reduce the amount of lost free space [Dri93, VH94]. These techniques work well within a statically compiled environment. However, they are poorly adapted to dynamic class loading environments like the Java virtual machine, as such techniques require

[4] In reality, we use a decreasing index, starting at at -1, to allow direct indexing in the *interface virtual table.*

dynamic reorganization of interface virtual tables when new classes and interfaces are loaded [Dri01].

Our approach differs. Instead of compressing interface virtual tables, we simply take advantage of our class loader memory manager to return the free space to the memory manager. The freed memory is then used to store all kinds of class loader related memory. In other words, we simply recycle the free space of sparse interface virtual tables within the class loader. The organization of sparse interface virtual tables is illustrated in Figure 4.3.

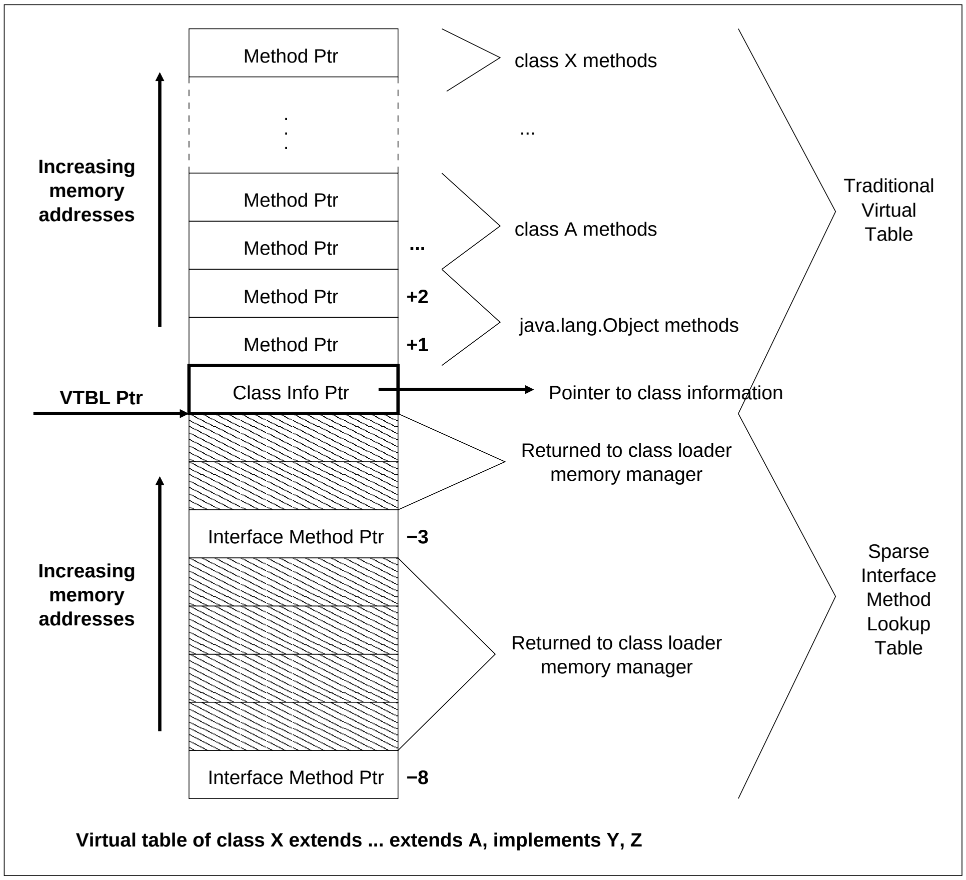

Method Ptr
class X methods
...
Increasing memory addresses
Method Ptr
class A methods
Method Ptr
...
Method Ptr
+2
Method Ptr
+1
java.lang.Object methods
Traditional Virtual Table
VTBL Ptr
Class Info Ptr
Pointer to class information
Returned to class loader memory manager
Interface Method Ptr
−3
Increasing memory addresses
Returned to class loader memory manager
Sparse Interface Method Lookup Table
Interface Method Ptr
−8
Virtual table of class X extends ... extends A, implements Y, Z

Figure 4.3: Sparse Interface Virtual Table Layout

Class Loader Memory Manager Internals

The internal design of a *class loader memory manager* is very simple. This memory manager keeps a constant size array of pointers to free memory blocks (called: *free array*). In *SableVM*, a free array has only 16 entries[5].

If it wasn't for sparse interface tables, a class loader memory manager would only need an `alloc()` function; it would not need a `free()` function[6]. Usually, a single pointer is used to manage free space as memory is allocated as a huge block using `malloc()` then redistributed incrementally.

But, because of sparse interface tables, a `free()` function is added which simply puts pointers to returned blocks into the *free array*. No memory aggregation takes place as holes in an interface virtual table cannot be neighboring other freed memory. If the free array overflows, the pointer to the smallest block is evicted[7]. `alloc()` always takes memory from the smallest, large enough block in the free array.

Our experimental results will show later that this rather simple strategy is quite effective.

4.2.3 Guarding Against Pathological Cases

As interface usage in most Java programs ranges from very low to moderate, we could argue that it is unlikely that the free space returned by interface virtual tables will grow faster than the rate at which it is recycled. However, in order to handle pathological cases, we also provide a very simple technique, which incurs no runtime overhead, to limit the maximal growth of interface virtual tables[8]. To limit this growth to N entries, we stop allocating new interface method indices as soon as index N is assigned to an interface method signature. Then, new interface method signatures are encoded using traditional techniques. The trick to make this possible is to use different *opcodes* to encode interface calls[9], based on whether the invoked

[5] This is a compile-time, easily modifiable constant.

[6] See Chapter 3 for explanations.

[7] The algorithm can be modified to use a linked list instead of an array to avoid memory loss.

[8] This is not currently implemented in *SableVM*.

[9] See Chapter 2 for instruction encoding techniques.

method signature has been assigned an index or not. The traditional technique used to handle overflow can safely ignore all interface methods which have already been assigned an index.

4.3 Experimental Results

We have experimented with our usual set of benchmarks[10] to measure the effectiveness of the proposed algorithm for recycling holes in the sparse virtual tables. Our results are shown in Table 4.1.

benchmark	**inter-**	**clas-**	**byte-**	**sparse ivt**	**holes**		**loss**
	faces	**ses**	**code**	(bytes)	(bytes)		(bytes)
compress	16	134	30,901	2,552	1,848	(72%)	0
db	17	130	34,255	2,984	2,212	(74%)	0
jack	17	178	62,791	4,436	3,608	(81%)	0
javac	21	269	101,027	3,696	2,844	(77%)	0
jess	20	270	52,985	30,340	28,504	(94%)	0
mpegaudio	24	167	76,509	5,296	4,504	(85%)	0
mtrt	18	155	42,037	3,400	2,520	(74%)	0
raytrace	18	154	41,863	3,400	2,520	(74%)	0
soot	190	794	223,557	211,912	193,328	(91%)	0
sablecc	24	374	93,335	129,340	112,840	(87%)	0

Table 4.1: Sparse Interface Virtual Tables

Columns of Table 4.1 contain respectively: (a) the name of the executed benchmark, (b) the number of loaded interfaces, (c) the number of loaded classes, (d) the total number of prepared bytecodes, (e) the total size of all sparse interface virtual tables in bytes, (f) the total size of free memory in sparse tables in bytes and as a percentage value of the total size of sparse interface tables, and finally (g) the total number of unrecycled memory bytes in sparse table holes.

4.3.1 Discussion

A few noteworthy results are:

[10] See Chapter 9 for details.

- Not a single byte of sparse table holes was lost.
- Sparse tables contain much free space: between 72% to 91% *on average*, in tested benchmarks.

Our most *interface intensive* benchmark is *Soot* which loaded 190 interfaces (this is much higher than the average number of interfaces loaded by usual Java applications). Its total sparse interface storage space is around 207Kb, and is smaller than the size of *prepared bytecodes*. The size of prepared bytecodes is shown as an indicator of the total requirement in class loader related memory. In a virtual machine, class loader memory also includes threaded-code or just-in-time compiled code, as well as various data structures to store information about classes, interfaces, arrays, methods and fields. For example, Table 2.2 (of Chapter 2) shows that the memory requirement for storing the inlined code sequences of the *Soot* benchmark is 574K, more than 3 times the total size of interface table holes. So, it seems this application could afford loading yet more interfaces and classes, and keep filling the holes without difficulty.

Of course, there could be some pathological cases, but they are unlikely to happen in human written code. This is because Java interfaces are of limited use; they do not provide an implementation for the method they declare. The use of interfaces is thus usually limited to those cases when single inheritance does not fulfill the designer's needs. This is the case with libraries like the Java Collection Framework (`java.util.*`).

One should also take into consideration that the size of a sparse interface table of a class is determined by the highest method index of an interface implemented by that class. Assuming a total of 1000 distinct interface method signatures were declared in all interfaces of an application, then the virtual machine is likely to fill all holes, as long as this class (or the next loaded class) requires more than 4Kb[11] of memory for storing compiled code and other information. As virtual table preparation happens early in the class linking process, before lazy method preparation, there is ample opportunity to recycle the memory of holes. We estimate that as long as no

[11] A little more than `sizeof(void *)` × 1000.

more than a few thousands *distinct* interface method signatures are declared in an application, recycling of hole memory won't be a problem.

4.4 Related Work

As we have said earlier, in [Dri01], K. Driesen surveys most of the implementation techniques that have been proposed and used for efficiently implementing polymorphic calls in various object-oriented programming languages. In this section we will discuss some of the closest work to our sparse interface virtual tables.

4.4.1 Selector Table Indexing

This technique is the simplest way of implementing dynamic method lookup, and is the basis of our sparse interface tables. It consists of constructing a two-dimensional table indexed by class and method signature. Both classes and method signatures are represented by unique, consecutive class or method signature codes. Unfortunately, the resulting dispatch tables is very large ($O(class \times signatures)$), and very sparse.

Our sparse interface tables breaks this table into disjoint rows. Each row is as short as its highest non-null entry. Yet, we measured between 72% to 94% free space.

Because of its enormous cost in memory, this technique is not used in real systems.

4.4.2 Row Displacement Compression

Row displacement compression is a technique used to minimize the loss of free space in *selector table indexing* two-dimensional dispatch tables. The idea, originally developed for compressing parsing tables in table-driven parsers [ASU86], is to break the two dimensional array into rows, then to fit rows into a one-dimensional array, so that non-empty entries overlap with empty ones.

In fact, slicing the two dimensional array can be done either on a class basis or on method signature basis. Chapter 4 of [Dri01] discusses why method-signature-based slicing yields significantly better compression than class-based slicing.

The biggest problem of this technique, in the context of Java, is that it is poorly adapted to dynamic loading environments. Implementing method-signature-based slicing, to obtain better compression, is particularly challenging in the presence of dynamic loading. It can require a complete reorganization of the compressed one-dimensional table when a new interface is loaded that causes a conflict (e.g. two non-empty entries overlap).

Our approach sidesteps compression by simply recycling memory for storing other things, and our experimental measurements show that there is an ample amount of other data to store in the holes of sparse interface tables. Yet, our approach only works well because it is limited to encoding *interface method* dispatch tables (thus a limited number of interfaces and method signatures). On the other hand, *row displacement compression* is much more appropriate for statically compiled, dynamically typed OO languages.

4.4.3 Interface Method Table Hashing

In [ACFG01] B. Alpern *et al.* propose an efficient interface method invocation for Java. Their idea is to associate a fixed-size *interface method table* with each class, then to use hashing to associate method signatures with interface method table entries. Hashing collisions are handled using custom-generated conflict resolution stubs.

Their measurements show that this technique cause little overhead for interface method calls over normal virtual method calls. Yet, there is some overhead. Our sparse tables completely eliminate this overhead, and are simpler to implement. Furthermore, our technique does not necessitate dynamically generating machine language encoded stubs.

Yet, we believe this technique would be best used in conjunction with our sparse interface tables, to handle overflow in pathological cases (when a very high number of distinct interface method signatures are declared).

4.5 Conclusions

In this chapter we introduced a *sparse interface virtual table* design that completely eliminates the overhead of interface method invocation over normal virtual method invocation. We proposed a simple algorithm that takes advantage of a partition-specific memory manager to recycle holes in the sparse tables.

Our experimental results show that this simple technique is highly effective. In all measured benchmarks, including in the interface-intensive Soot application, no memory loss resulted from using sparse tables.

Chapter 5
Bidirectional Object Layout

The Java heap is by definition a garbage-collected area. A Java programmer has no control on the deallocation of an object. Garbage collectors can be divided into two major classes: tracing and non-tracing collectors. Non-tracing collectors (mainly *reference counting*) cannot reclaim cyclic data structures, are a poor fit for concurrent programming models, and have a high reference count maintenance overhead. For this reason, Java virtual machine designers usually opt for a tracing collector.

There exist many tracing collectors [JL96]. The simplest models are mark-and-sweep, copying and mark-compact. The common point to all tracing collectors (including advanced generational, conservative and incremental techniques) is that they must trace a subset of the heap, starting from a root set, looking for *reachable* objects. Tracing is often one of the most expensive steps of garbage collection [JL96]. For every root, the *garbage collector* (gc) looks up the type of the object to find the offset of its reference fields, then it recursively visits the objects referenced by these fields.

In this chapter we introduce a new bidirectional object layout that groups all reference fields to allow simple and efficient gc tracing.

This chapter is structured as follows. In Section 5.1, we discuss the traditional layout for object instances. In Section 5.2, we introduce our bidirectional object layout. In Section 5.3, we present our experimental results. In Section 5.4 we discuss related work. Finally, in Section 5.5, we present our conclusions.

5.1 Traditional Layout

To provide efficient field access, it is desirable to place fields at a constant offset from the object header, regardless of inheritance. This is easily achieved in Java as instance fields can only be declared in classes (not in interfaces), and classes are restricted to single inheritance. Fields are usually laid out consecutively after the object header, starting with super class fields then subclass fields, as shown in Figure 5.1. When tracing such an object, the garbage collector must access the object's class information to discover the offset of its reference fields, then access the superclass information to obtain the offset of its reference fields, and so on. As this process must be repeated for each traced object, it is quite expensive.

There are two improvements that are usually applied to this naive representation. First, reference fields can be grouped together in the layout of each inherited class. Secondly, each class can store an array of offsets and counts of reference fields for itself and all its super classes. This is shown in Figure 5.2. The number of memory accesses needed to trace an object, in this case, is n (the number of references) + 3 (virtual table pointer, ref offsets pointer, array size) + 2 * array size (each array element has two values: base offset and reference number). Two nested loops (and loop variables) are required: one to traverse the array, and one for each array element (accessing the related number of references).

5.2 Bidirectional Object Layout

We propose a new object layout that further reduces the number of memory accesses required to trace an object. Our solution is to group all reference fields consecutively. To maintain the constant offset property, we simply grow objects in both directions, placing non-reference fields after the object header, and reference fields in front of it. Figure 5.3 illustrates the layout of object instances. In the object instance layout, the instance starting point is *possibly* a reference field. The instance grows both ways from the object header, which is located in the middle of the instance. References are placed before the header, and other fields are placed after it. Figure 5.4 illustrates the

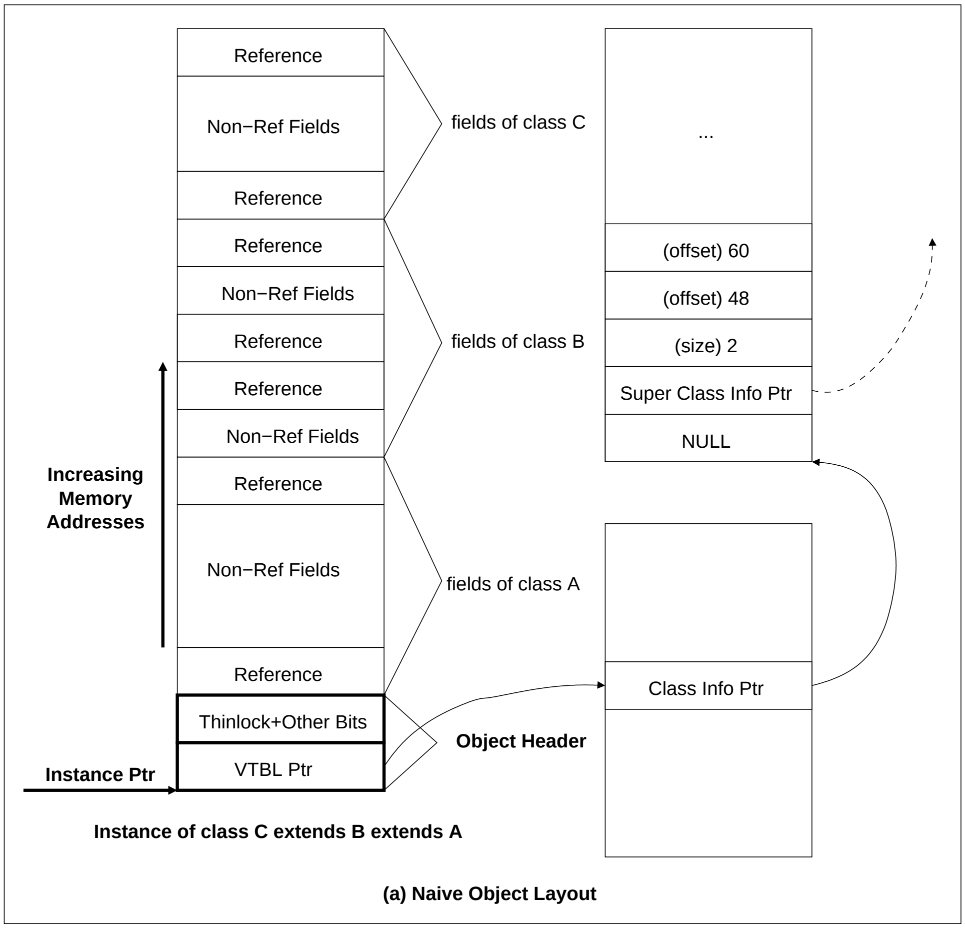


Figure 5.1: Naive Object Layout

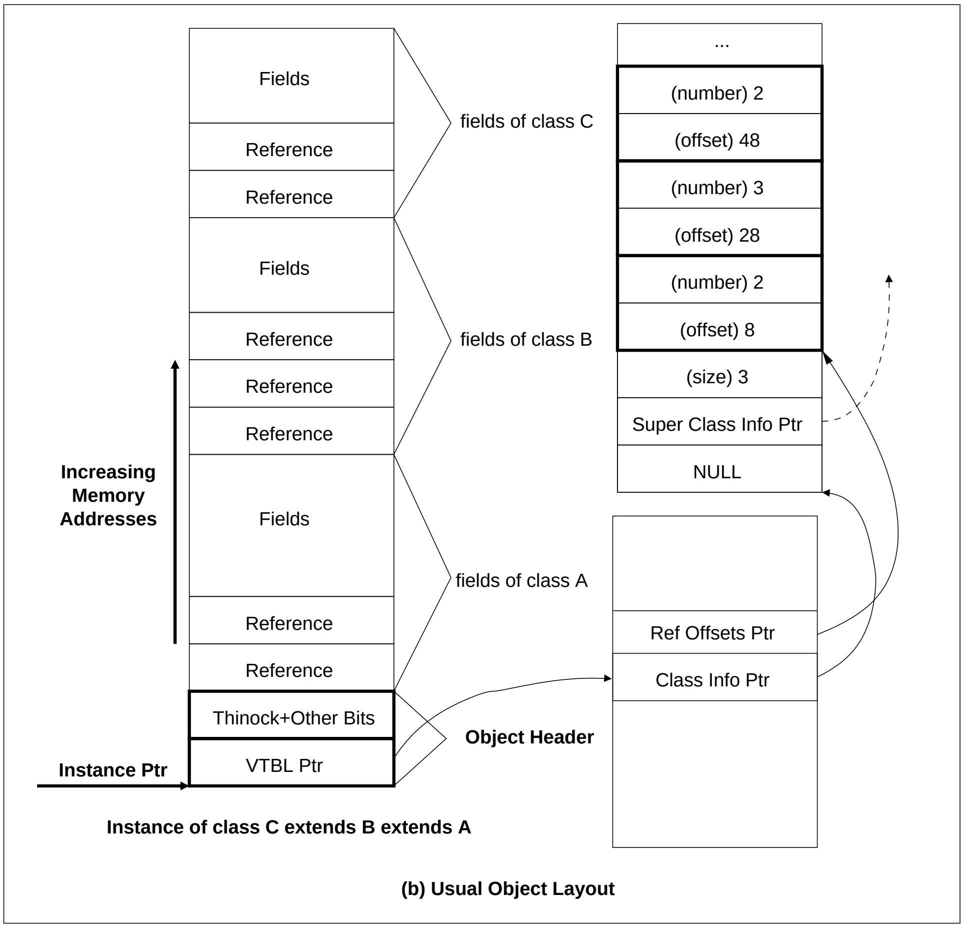


Figure 5.2: Traditional Object Layout

layout of array instances. Array elements are placed in front or after the array instance header, depending on whether the element type is a reference or a non-reference type, respectively.

The object header contains two words (three for arrays). The first is a *lock word* and the second is a virtual table pointer. We use a few low-order bits of the lock word to encode the following information:

- We set the last (lowest order) bit to one, to differentiate the lock word from the preceding reference fields (which are pointers to aligned objects, thus have their last bit set to zero).
- We use another bit to encode whether the instance is an object or an array.
- If it is an array, we use 4 bits to encode its element type (boolean, byte, short, char, int, long, float, double, or reference).
- If it is an object, we use a few bits to encode (1) the number of reference fields and (2) the number of non-reference field words of the object, (or special overflow values, if the object is too big).

We also use two words in the virtual table to encode the number of reference-field and non-reference-field *words* of the object if the object is too big to encode this information in the header.

5.2.1 Tracing Objects

At this point, we must distinguish the two ways in which an object instance can be reached by a tracing collector. The first way is through an object reference that points to the object header (which is in the middle of the object). The second way is through its starting point, in the *sweep* phase of a mark-and-sweep gc, or in the *tospace* scanning of a copying gc. In both cases, our bidirectional layout allows the implementation of simple and elegant tracing algorithms.

In the first case, the gc accesses the lock word to get the number of references n (one shift, one mask). If n is the overflow value (big object), then n is retrieved

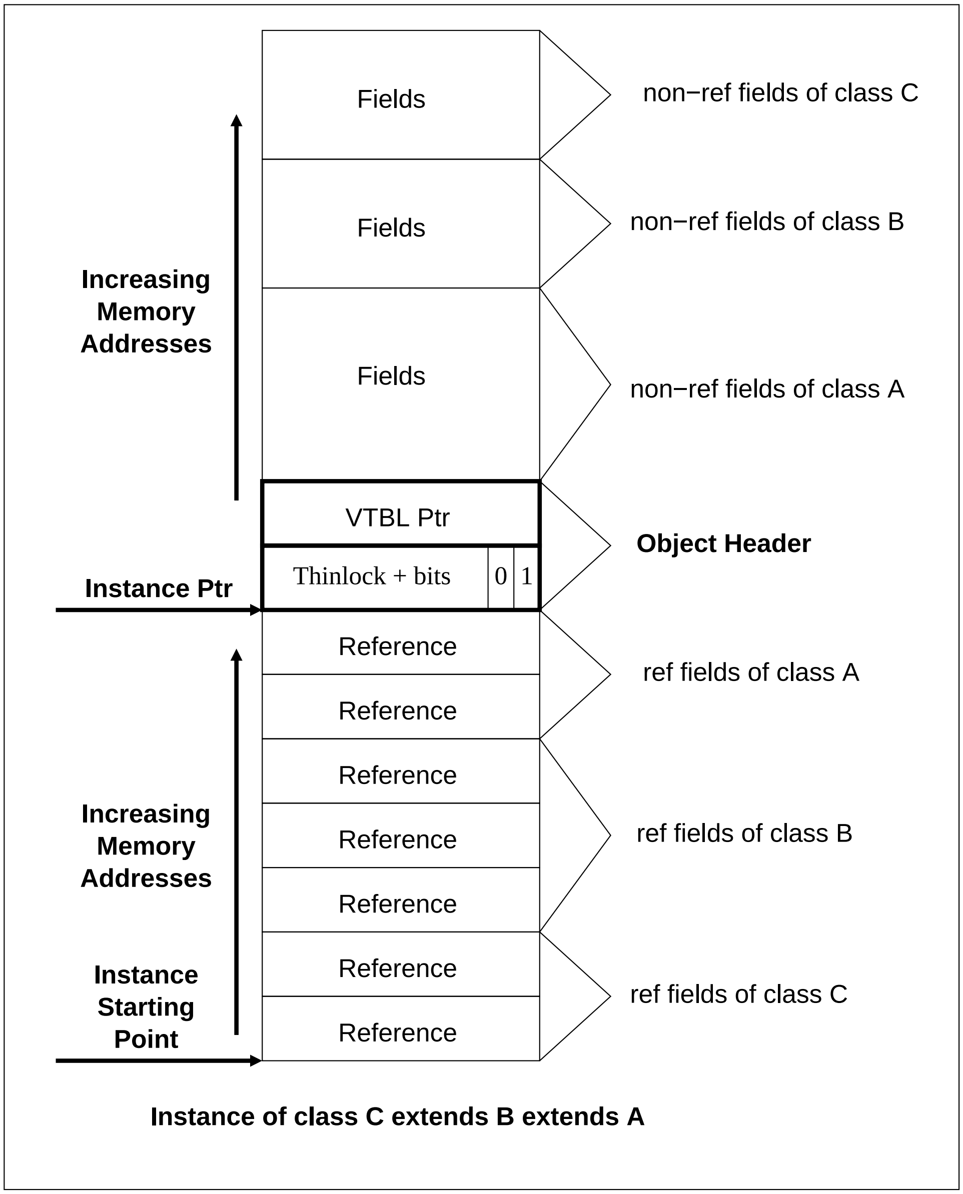


Figure 5.3: Bidirectional Object Instance Layout

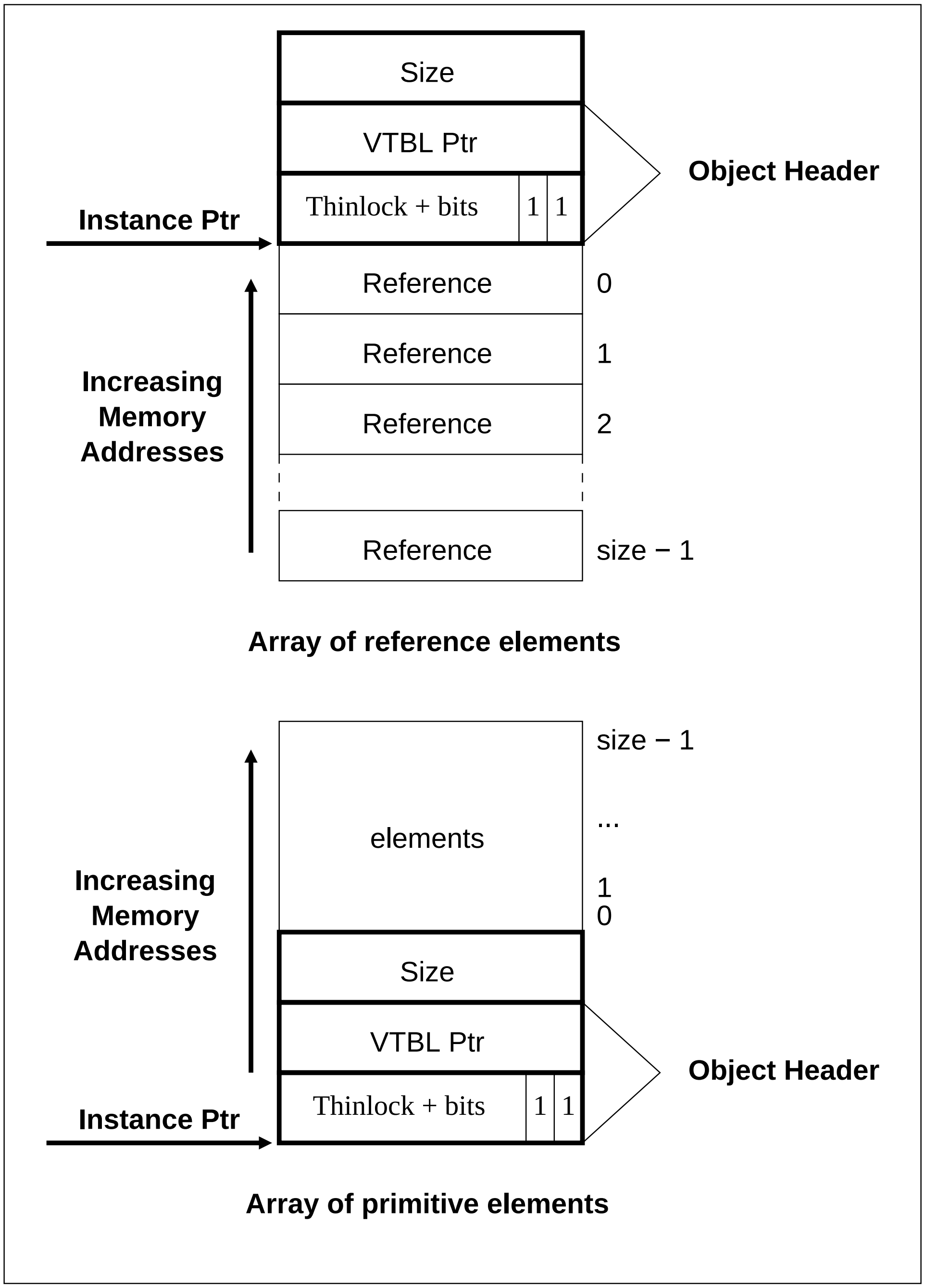


Figure 5.4: Bidirectional Array Instance Layout

from the virtual table. Finally, the gc simply traces n references in front of the object header.

In the second case, the object instance is reached from its starting point in memory, which might be either a reference field or the object header (if there are no reference fields in this instance). At this point, the gc must find out whether the initial word is a reference or a lock word. But, this is easy to find. The gc simply needs to check the state of the last bit of the word. If it is one, then the word is a lock word. If it is zero, then the word is a reference.

So, for example, a copying collector, while scanning the *tospace* needs only read words consecutively, checking the last bit. When set to zero, the discovered reference is traced, when set to 1, the number of non-reference field words (encoded in the lock word itself, or in the virtual table on overflow) is used to find the starting point of the next instance.

In summary, using our bidirectional layout, a gc only accesses the following memory locations while tracing: reference fields and lock word, for all instances (objects and arrays), and at most three additional accesses for objects with many fields (virtual table pointer and two words in the virtual table itself).

5.3 Experimental Results

We have experimented with our usual set of benchmarks[1] to measure the effectiveness of the proposed layout. Our results are shown in Table 5.1.

Columns of Table 5.1 contain respectively: (a) the name of the executed benchmark, (b) the total garbage collection time using the traditional object layout, (c) the total garbage collection time using the bidirectional object layout, (d) the total execution time using the traditional object layout, and (e) the total execution time using the bidirectional object layout.

[1] See Chapter 9 for details.

benchmark	gc (trad.)	gc (bidir.)		total (trad.)	total (bidir.)	
	(sec.)	(sec.)		(sec.)	(sec.)	
compress	0.238	0.238	(1.00)	129.58	131.64	(0.98)
db	0.472	0.479	(0.99)	89.91	87.64	(1.03)
jack	0.115	0.115	(1.00)	39.50	38.16	(1.04)
javac	2.845	2.875	(0.99)	89.45	89.37	(1.00)
jess	0.273	0.276	(0.99)	54.57	53.57	(1.02)
mpegaudio	0.000	0.000	(–)	135.86	136.97	(0.99)
mtrt	1.027	1.029	(1.00)	97.52	100.39	(0.97)
raytrace	0.679	0.683	(0.99)	113.38	113.55	(1.00)
soot	19.943	20.021	(1.00)	552.04	548.13	(1.01)
sablecc	0.172	0.173	(0.99)	26.12	26.09	(1.00)

Table 5.1: Garbage Collection Time

5.3.1 Discussion

Our results show that garbage collection time is not significantly affected by the object layout, for the tested benchmarks. Yet, object layout seems to have a bigger impact on the overall running time of applications. In the *db*, *jack*, and *mtrt* benchmarks we see a difference of 3% or a little more. In absolute value, the difference of execution time is much bigger than the difference in garbage collection time.

This is probably caused by the indirect effect of the reversed layout *reference arrays*. Effectively, reference arrays grow down from the object header. It is thus quite possible that the change of access order of array elements has some effect on the data cache.

5.4 Related Work

We now mention some previous related work. The idea of using a bidirectional object layout (without grouping references) has been investigated [Mye95,PW90] as a means to provide efficient access to instance data and dispatch information in languages supporting multiple inheritance (most specifically C++). In [Bar88], Bartlett proposed a garbage collector which required grouping pointers at the head of structures; this was not achieved using bidirectional structures, though.

5.5 Conclusions

In this chapter we introduced a bidirectional object layout which groups reference fields at the start of object instances. Such a layout simplifies garbage collection tracing.

Our experimental results show that using a bidirectional object layout causes no significant change to the execution time of garbage collection, yet it can sometime affect the overall benchmark execution time, probably due its impact on cache behavior.

This experiment illustrates the simplicity of implementing and testing various research data structures in *SableVM*.

Chapter 6
Space-Efficient Garbage Collection Maps

The Java virtual machine specification has a feature that makes computing type-precise root sets difficult for type-precise garbage collection; it allows local variables to have *subroutine*-call-sequence-specific types. Existing algorithms for computing garbage collection maps of local variables and *operand stack* locations are relatively complex, full-blown data-flow analyses, and the resulting maps are relatively spacious (10% to 20% of code size) [ADM98, SLC99]. In this chapter we introduce a simple algorithm that computes space-efficient stack and local variable maps for type-precise garbage collection.

This algorithm is best suited for simple or small Java virtual machines, as it reduces (a) the complexity of map computation, and (b) map storage space.

On the other hand, this algorithm does not perform a *live-variable* analysis for reducing unreclaimed garbage, and it can add a little runtime overhead to some method calls.

This chapter is organized as follows. In Section 6.1, we discuss the difficulty of computing type-precise garbage collection maps in Java. In Section 6.2, we introduce our algorithm. In Section 6.3, we present our experimental results. In Section 6.4 we discuss related work. Finally, in Section 6.5, we present our conclusions.

6.1 Type-Precise Garbage Collection Maps

The objective of garbage collection is to reclaim space consumed by objects that will not be used again. Precisely computing the set of objects that will be reused, at a certain point of program execution, is an *undecidable* problem, as program execution flow can depend on external data entry. Garbage collection algorithms compute, instead, the set of objects which are reachable from a root set, and recycle memory used by *unreachable* objects.

There are two main approaches to computing root sets. One approach is to compute a *precise* root set which includes all local and global *reference variables*. The second approach, called *conservative*, treats all local and global variables (regardless of their type) as potential roots.

Whether precise or conservative garbage collection is best suited for an environment or an application is still debated among researchers. The objective of the *SableVM* framework is to permit as much experimentation as possible. In that goal, it needs to support both types of garbage collection.

Supporting conservative garbage collection is simple. The only requirement is not to hide pointers to objects in memory using arithmetic operations (e.g. *xor*). A conservative garbage collector analyzes the content of an ambiguously typed variable to detect whether the stored value looks like a valid object reference. If it does, the garbage collector assumes it likely is a reference and acts accordingly. A conservative garbage collector can potentially retain more garbage, but practice has showed this not to be significant [BW88]. The simplicity of providing an ambiguous *root set* makes it possible to easily plug a general-purpose conservative collector into a system.

Supporting precise collection, on the other hand, proves to be more difficult, as a type-precise root set should be provided to the garbage collector. Usually, as is the case in the *SableVM* framework, precise garbage collection is only allowed to happen at predetermined execution points. At these *gc locations*, a map (usually encoded as a bit array) is provided to the garbage collector to distinguish between reference and non-reference variables.

6.1.1 The Gosling Property

As Java classes are dynamically loaded and can originate from an untrusted source, the Java virtual machine specification includes a bytecode verifier and states the rules for ensuring that no executed bytecode program cause memory corruption or other harm to the virtual machine.

Java bytecode is *stack-based*. For example, the `a = b + c;` Java statement is typically compiled to `iload_1 ; iload_2 ; iadd ; istore_0`.

The Java virtual machine imposes strict constraints on bytecode. In particular, the *Gosling property* states that, using a simple data-flow analysis, it should be verifiable that the stack size of a bytecode instruction is constant and that the type of each local variable and stack location is appropriate, regardless of the path taken to reach that instruction. A more precise definition of all virtual machine constraints is given in the Java virtual machine specification [LY99].

But, unfortunately, the Java designers allowed for one exception to the *Gosling property*.

6.1.2 A Notable Exception: Subroutines

Java bytecode includes two special instructions: `jsr` and `ret`, that were mainly introduced for implementing the `finally` construct of the Java programming language. The `jsr` instruction jumps to an address (specified as operand), and pushes a return address value *on the operand stack*. The `ret` instruction jumps to the address found *in the local variable* specified as operand. The code included within the target of a `jsr` instruction and its `ret` statement is usually called a *subroutine*[1].

To allow for the asymmetrical treatment of the *return address*, other bytecode instructions are allowed to swap and duplicate address values on the operand stack, and the `astore` instruction is allowed to pop a return address and store it into a local variable, so that it can later be used as operand to a `ret` instruction.

The rules for bytecode verification contain an explicit exception to the *Gosling*

[1] Unlike real subroutines, this code uses the same local variables and operand stack as the calling method. No *Java stack frame* is pushed or popped on execution of `jsr` and `ret` instructions.

property, allowing a *local variable* to hold a *call sequence* specific type within a subroutine, as long as this variable is neither read or written within the said subroutine.

This highly complicates the computation and storage of stack maps, as whether a local variable stores a *reference* value or not is dependent on the execution path to reach an instruction. To further complicate the problem, no traces are left on the operand stack or in local variables that clearly determine the `jsr` call sequence that lead to the execution of an instruction. So, a simple bit array encoding is not sufficient for local variable *gc maps*, within subroutines.

6.2 A Simple, yet Efficient Algorithm

The problem that we faced, when designing *SableVM* was that in order to precisely compute *gc maps*, we would have to encode a complete data flow analysis. As *gc maps* are even required for code loaded using the *bootstrap class loader*, this data flow analysis would have to be written in C, the implementation language of our virtual machine. Furthermore, simple bit array *gc maps* would not have been sufficient.

The complexity of existing algorithms for computing precise *gc maps* motivated our research for a simpler algorithm.

6.2.1 The Basic Idea

While bytecode verification is an important part of a commercial virtual machine, it is not necessarily as important within a research framework. Yet, existing algorithms to compute precise *gc maps* do most of the verification work.

An Important Assumption

In order to simplify the algorithm, we decided to make the reasonable assumption that the code for which we would compute gc maps would be *verifiable*. In other words, if verification was applied to this code, it would succeed. We say that it is a reasonable assumption, as code loaded by the *bootstrap* class loader is usually shipped with a virtual machine, and can thus be *pre-verified*. Other code, loaded through *system* and *user* class loaders written in Java, can be analyzed at link time by a verifier also written in Java[2].

Splitting Locals

Given the *verifiable code* assumption, we can devise a relatively simple algorithm which takes advantage of the type precision of most Java bytecode instructions to determine the type of local variables and stack locations.

In order to eliminate the *path-specific* property of local variable types, we identify all local variables which are used to store both *reference* and *non-reference* values. Then we split each of the identified local variables into two local variables: one which only stores reference values, and one which only stores non-reference values. The proposed splitting can cause an increase in the number of local variables of a method, but it greatly simplifies the computation of gc maps.

Single Locals Map

One of the most important consequences of the splitting of local variables is that, after splitting, a *single* local-variable gc map is required per Java method.

This can potentially save much memory, as otherwise, a local variable gc map would be required at every *gc check point*[3].

A side effect of using a single local map per method is that some *reference* local variables will need to be initialized to *null* on method entry. This is because the garbage collector would not know otherwise that the content of an uninitialized reference local variable is garbage.

[2] Linking of code loaded by the bootstrap class loader cannot *easily* depend on executing other Java code (chicken-egg problem).

[3] Some virtual machines do a lazy computation of gc maps in an attempt to reduce storage space [SLC99].

Grouping References

The storage of local variable gc maps can be further reduced by grouping all non-parameter local variables[4]. Doing so allows for encoding the map in two parts: a bit array for formal parameter local variables, and a single integer value indicating the number of non-parameter reference local variables.

Reducing the size of the bit array makes it more likely that other methods will use an identical bit array. This is useful to reduce memory consumption, when memoizing bit maps, as we explain later.

Also, having a single number for non-parameter locals simplifies the initialization of *reference* local variables. Parameter variables need not be initialized (they already hold a value provided by the caller), so a simple loop can be used to initialize the remaining non-parameter reference to *null* on method entry.

Operand Stack Maps

We also need to compute *operand stack* gc maps. One such map must be computed for every *gc check point*[5].

Luckily, many operand stack maps are small. For example, the operand stack is usually empty on branch instructions. Also, the bit array needs to be only as big as the highest index of a reference value on the stack.

Having a small (or an empty) bit array increases the opportunities for sharing stack maps across different locations.

Memoization

As the reader might have guessed by now, we maintain, in *SableVM*, a central repository of *bit array gc maps*. This repository is implemented as a *splay tree*[6], ordered by size and bit content.

[4] The formal parameters of a methods are mapped as the first local variables of a method frame.

[5] Backward branch instructions, method calls, and allocation instructions.

[6] A *splay tree* is a binary tree with *caching* property: the last accessed node is always made root of the tree. The amortized cost for n tree operations is $O(n \log n)$.

Example

Figure 6.1 illustrates the splitting and reordering of local variables. In this example, we assume there are no method parameters. First, the algorithm analyses locals usage. Then it splits locals which are used to store both *reference* and *non-reference* values. Then it reorders locals, assigning lowest numbers to *reference* variables. Finally, the bytecode is rewritten to use the newly assigned local variable numbers. Note how local 0 becomes local 1, and local 1 becomes locals 0 and 2.

Original Bytecode

```
ICONST_M1
ISTORE_0
ACONST_NULL
ASTORE_1
ILOAD_0
ISTORE_1
```

Locals Usage

Local 0: *non-ref* ⇒ no splitting.

Local 1: *ref* and *non-ref* ⇒ must be split.

Locals Splitting and Reordering

Local 0: *non-ref* = 1

Local 1: *ref* = 0, *non-ref* = 2

New Bytecode

```
ICONST_M1
ISTORE_1
ACONST_NULL
ASTORE_0
ILOAD_1
ISTORE_2
```

Figure 6.1: Locals Splitting and Reordering

6.2.2 What is a Subroutine?

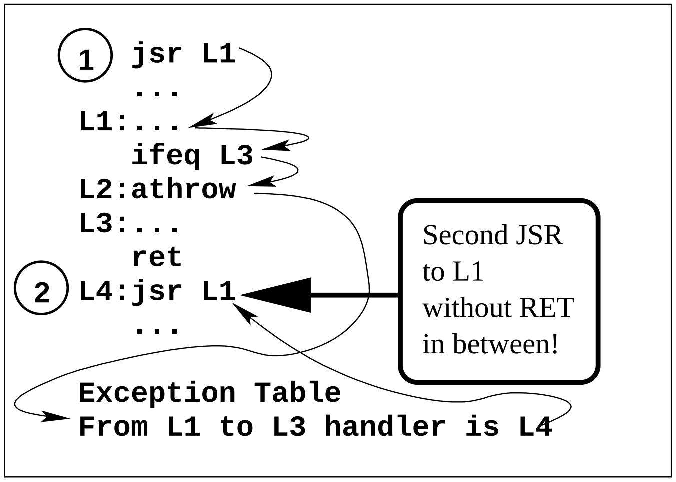
Now that we have exposed our main ideas, and before we give a detailed algorithm description, we need to discuss the *subroutine* problem.

The Java language specification motivates the existence of the `jsr` and `ret` instructions by explaining their use for implementing the `try {...} finally {...}` Java construct.

But, the verification rules governing the use of `jsr` and `ret` are stated in terms of bytecode instructions, not in term of Java programming language constructs.

So, in the context of bytecode (which is not necessarily generated by a Java compiler), the concept of *subroutine* becomes more difficult to grasp. In fact, the wording used in the Java language specification is *ambiguous*.

This is best explained using some examples. Figure 6.2 illustrates a case where, in the course of a data-flow analysis, two `jsr` instructions to the same *target* are seen, without a `ret` instruction between them (in the control flow). The problem is that the Java virtual machine states that *subroutines may not be recursively called*. Yet, this example passes verification successfully (using the *reference* virtual machine by Sun Microsystems).

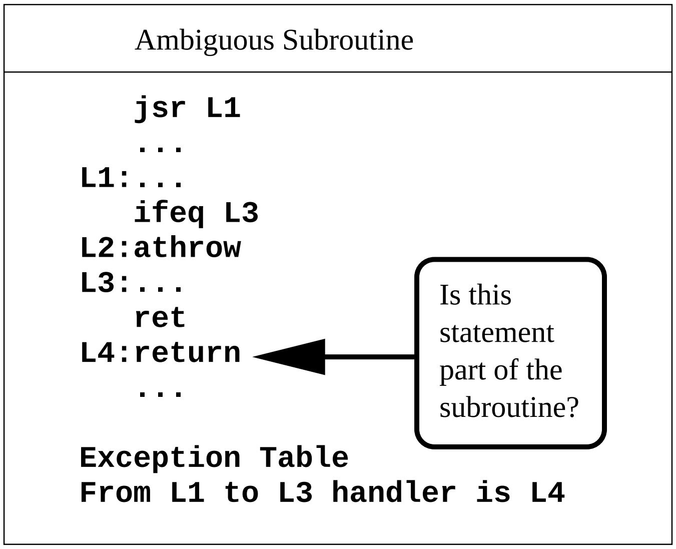


```
     jsr L1
     ...
  L1:...
     ifeq L3
  L2:athrow
  L3:...
     ret
  L4:jsr L1
     ...

  Exception Table
  From L1 to L3 handler is L4
```

Figure 6.2: Seemingly Recursive Subroutine

By analyzing this example further, we discover that a subroutine may be exited through *exceptional* control flow, such as the explicit `athrow` instruction, in our example. But, the problem remains: how can the data-flow analyzer determine whether a subroutine is exited or not, when taking an exceptional control flow?

In order to investigate this question, we have modified our example a little. We replaced the second `jsr` instruction by a `return` instruction (which ends a method). The result is illustrated in Figure 6.3.



Ambiguous Subroutine

```
   jsr L1
   ...
L1:...
   ifeq L3
L2:athrow
L3:...
   ret
L4:return
   ...

Exception Table
From L1 to L3 handler is L4
```

Figure 6.3: Ambiguous Subroutine

This segment of code is legal and it is again accepted by the *reference* virtual machine verifier. In this example, the `return` instruction could as well be within the subroutine (`return` may be called from within a subroutine), or it could be outside the subroutine. In either case, the verification *constraints* would be met. So, from a pure *bytecode* point of view, the boundaries of subroutines are ambiguous!

We should note that it is possible to write valid Java programs that generate code which is similar to our examples, using loops and nested `try-finally` and `try-catch` constructs.

So, to avoid pitfalls, we shall avoid using the ambiguous concept of *subroutines* when describing our algorithm[7].

[7] In order to prevent recursion, a data-flow analyzer can simply *invalidate* all copies of the target

6.2.3 Algorithm Description

We now give a precise description of our simple algorithm to compute gc maps. We remind the reader of the requirement that analyzed bytecode *must* be verifiable in order for our algorithm to work.

Data Structures

Our algorithm consists, in fact, of a simplified data-flow analysis, where each statement is only analyzed once. This data-flow analysis simulates the execution of bytecode instructions on an abstract method frame. The abstract method frame has an operand stack and local variables that can only hold integer values. Integer values stored in local variables and stack locations have the following meaning:

- -2: non-reference value
- -1: reference value
- 0 or more: start offset of a subroutine

One important difference between the *abstract interpretation* done by our data-flow analyzer and real execution is the treatment of the `jsr` instruction. The real instruction pushes the address of the return address on the stack. Our data-flow analyzer pushes the subroutine start offset on the stack, instead.

In addition, a global structure using a few bits of storage for each local variable is required. It records whether a local variable has been used to store *reference*, *one-word-non-reference*, and/or *two-words-non-reference*[8] values.

Another structure stores instruction specific data. For example, it records whether the stack operand to an `astore` instruction is a reference value or a return address.

Of course, as we said earlier, bit array gc maps are memoized using a global *splay tree*.

return address, stored in local variables and on the operand stack, when executing a `ret` instruction.

[8] The operand stack must also handle 64-bit `long` and `double` values.

Data Flow Analysis

A simple work-list algorithm is used. Initially, the first bytecode instruction of the analyzed method is pushed into the work-list. On each iteration, an instruction is retrieved from the work-list, its execution simulated within the abstract environment, then the instruction is marked as *done*. Then, all successor instructions (determined by regular and exceptional control flow) which are not *done* nor already in the work-list, are added to the work-list. The algorithm execution continues until the work-list is empty.

This algorithm only analyzes each statement once. This is sufficient, as the verifiability of analyzed bytecode ensures that the stack layout we compute the first time an instruction is seen is valid and would be the same *regardless* of execution path.

The abstract interpretation applied to the 200 bytecode instructions is fairly intuitive, except for the `jsr` and `ret` instructions which we will discuss in more details later. This interpretation consists of pushing or popping integer values on or off the stack, and updating appropriately the global local variable bits and the instruction specific data. Figure 6.4 shows the pseudo code of the abstract interpretation for a few bytecode instructions.

We will not describe in details how to analyze each of the 200 bytecode instructions. Instead, we invite the reader to look at our implementation which can be found in the file `src/libsablevm/prepare_code.c` of *SableVM* [Sabb].

The only special treatment is the handling of `jsr` and `ret` instructions. The idea is that in order to put the instruction following a `jsr` on the work-list, the operand stack layout (and size) of the related `ret` instruction must be known. As the data-flow analyzer might not yet know whether there is a related `ret` instruction or not, `jsr` simulation proceeds as follows:

1. The `jsr` target address (or more precisely: offset) is pushed onto the abstract operand stack.

2. If the target instruction has a related `ret` instruction on record, the operand stack of this related instruction is used in conjunction with the *current* local

```
#define NON_REF (-2)
#define REF (-1)

case ACONST_NULL or NEW
{
  /* push a reference on stack */
  stack[stack_size++] = REF;
}

case ISTORE_3 or FSTORE_3
{
  local_bits[3].used_as_nonref = true;
  stack_size--;
}

case LSTORE_3 or DSTORE_3
{
  local_bits[3].used_as_64bit = true;
  stack_size -= 2;
}

case ASTORE_3
{
  if (stack[--stack_size] >= 0)
  {
    /* there's a jsr offset on the stack */
    local_bits[3].used_as_nonref = true;
    instruction->operand_is_jsr_offset = true;

    /* save the jsr offset in the appropriate local */
    locals[3] = stack[stack_size];
  }
  else
  {
    /* there's a reference on the stack */
    local_bits[3].used_as_ref = true;
  }
}

case IADD, ISUB, FADD, FSUB, MONITORENTER,
     MONITOREXIT, or POP
{
  /* one 32-bit value popped */
  stack_size--;
}
```

Figure 6.4: Abstract Interpretation of Some Bytecode Instructions

variable layout as an environment[9] for adding the instruction following the `jsr` on the work-list.

3. If the target instruction doesn't have a related `ret` instruction on record, the instruction following the `jsr` is added to a pending-list in the target instruction record.

Simulation of the `ret` instruction proceeds as follows:

1. The address of the related `jsr` *target* is retrieved from the specified local variable.

2. This `ret` instruction is recorded as related to the `jsr` *target* instruction on record.

3. All instructions on the related `jsr` *target* pending-list are added to the work-list, with an appropriate environment.

Operand Stack Maps

Operand stack gc maps are computed at each *gc check point* as the related instruction is processed.

In order to simplify garbage collection, pointers to the *operand stack gc maps* are always stored in the *code array*[10] at `pc - 1`, where `pc` is the program counter value seen by the garbage collector.

Locals Splitting and Reordering

Once the data-flow analysis is finished, local variables are split according to their usage. Non-parameter local variables are reordered after splitting so that all reference locals are first and contiguous. Finally, bytecode instructions are updated appropriately, and local variable gc map information is stored in the method structures.

[9] This simulates the exception to the Gosling property.

[10] See Chapter 2.

6.3 Experimental Results

We performed two sets of experiments. The first set of experiments measured the total storage size for computed gc maps. The second set of experiments measured the increase in number of local variables caused by variable splitting. We have performed our experiments with our usual set of benchmarks[11].

6.3.1 Storage Size

The storage size of a single garbage collection map is composed of *splay tree* related fields and a bit array. There are 5 splay tree related fields per map: parent, left, and right pointers, bit array length, and bit array pointer. This takes a total of 20 bytes of storage overhead per gc map on the Linux/x86 platform. Our measurements are shown in Table 6.1.

benchmark	**maps**	**maps size**	**check points**	**methods**	**total size** (bytes)
compress	27	644 bytes	5,016	411	22,352
db	28	668 bytes	5,433	461	24,244
jack	33	788 bytes	9,752	689	42,552
javac	71	1,700 bytes	15,449	1,238	68,448
jess	43	1,028 bytes	8,804	892	39,812
mpegaudio	37	884 bytes	8,679	581	37,924
mtrt	43	1,028 bytes	6,849	588	30,776
raytrace	43	1,028 bytes	6,819	583	30,636
soot	74	1,776 bytes	39,653	3,475	174,228
sablecc	43	1,028 bytes	15,545	1,701	70,012

Table 6.1: GC Maps Storage Size

Columns of Table 6.1 contain respectively: (a) the name of the executed benchmark, (b) the total number of garbage collection maps for prepared methods, (c) the total storage size (including overhead) for all garbage collection maps, (d) the total number of garbage collection check points in prepared methods, (f) the number of prepared methods, and (f) the total storage size related to gc maps.

[11] See Chapter 9 for details.

The total size includes the size of gc maps, the size of a per check-point 32-bit pointer, and the size of a per method 32-bit integer.

Discussion

We were very pleasantly surprised by the small number of distinct garbage collection maps. Even in the biggest benchmark, Soot, which contains near 40,000 check points, a total of only 74 distinct maps are computed. The total storage space for these maps is less than 2Kb, most of which is memoization data structure overhead.

The following factors helped in reducing the number of distinct garbage collection maps. First, we do not compute check point specific maps for *local variables*, only method-specific maps. Also, local variable bit arrays are limited to formal parameters of methods; all non-parameter locals are grouped and a single integer is needed to map them.

Even the *total storage size* numbers are quite low. Most of this space is used to store a pointer to the gc map at `pc - 1`. For our biggest benchmark, the total storage size is only 170Kb.

Given the small number of distinct gc maps, the total storage could be dramatically reduced by using, at check points, a single byte of storage instead of a full pointer, indexing into a global table of gc maps (allowing for up to 255 maps + one value for overflow). The per method integer could also be stored using fewer bytes. Yet, we do not think any of this necessary, given the small total size of storage and the complexity of storing single bytes in aligned code arrays.

6.3.2 Number of Local Variables

In our second set of experiments, we measured the increase in the number of local variables. Our first results were unexpected; they indicated a *reduction* in the number of local variables, for many benchmarks. Suspecting a discrepancy in our code, we verified our code, but it seemed sound. This prompted us to investigate the problem. We quickly discovered the cause of the reduction in number of local variables: our algorithm gets rid of *dead* local variables.

In other words, some of the Java compilers used to compile the bytecode of our benchmarks and class libraries do emit bytecode which does not use all the local variables indicated by the *max_locals* value of the *code attribute* of methods. We used the *Jikes* [Jik] compiler for compiling the class libraries and the *Soot* and *SableCC* benchmarks. We do not know which compiler was used to compile the SPECjvm98 benchmarks.

So, we decided to also measure the increase in the number of *live* local variables. Our results are shown in Table 6.2.

benchmark	**bytecode locals**	**increase**	**live locals**	**increase**
compress	1,092	-7 (-0.6%)	1,056	29 (2.7%)
db	1,260	-9 (-0.7%)	1,212	39 (3.2%)
jack	1,771	-6 (-0.3%)	1,703	62 (3.6%)
javac	4,040	44 (1.1%)	3,750	334 (8.9%)
jess	2,225	2 (0.1%)	2,063	164 (7.9%)
mpegaudio	1,663	-11 (-0.7%)	1,606	46 (2.8%)
mtrt	1,648	-12 (-0.7%)	1,593	43 (2.7%)
raytrace	1,636	-12 (-0.7%)	1,582	42 (2.7%)
soot	8,517	60 (0.7%)	7,663	914 (11.9%)
sablecc	3,711	18 (0.5%)	3,303	426 (12.9%)

Table 6.2: Local Variable Count

Columns of Table 6.2 contain respectively: (a) the name of the executed benchmark, (b) the total number of local variables of prepared methods (before splitting), (c) the increase in local variables after splitting, expressed in absolute value and percentage, (d) the total number of *live* local variables of prepared methods (before splitting), and (c) the increase in *live* local variables after splitting, expressed in absolute value and percentage.

Discussion

While we think that Java compilers should be fixed not to generate *dead* local variables, a virtual machine must nonetheless execute verifiable bytecode generated by any compiler.

We expected an increase in the number of local variables, due to splitting. The raw result values (including dead variables) did not indicate a significant increase or reduction in the total number of variables.

We think that measurements on *live* variables are better indicators of the side effects of our proposed algorithm, as most modern virtual machines include *just-in-time* compilers or adaptive optimizers that most likely ignore dead local variables.

So, when we measure the effect of splitting on *live* local variables, we notice a more significant increase in the number of local variables, as expected. In the bigger benchmarks, the increase reaches up to 13%.

The noticeable increase in number of local variables could be an important factor to consider before adopting our algorithm in a high-performance system, as more local variables could lead to higher register pressure. Furthermore, developers of high-performance systems likely have the resources to implement more complex gc map computation algorithms.

6.4 Related Work

There has been much research done on various technique for garbage collection. In [JL96], R. Jones and R. Lins review most of the literature on the subject. In this section, we simply review related work specifically on computing gc maps in the Java virtual machine.

In [ADM98], O. Agesen *et al.* introduced a data flow analysis over a reduced type lattice to compute stack maps and record local variable usage conflicts in subroutines. Their technique records precise conflict information:

- *ref-uninit*: The local variable holds a reference value in some call sequence, and is uninitialized on another.
- *ref-nonref*: The local variable holds a reference in some call sequence, and a non-reference value on another.
- *ref-nonref-uninit*: The local variable holds a reference in some call sequence, a non-reference value on another, and is uninitialized on another.

Using this information, their technique then splits only those local variables involved in ref-nonref and ref-nonref-uninit conflicts. Also, the bytecode is modified to add `null` initialization to local variables involved in ref-uninit and ref-nonref-uninit conflicts. In the paper, O. Agesen *et al.* do not explicitly address the details of handling `long` and `double` types.

This technique is more precise than the technique introduced in this chapter as it minimizes the number of split local variables, and only initializes a subset of non-parameter reference local variables. But, on the other hand, this technique is more complex to implement, as it requires a full data-flow analysis, and it potentially requires more storage space for stack maps. Unfortunately, the paper does not report the total storage size related to stack maps. They did say that they do not compress (or memoize) stack maps.

Another interesting result of this paper, is that adding liveness analysis has no significant impact on the size of reachable objects in heap for most benchmarks. The only exception to this was in a benchmark specifically constructed to challenge garbage collectors.

In [SLC99], Stichnoth *et al.* introduce a technique to support garbage collection at every instruction, instead of at specific garbage collection check points. Their technique requires a full data flow analysis. They use an original technique to deal with the `jsr/ret` problem. Instead of splitting variables and rewriting the bytecode, they use a gc-time recursive recovery technique to deduce the type (and liveness) of variables. Their rationale for supporting garbage collection *at every instruction* is that it reduces latency in multi-threaded applications (yet they have not reported any timings to support this conjecture), and a simpler design for the virtual machine. We dispute both of these reasons, in the context of their technique.

We think that the latency of reaching a gc check point is negligible, as long as a checkpoint is present in every loop iteration. In [ADM98], O. Agesen computed that there is a gc check point every 7.9 bytecodes on average.

We also think that the difficulty of computing the type of variables at garbage collection time outweighs, by far, the difficulty of specializing back-branches in code arrays. The constant-time (per branch instruction) *overhead* of specializing branches is paid only once, at method preparation time, while the overhead of type recovery (linear worst case, in the size of method) must be paid at every garbage collection.

This paper also proposes a rather complex encoding of gc maps to compress them. This compression uses Huffman encoding to store the delta of each instruction, and uses *sequential bit streams* to store gc maps, so that no bit is wasted. All this encoding increases the complexity of garbage collection, as bit streams must be decoded for each inspected method frame on the Java stack.

Interestingly, the paper reports absolute values for the size of compressed gc maps. Specifically, it reports a total size of 22,920 bytes for the *compress* benchmark (compared to 22,352 in *SableVM*) and 93,385 bytes for the *javac* benchmark (compared to 68,448 bytes in *SableVM*). It should be noted that *SableVM*'s gc maps are accessed in constant time using a pointer at `pc - 1` which accounts for most of the total storage space. The storage size for gc maps alone, in *SableVM*, is less than 2Kb for the *javac* benchmark. In contrast, using Stichnoth's technique, retrieving a gc map is a complex, non-constant-time operation.

6.5 Conclusions

In this chapter we have introduced a simple and effective technique to compute space-efficient gc maps. By assuming that it is fed *verifiable* bytecode, the algorithm performs a simple analysis of bytecode to compute stack maps and split local variables according to their usage. Storage space is reduced by reordering local variables in methods, and using a single local variable map per method.

This technique is best suited to simpler virtual machines, and might not be appropriate for more complex high-performance virtual machines, unless additional analysis is done to reduce initialization overhead and local variable splitting (using an analysis similar to [ADM98]).

Our experimental results show that the number of distinct gc maps computed by our algorithm is very low. In our biggest benchmark, which has near 40,000 check points and 3,475 methods, only 74 distinct bit maps were necessary.

Chapter 7
Spin-Lock-Free Thin Locks

The Java virtual machine, in collaboration with the standard libraries, provides a multi-threaded execution environment to Java programs. Synchronization between threads is provided through recursive mutual-exclusive locks (called *monitors* in the Java virtual machine specification [LY99]). The bytecode instruction set specifically includes the MONITORENTER and MONITOREXIT instructions which respectively acquire and release a lock. Also, methods can be declared *synchronized*, causing the virtual machine to automatically acquire a lock on method entry and release it on method exit. Locks are associated with object instances; more precisely, every object instance has its own lock which can be acquired and released by running threads.

Most of the Java standard library classes and methods are *thread-safe*. In other words, these classes and methods make an extensive use of synchronization to protect internal data in multi-threaded programs.

To fully implement the semantics of the recursive locks of Java, a naive implementation would include a POSIX mutex, a POSIX condition variable and an integer recursion count into every object instance. This would add at least three words to every object instance. To significantly reduce this overhead, early Java virtual machine implementations used a global hash table to store lazily-created locks. On every lock and unlock operation, the global hash table is accessed to retrieve the lock associated with the object instance under synchronization. To preserve the integrity of the hash table, a global lock must be acquired and released on every access. This global synchronization causes significant execution overhead.

To improve the efficiency of the lock and unlock operations, various approaches were developed, such as the use of thread-local lock caches to reduce the number of costly global synchronization operations. In 1998, a very elegant algorithm was introduced by D. Bacon (and then improved by T. Onodera), to eliminate the need for a global hash table. Its main idea is to add a bimodal lock word in every object instance. The two modes of a lock are: thin and fat. In the thin mode, no additional storage is required for the lock. In the fat mode, the lock word contains the address of a full lock structure (mutex, condition variable and recursion count)[1]. The state of the lock word is indicated by its most significant bit; when this bit is set, the lock word is in the fat mode.

In this chapter we introduce an improvement to Onodera's bimodal field locking algorithm [OK99], which is a modified version of Bacon's thin lock algorithm [BKMS98] without busy-wait transitions from thin to fat mode.

This chapter is structured as follows. In Section 7.1 we discuss the related work, namely Bacon and Onodera's algorithms. In Section 7.2, we introduce our improvements to eliminate Bacon's algorithm busy-wait without adding storage overhead to object instances as does Onodera's solution. Finally, in Section 7.3, we present our conclusions.

7.1 Thin Locks

7.1.1 Bacon Algorithm

Bacon's thin lock algorithm can be summarized as follows. Each object instance has a one word lock in its header[2]. To acquire the lock of an object, a thread uses the *compare-and-swap* atomic operation to compare the current lock value to zero, and replace it with its thread identifier. If the lock value isn't zero, this means that either

[1] More precisely, the lock word includes the index in some data structure of the full lock, as not enough bits are available in the lock word to store an address.

[2] This is a simplified explanation of the algorithm. In reality, only 24 bits of that word are used for locking on 32 bit systems. 8 bits remain free for other uses. Refer to [BKMS98] for details.

the lock is already inflated, in which case a normal locking procedure is applied, or the lock is thin and is already acquired by some thread. In the latter case, if the owning thread is the current one, a nesting count (in the lock word) is increased. If the owning thread is not the current one, then there is contention, and Bacon's version of the algorithm busy-waits, spinning until it acquires the lock. When the lock is finally acquired, it is inflated[3]. Unlocking non-inflated locks is simple. On each unlock operation, the nesting count is decreased. When it reaches 0, the lock word is replaced by zero, releasing the lock.

The advantages of this algorithm are that a single atomic operation is needed to acquire a thin lock in absence of contention, and more importantly, no atomic operation is required to unlock an object[4].

Performance Improvements

Due to the thread-safe nature of the Java libraries, even single-threaded Java applications may spend a significant portion of their execution time performing useless synchronization. In [BKMS98], Bacon measured that replacing a normal heavy-weight implementation of Java monitors by thin-locks yields a median speedup of 1.22 and a maximum speedup of 1.7 on a set of *real programs*, which is a significant performance improvement.

7.1.2 Onodera's Proposed Improvement

Onodera proposed a technique to eliminate the busy wait in case of contention on a thin lock, using a single additional bit in each object instance. The role of this *contention bit* is to indicate that some other thread is waiting to acquire the current thin lock. Onodera's algorithm differs from the Bacon's algorithm at two points. First, when a thread fails to acquire a thin lock (because of contention), it acquires a fat monitor for the object, sets the contention bit, checks that the thin lock was not

[3] The lock is inflated so that future contention on the same lock won't cause busy wait.

[4] Unlike Agesen's *meta-lock* algorithm [ADG$^+$99] which requires an atomic operation for unlocking objects.

released, then puts itself in a waiting state. Second, when a thin lock is released (e.g. lock word is replaced by zero), the releasing thread checks the contention bit. If it is set, it inflates the lock, and notifies all waiting threads[5].

An Expensive Bit

The overhead of Onodera's algorithm over Bacon's is the contention bit test on unlocking, a fairly simple operation, and the one bit per object instance. This bit has the following restriction: it must not reside within the lock word. This is a problem.

It is important to keep the per-object space overhead as low as possible, as Java programs tend to allocate many small objects. It is now common practice to use 2 word headers in object instances; one word for the virtual pointer, and the second for the lock and other information. The contention bit cannot reside in either of these two words (putting this bit in the virtual table pointer word would add execution overhead to method invocation, field access, and any other operations dereferencing this pointer). As objects need to be aligned on a word multiple for the atomic operation to work, this one bit overhead might well translate into a whole word overhead for small objects. Furthermore, it is likely that the placement of this bit will be highly type dependent, which complicates the unlocking test.

7.2 Eliminating Busy-Wait Without Inflating Objects

Our solution to the expensive bit problem is to put the *contention bit* in the thread structure, instead of in the object instance. This simple modification has the advantage of eliminating the per-object overhead while maintaining the key properties of the algorithm, namely, fast thin lock acquisition with a single atomic operation, fast thin lock unlocking without atomic operations, and no busy-wait in case of contention.

To achieve the desired result, we modify Onodera's algorithm as follows. In *SableVM*, each thread has a related data structure containing various information,

[5] This is a simplified description. Please refer to the original paper [OK99] for details.

like stack information and exception status. In this structure, we add (a) the contention bit, (b) a *contention lock*[6], and (c) a linked list of (waiting thread, object) tuples. Then we modify the lock and unlock operation as described in the following two subsections.

7.2.1 Modifications to the Lock Operation

The lock operation is only modified in the case of contention on a thin lock.

When a thread x_t fails to acquire a thin lock on object z_o due to contention (because thread y_t already owns the thin lock), then (1) thread x_t acquires the contention lock of the owning thread (y_t), and (2) sets the contention bit of thread y_t, then (3) checks that the lock of object z_o is still thin and owned by thread y_t. If the check fails, (4a) the contention bit is restored to its initial value, the contention lock is released and the lock operation is repeated. If the check succeeds, (4b) the tuple (x_t, z_o) is added to the linked list of thread y_t, then thread x_t is put in the waiting state, releasing the contention lock of thread y_t. Later, when thread x_t wakes up (because it was signalled), it repeats the lock operation[7].

7.2.2 Modifications to the Unlock Operation

The unlock operation is modified to check the contention bit of the currently executing thread. This check is only done when a lock is actually released (as locks are recursive), after releasing the lock.

When the lock of object b_o is released by thread y_t, and if the contention bit of thread y_t is set, then (1) thread y_t acquires its own contention lock, and (2) iterates over all the elements of its tuple linked list. For each tuple (x_t, z_o), if ($z_0 = b_o$), thread x_t is simply signalled. If ($z_o \neq b_o$), the lock of object z_o is inflated[8] (if it is

[6] The contention lock is a simple non-recursive mutex.

[7] After releasing the contention lock of thread y_t that was automatically re-acquired on wake-up due to POSIX thread semantics.

[8] Notice that thread y_t necessarily owns the lock of object z_o, as only one lock (the lock of b_o) has been released by thread y_t since it last cleared its contention bit and emptied its tuple list.

thin), then thread x_t is signalled. Finally, (3) thread y_t empties its tuple linked list, clears its contention bit, and releases its contention lock.

7.2.3 Explanations

Our technique is best explained using an example of two threads T1 and T2 executing the program segments in Figure 7.1. We assume that T1 has succeeded at acquiring the thin lock of both o1 and o2 (and still owns them) and that T2 tries to acquire the lock of o1. As it is already owned by T1, there is contention[9].

Thread 1 (T1)	Thread 2 (T2)

`...`
`synchronized (o1)`
`{`
`  synchronized (o2)`
`  {`
`     ...`
`     /* execution point */`
`     ...`
`  }`
`}`
`...`
	`...`
	`/* execution point */`
	`synchronized (o1)`
	`{`
	`}`
	`...`

Figure 7.1: Contention Example

The goal of our algorithm is to avoid busy-wait in such a situation. Our strategy is to try to put T2 to sleep while making sure it will be awaked by the thread owning o1 when either o1 is unlocked or it is inflated.

To avoid any possible deadlock on thread contention locks, our algorithm was designed so that a thread never acquires more than a single thread contention lock at any time.

Going to Sleep Safely

We want to put T2 to sleep on the contention lock of the current owning thread of o1, which is T1. This will be safe if we can guarantee that T1 will effectively awake T2

[9] To simplify the text, we will say *o1* instead of *the lock of o1*.

when releasing or inflating o1. To make sure this happens, T2 acquires the contention lock of T1 and sets its contention bit, then it verifies that T1 still owns o1. If T1 does effectively still own o1, then we are assured that it will see a raised contention bit when it later unlocks or inflates o1, thus T2 can safely go to sleep on the contention lock of T1. If T1 is not the owner anymore, T2 undoes all modifications and repeats the process with the new owning thread of o1.

One could argue that T2 could end up chasing other threads, but this will only happen if the scheduling priority of T2 was low enough to let the other threads have enough time to acquire o1, do their work and release o1 before T2 has time to set the contention bit and check the thin lock. If this is the case, then the priority of T2 is low enough as not to starve the system.

Awaking Other Threads

The second part of our algorithm consists of awaking other threads when a lock is released or inflated.

Our design goal is to keep the unlocking code as simple as possible, to minimize overhead in the frequently executed unlock operation.

When a thread T1 releases a lock o2, it checks its own thread contention bit. If it is unset, T1 resumes normal execution, as no other thread is sleeping on its contention lock. If it is set, T1 acquires its own contention lock, then (a) awakes all other threads waiting on o2 and (2) inflates all other locks under contention and currently owned by T1 (such as o1 in our example), and awakes all threads waiting on these locks. Finally, T1 resets its contention bit and releases the contention lock and resumes normal execution.

Why not awake a single thread waiting on o2, instead of all of them? Because, in order for our algorithm to work, the other waiting threads must sleep on the contention lock of the thread owning o2. As these threads are currently sleeping on the contention lock of T1, and T1 is not the owner of o2 anymore, they must be awaked so that they can go to sleep on the contention lock of the new owner of o2.

Why inflate other locks than o2? Because, if T1 did not inflate them, it would have to keep its contention bit raised, causing higher overhead to all unlock operations on T1 as long as contention exists on a thin lock owned by T1.

Note that T1 owns its contention lock at the time it resets its contention bit. This ensures that no other thread will modify it concurrently.

7.3 Conclusions

The technique introduced in this chapter aims to solve a specific problem in an otherwise elegant existing technique for efficiently locking and unlocking objects in a Java virtual machine. The technique might seem relatively simple, yet it has a significant impact on a virtual machine robustness in multi-threaded environments, by preventing spin-locking on contention. An earlier solution had been proposed, but it came at a high cost in object storage space, potentially adding a complete word to object instances. Our solution elegantly avoids object instance inflation by using thread-specific storage for contention-related data structures.

Chapter 8
Portability and Extensibility

In this chapter we discuss the various technical aspects of the *SableVM* framework. In particular, we discuss the issues related to portability and extensibility of *SableVM*.

8.1 Portability of SableVM

In order to write a highly portable virtual machine, we had to be very careful in our usage of the ISO C language. In particular, we avoided all language features with *implementation-defined*, *undefined* or *unspecified* behavior [SAI^{+}90]. But unfortunately, efficiently implementing some Java features do require using a few system dependent features.

We discuss how we isolated all system specific features, then discuss some required architecture-level features and state some limitations of the current implementation.

8.1.1 System-Specific Files

All system-specific code is isolated in three specific files of *SableVM*. These files are:

```
src/libsablevm/include/jni_system_specific.h
src/libsablevm/system.h
src/libsablevm/system.c
```

They contain the definitions for size-specific integer and floating-point types, as well as functions to retrieve and set platform-specific header bits (lock-word) of object instances. They also contain the only two inline assembly functions described later in this chapter. Porting *SableVM* to a new platform consists mainly of modifying these three files.

Porting to Alpha Platform

It only took Grzegorz B. Prokopski, a new *SableVM* user, less than 24 hours, and less than 50 lines of commented code to port the framework to the 64 bit Alpha architecture[1]. The *unified diff* is shown in Appendix B. This is quite impressive for an efficient virtual machine. In fact, most of the work, which consisted of defining the appropriate `typedef` declarations for Alpha-specific types, took only a few minutes. Most of the remaining time was spent in a discussion between Mr. Prokopski and the author, to explain the need for an assembly-written *compare-and-swap* atomic operation, and in waiting for answers on the `debian-alpha` mailing-list [Deb].

Other Ports

Based on feed-back from users, *SableVM* is also known to run on the FreeBSD/x86, Debian/ia64 (Intel's new Itanium processor), Debian/PowerPC, and Debian/ARM platforms. The PowerPC processor, in particular, has a different byte ordering than the Intel x86 processor. All of these ports were done by *SableVM* users. In all cases, the *diff* is short, and it took only a few hours to make the port.

8.1.2 Architecture-Level Features

The main challenge in porting *SableVM* is that it requires two architecture-specific instructions, on modern processors, which are *not* expressible in the C language.

[1] He did not implement the *iflush* assembly function required for the inline-threaded engine; but the direct-threaded and switch-threaded engines worked well within 24 hours of his first attempt at porting the system.

Compare-And-Swap

The first required architecture-specific instruction is a *compare-and-swap* operation, which is sometimes provided as-is by the processor, or must be constructed as a sequence of processor-specific machine instructions.

This operation is necessary for the operation of *thin locks*. This instruction is implemented in the `system.c` file, and must be adapted for every new platform.

IFlush

Another architecture-specific instruction is required for getting *inline-threading* to work on processors with distinct *instruction* and *data* caches. This *iflush* instruction, described in Section 2.2.3 is unnecessary otherwise (i.e. no inline-threading, or unified cache).

As usual, this function is also implemented in the `system.c` file.

8.1.3 Limitations of the Current Implementation

We should mention a limitation of our current implementation related to multiprocessor systems. Correctly and efficiently implementing the Java semantics on modern multi-processors systems requires the usage of architecture-specific cache-related instructions. This is because modern multi-processor systems implement various *weak memory models*. An investigation of this problem revealed that there are few similarities between the various architectures. For example, various architectures have different semantics for *memory barriers*. We thus decided to postpone the research on this issue to future work.

8.2 Extensibility

One of the important goals of our work was to develop an *easily modifiable* framework. Yet supporting various interpreter engines and implementing nearly identical features can lead to code growth and duplication.

To avoid this, many systems use C macros. Unfortunately, the C preprocessor has many limitations, such as macros cannot generate new macros. Also, while using complex macros is an effective technique to reduce code growth and duplication (good for maintenance), it has the marked disadvantage that reading complex macro code is difficult, and the syntax for multi-line macros is inelegant.

A more unfortunate consequence of using complex macros is the loss of clear debuggable code, which can be traced through using a debugger, querying for variable values, etc.

For this reason[2], we decided to use the *GNU m4* [M4] general purpose macro processor.

8.2.1 Abstraction Levels Using m4

We developed a set of useful *m4* macros. Using the macros, we only need to provide a single implementation for a bytecode. The macro processor does all the work of generating multiple versions of this code into separate files.

To compile *SableVM*, a Makefile first invokes the *m4* processor and then invokes the C compiler on the generated code.

Interestingly, we were able to define *m4* macros which look like legitimate C code to a C indenter program. So, by giving the extension `.m4.c` to macro files, we are able to fool text editors to think that the code is actually C code, and thus get C syntax coloring during development. Another advantage is that we can also apply the GNU `indent` program on this source code to get a *uniform* indentation style across the application.

For the sample bytecode instruction implementation shown in Figure 8.1, the *m4* processor automatically generates many implementations of this code, one of which is shown in Figure 8.2.

Notice how the generated source code is commented, and appropriate for reading and debugging.

[2] After suffering from the difficulty to debug complex C macros in an early implementation of *SableVM*...

```
/*
----------------------------------------------------------------------
ACONST_NULL
----------------------------------------------------------------------
*/

        m4svm_instruction_head (ACONST_NULL, SVM_INTRP_FLAG_INLINEABLE, 0);

        stack[stack_size++].reference = NULL;

        m4svm_instruction_tail ();
```

Figure 8.1: Source Code

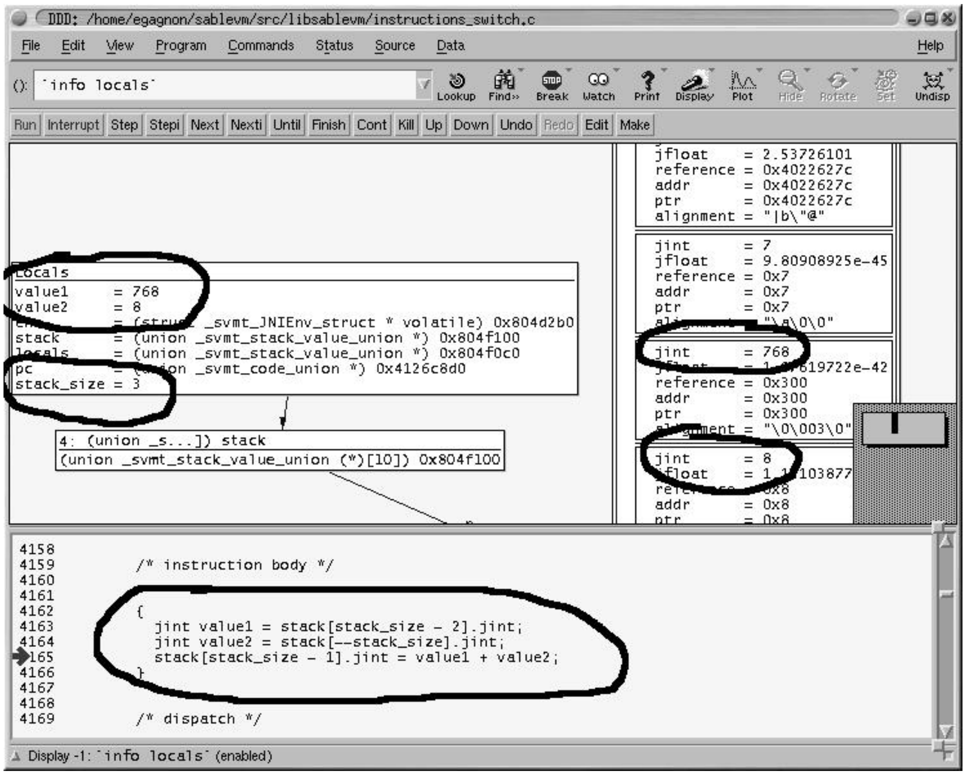
```
/*
----------------------------------------------------------------------
ACONST_NULL
----------------------------------------------------------------------
*/

    case SVM_INSTRUCTION_ACONST_NULL:
      {
        env->vm->instructions[instr].param_count = 0;

        /* implementation address */
        env->vm->instructions[instr].code.implementation = &&START_ACONST_NULL;
        env->vm->instructions[instr].inlined_code.implementation =
          &&INLINED_START_ACONST_NULL;

        /* code size */
        env->vm->instructions[instr].inlined_size =
          ((char *) &&END_ACONST_NULL) - ((char *) &&INLINED_START_ACONST_NULL);

        /* can the implementation be relocated? */
        env->vm->instructions[instr].flag = SVM_INTRP_FLAG_INLINEABLE;

        break;

      START_ACONST_NULL:
#ifndef NDEBUG

        if (env->vm->verbose_instructions)
          {
            _svmf_printf (env, stdout,
                          "[verbose instructions: executing @\%p ACONST_NULL]\n",
                          (void *) (pc - 1));
          }

#endif

      INLINED_START_ACONST_NULL:
        /* instruction body */

        stack[stack_size++].reference = NULL;


      END_ACONST_NULL:
        /* dispatch */
        goto *((pc++)->implementation);
      }
```

Figure 8.2: Generated Code

8.2.2 Debugging SableVM

One interesting aspect of the *SableVM* framework is that the execution of the virtual machine can be easily traced using a debugger. The usage of *m4* for the generation of *commented* source code does really help providing a very accessible, easy to learn system, as new users need not understand the *m4* code; they can simply look through the commented generated code, and trace it using a debugger.

Figure 8.3 shows a debugging session within the DDD debugger [DDD]. The `IADD` bytecode instruction body is being executed within the *switch-threaded* engine of *SableVM*. Notice how the debugger displays the *operand stack* content on the right part of the figure, and the value of C local variables `value1`, `value2`, and `stack_size` on the left part. In the bottom part, we see that line 4165 is about to be executed.



DDD: /home/egagnon/sablevm/src/libsablevm/instructions_switch.c

(): `info locals`

```
Locals
value1     = 768
value2     = 8
           = (struct _svmt_JNIEnv_struct * volatile) 0x804d2b0
stack      = (union _svmt_stack_value_union *) 0x804f100
locals     = (union _svmt_stack_value_union *) 0x804f0c0
pc         = (union _svmt_code_union *) 0x4126c8d0
stack_size = 3
```

```
4: (union _s...]) stack
(union _svmt_stack_value_union (*)[10]) 0x804f100
```

```
jfloat    = 2.53726101
reference = 0x4022627c
addr      = 0x4022627c
ptr       = 0x4022627c
alignment = "|b\"@"
```

```
jint      = 7
jfloat    = 9.80908925e-45
reference = 0x7
addr      = 0x7
ptr       = 0x7
alignment = "\a\0\0"
```

```
jint      = 768
jfloat    = 1.07619722e-42
reference = 0x300
addr      = 0x300
ptr       = 0x300
alignment = "\0\003\0"
```

```
jint      = 8
jfloat    = 1.1 103877
reference = 0x8
addr      = 0x8
ptr       = 0x8
```

```
4158
4159        /* instruction body */
4160
4161
4162        {
4163          jint value1 = stack[stack_size - 2].jint;
4164          jint value2 = stack[--stack_size].jint;
4165          stack[stack_size - 1].jint = value1 + value2;
4166        }
4167
4168
4169        /* dispatch */
```

Display -1: `info locals` (enabled)

Figure 8.3: Debugging Session

Such a precise and clear trace of the execution of a bytecode instruction is not available to developers working with compiler-based Java virtual machines.

8.3 Conclusions

We have designed *SableVM* to be portable. As the implementation of some features are intrinsically platform dependent, we have isolated all the platform-specific source code in well identified source files.

In order to ensure easy maintenance and extensibility, we have used the *GNU m4* macro processor to generate commented code, avoiding source code duplication. We have, in fact, developed a set of elegant *m4* macros which can be conveniently hidden in otherwise legitimate looking C code.

We do think that the architecture of *SableVM* achieves our goals of portability and extensibility.

Chapter 9
Overall Performance Measurements

In this chapter we present our overall performance measurements, comparing the running times of various benchmarks on *SableVM* and other virtual machines. The test platform, the virtual machines and the benchmarks discussed in this chapter were also used for performing experiments in preceding chapters.

This chapter is structured as follows. In Section 9.1, we describe the platform used in our tests. In Section 9.2, we discuss our choice of comparative Java virtual machines. In Section 9.3, we discuss our choice of benchmarks. In Section 9.4, we present our experimental results. Finally, in Section 9.5, we present our conclusions.

9.1 Test Platform

We have performed all our experiments on a single 1.5GHz Pentium 4 based system, with 1.5 Gb of RAM, 256 Kb of cache memory, and a 7,200 RPM hard disk, running Debian/Gnu Linux with kernel version 2.4.18. All daemon processes were turned off during the tests.

All execution time measurements are based on $(system + user)$ time returned by the GNU `time` command, and are the average execution time of 3 runs of each program.

9.2 Virtual Machines

In this chapter we compare the performance of *SableVM* to other virtual machines. We have chosen two sets of virtual machines to compare with: interpreters and compiler systems.

9.2.1 Interpreters

Kaffe Interpreter

We have chosen to compare the performance of *SableVM* with that of the interpreter of *Kaffe* virtual machine (version 1.0.7), as it is one of the most popular open-source virtual machines.

The *Kaffe* interpreter is a naive Java bytecode interpreter. The designers of Kaffe did not try to optimize its performance. They devoted most of their time building an efficient just-in-time compiler. The Kaffe virtual machine does not support precise garbage collection; instead, it relies on the Boehm-Weiser conservative collector for C.

JDK 1.4.0 Interpreter

We have also chosen to compare *SableVM* with a state-of-the-art interpreter. To do so, we selected the *HotSpot Client VM* interpreter included within the JDK 1.4.0 for Linux (build 1.4.0-b92).

This interpreter has to be relatively efficient, as it is used by the *mixed mode* high-performance HotSpot engine. It benefits from all the highly sophisticated HotSpot framework features, including efficient heap allocators and generational collection. The interpreter is known to be partly coded in assembly language[1].

Note: To select the interpreter engine, we used the `java -Xint` command.

[1] We cannot assert of this claim, as we have not signed a non-disclosure agreement to get access to the source code of the system.

9.2.2 Compiler Systems

Jikes RVM (Baseline, Semi-Space)

We selected the most basic configuration of the Jikes RVM (version 2.1.1 for Linux), so that we could compare the performance of *SableVM*'s inline-threaded engine to that of a simple just-in-time compiler, using a similar semi-space copying garbage collector.

Open Intel Platform

We also compares *SableVM* with Intel's ORP version 1.0.9, pre-packaged for Debian. We selected ORP as it is another open-source virtual machine using the GNU Classpath class library. ORP uses a JIT engine, and is written in C++.

Kaffe (JIT3)

We also compared *SableVM* with the most efficient Kaffe (version 1.0.7) just-in-time compiler engine, to be fair after comparing *SableVM* to its slow interpreter engine.

JDK 1.4.0 (Mixed-Mode)

Finally, we compared the performance of *SableVM* to that of the Client HotSpot VM (build 1.4.0-b92), in mixed mode execution. This virtual machine is a state-of-the-art system, aiming at achieving the highest performance in a client environment.

9.3 Benchmarks

We have selected the SPECjvm98 [SPE] benchmarks, as they are widely used for collecting experimental measurements in research papers on Java virtual machines. We should note that none of the results shown in this thesis represent official SPEC performance measurements, as we have not followed the official run rules of the SPEC committee. We have run unmodified SPECjvm98 programs, but we used custom wrapper scripts to collect the various measurements.

We have also chosen two benchmarks developed by the Sable Research Group of McGill University, Soot 1.2.3 [Soo] and SableCC 2.17.3 [Saba], for their highly object-oriented design, and their use of Java interfaces.

Soot is a bytecode analysis and optimization framework. In our test, we gave the *javac* SPECjvm98 benchmark classes as input to Soot[2].

SableCC is a compiler generator (or *compiler compiler*) that generates DFA-based lexers, LALR(1) table-based parsers, and a complete set of Java classes (source code) for building and traversing abstract syntax trees. In our tests, we gave SableCC the grammar of *Simple C*[3] as input[4].

9.4 Results

We now present our overall comparative performance measurements. For these tests we used a version of SableVM with an inline-threaded engine, signal-based null checks, bidirectional layout, and precise semi-space copying collector. All running times are expressed in seconds, and are the average CPU time (i.e. *user+system* time) of three runs of the benchmarks. For every virtual machine, other than *SableVM*, we also show the speedup achieved by *SableVM* over the measured virtual machine between parentheses.

Our first set of experiments compared *SableVM* to other interpreters, namely the Kaffe interpreter version 1.0.7 and the JDK 1.4.0 HotSpot Client VM interpreter. Results are shown in Table 9.1.

Our second set of experiments compared *SableVM* to compiler-based virtual machines, namely the Jikes RVM (baseline, semi-space), Intel's ORP virtual machine, Kaffe's JIT3 engine, and JDK 1.4.0 HotSpot Client VM (mixed-mode). Results are shown in Table 9.2.

[2] Command: `java soot.Main -d newClasses --app -W spec.benchmarks._213_javac.Main`

[3] This grammar can be found on the SableCC web site, along with other grammars.

[4] Command: `java org.sablecc.sablecc.SableCC simplec.sablecc`

		Kaffe		JDK	
benchmark	**SableVM**	**interpreter**		**interpreter**	
	(sec.)	(sec.)		(sec.)	
compress	131.64	1048.35	(7.96)	175.87	(1.34)
db	87.64	364.87	(4.16)	82.82	(0.95)
jack	38.16	307.70	(8.06)	30.46	(0.80)
javac	89.37	405.75	(4.54)	49.94	(0.56)
jess	53.57	297.94	(5.56)	39.25	(0.73)
mpegaudio	136.97	677.88	(4.95)	141.19	(1.03)
mtrt	100.39	351.08	(3.50)	46.67	(0.46)
raytrace	113.55	382.97	(3.37)	45.28	(0.40)
soot	548.13	failed	(–)	390.68	(0.71)
sablecc	26.09	failed	(–)	26.64	(1.02)

Table 9.1: Comparative Performance: *SableVM* vs. Interpreters

		Jikes				**Kaffe**		**JDK**	
benchmark	**SVM**	**RVM**		**ORP**		**JIT3**		**1.4.0**	
	(sec.)	(sec.)		(sec.)		(sec.)		(sec.)	
compress	131.64	43.77	(0.33)	15.22	(0.12)	18.24	(0.14)	19.47	(0.15)
db	87.64	48.88	(0.56)	27.88	(0.32)	41.90	(0.48)	28.86	(0.33)
jack	38.16	24.08	(0.63)	7.01	(0.18)	50.92	(1.33)	6.78	(0.18)
javac	89.37	36.00	(0.40)	failed	(–)	46.67	(0.52)	15.25	(0.17)
jess	53.57	29.67	(0.55)	6.74	(0.13)	38.56	(0.72)	6.61	(0.12)
mpegaudio	136.97	34.94	(0.26)	6.84	(0.05)	32.82	(0.24)	10.60	(0.08)
mtrt	100.39	20.00	(0.20)	6.73	(0.07)	32.93	(0.33)	5.33	(0.05)
raytrace	113.55	18.90	(0.17)	5.85	(0.05)	31.69	(0.28)	4.51	(0.04)
soot	548.13	483.02	(0.88)	failed	(–)	failed	(–)	68.97	(0.13)
sablecc	26.09	19.78	(0.75)	failed	(–)	failed	(–)	6.58	(0.25)

Table 9.2: Comparative Performance: *SableVM* vs. Compilers

9.4.1 Discussion

First, we should stress that comparing different virtual machines on total execution time is not always very accurate, as the running time of Java applications is often dependent on the efficiency of standard class library code. It would be nearly impossible to abstract library execution running time out of the total execution time. So, one must be very careful before drawing conclusions from execution time measurements.

Results in Table 9.1 show that *SableVM* is significantly faster than a naively implemented bytecode interpreter. It achieves a speedup ranging from 3.37 to 8.06 over Kaffe's interpreter engine.

Also, results in Table 9.1 show that *SableVM* achieves comparable performance with a state-of-the-art Java interpreter (JDK 1.4.0), by getting a speedup ranging from 0.40 to 1.34. This is quite an achievement for a relatively simple and highly portable virtual machine, with a very basic non-generational copying garbage collector, which does not do any fancy optimizations for exception handling, multi-threaded heap allocation, and other features.

Of course results in Table 9.2 remind us that *SableVM*'s engine is clearly an interpreter, not a compiler. For the *raytrace* benchmark, *SableVM* is more than 25 times slower than the JDK HotSpot VM Client (mixed-mode). Yet, when compared with a relatively naive just-in-time compiler engine, such as the Jikes RVM's baseline compiler, *SableVM* achieves a decent comparative performance. On the *Soot* benchmarks, it achieves 88% of the performance of Jikes RVM (baseline), and on *SableCC*, it achieves 75% of Jikes RVM (baseline).

Also, *SableVM* performs relatively well (considering it is an interpreter) against the Kaffe JIT3 engine. In fact, it outperforms it by 33% on the *jack* benchmark. We have not identified, at this point, the reason for the bad performance of Kaffe on this specific benchmark. We do not think it is normal for an interpreter to outperform a compiler-based virtual machine unless compile-time overhead (and code storage space) justifies it, which is not the case here (as indicated by the running times of other compiler-based virtual machines).

For many benchmarks, *SableVM* achieves more than a third of the performance of a naive JIT (Jikes RVM). It also achieves 33% or more of the performance Kaffe's most efficient JIT for a majority of benchmarks. Given the huge difference in the complexity of compiler-based systems and a highly-portable interpreter, we think that the performance of *SableVM* offers an attractive (portability and simplicity)-performance tradeoff for doing research within the Java virtual machine.

9.5 Conclusions

Our experimental results show that *SableVM* largely outperforms a naive Java bytecode interpreters, and offers comparative performance to a state-of-the-art interpreter used within a modern mixed-mode adaptive system.

The performance of *SableVM* is largely inferior to that of modern adaptive systems, but it is not too far from the performance of a naive just-in-time compiler on some large benchmarks. Overall, *SableVM* offers, in our view, a very attractive (portability and simplicity)-performance tradeoff.

Chapter 10
Future Work and Conclusions

In this last chapter we discuss future work on *SableVM* and present our overall conclusions. This chapter is structured as follows. In Section 10.1, we discuss various future research avenues, and in Section 10.2, we present the overall conclusions of this thesis.

10.1 Future Work

10.1.1 SableVM in the Field

The first part of our future work has already started. It consists of releasing *SableVM* publicly, gathering feedback from the research community, and establishing new research and development collaborations.

We hope to further develop the already started collaboration between the *SableVM* and the *GNU Classpath* projects for building stable and robust Java virtual machine and libraries.

We also seek to attract other research projects to merge their work within the *SableVM* framework, when possible, to reduce duplication of effort. We think that our work on building a robust Java virtual machine research infrastructure can benefit them, and free them to concentrate their development efforts only on their specialized parts.

We also hope to attract graduate students to work specifically on improving parts of *SableVM* by implementing existing and innovative techniques. For example, the current heap allocator of *SableVM* is rather naive, and uses a global lock on every object instance allocation. Improving *SableVM*'s allocator is a suitable project for early graduate courses, covering garbage collection and memory management.

10.1.2 Profiling Memory Usage

A longer term project is to build a complete memory profiling framework, in *SableVM*, as a tool for both researchers and Java *developers* to better understand memory usage in Java programs.

10.1.3 Investigate Compilation to V-CODE

Our experimental results have shown, without any doubt, that *SableVM*'s interpreter engine does indeed perform as an efficient *interpreter*, but that it is often *much* slower than just-in-time and adaptive engines.

Even though achieving the absolute highest performance is not the main goal of our research, we would like to investigate the performance we could achieve by adding a *retargetable* compiler engine, based on *V-CODE* [Eng96]. This would provide a new level of performance-portability tradeoff to users of the *SableVM* framework.

10.2 Conclusions

In this thesis, we have introduced the *SableVM* research framework. The objective of our research was to design and implement a portable and easily modifiable virtual machine that could be used for research on various aspects of Java bytecode execution. We also wanted to evaluate the performance achievable by such a portable system.

More specifically, in this thesis we introduced a *preparation sequence* technique to allow the efficient implementation of an *inline-threaded* interpreter engine in a multi-threaded environment. Then we introduced a *logical* partitioning of the runtime memory of a Java virtual machine that greatly simplifies memory management, and opens interesting opportunities for further optimizations. One such optimization, that we also introduced, is the implementation of *sparse interface virtual tables*, without memory loss. Our technique takes advantage of the *class-loader specific* memory manager to recycle the memory holes in the sparse tables. We also introduced a simple technique for computing space-efficient maps for *precise* (or type-accurate) garbage collection. Then we introduced a bidirectional layout that simplifies garbage collection tracing, and we introduced a technique to eliminate *spin locking* from thin locks.

Our experimental results showed that inline-threading Java code yields significant performance improvement over both traditional switch based interpretation and direct-threaded interpretation. They also showed that our simple technique for recycling sparse interface table holes is highly effective, resulting in no memory loss across all our tests. Our results showed that our technique for computing gc maps builds very few distinct bit maps, only 74 maps for near 40,000 gc check points (approximately 1 bit map per 535 check points). This algorithm, though, causes an increase of up to 13% in the number of *live* local variables due to splitting. Our measurements showed that the object layout has no significant impact on garbage collection time, but can sometimes affect total execution time of benchmarks positively or negatively.

Finally, our overall comparative performance measurements showed that a highly-portable, simple-to-modify virtual machine implementing the techniques proposed in this thesis can achieve comparable performance to a state-of-the-art interpreter-based virtual machine, and is significantly faster than a naively implemented Java bytecode interpreter. They also revealed that, while such an interpreter greatly under-performs high-performance adaptive systems, it still offers an acceptable performance relative to naive just-in-time compilers.

The portability of *SableVM* was demonstrated by the simplicity of porting it to other platforms. In particular, porting *SableVM* to the Debian/Alpha system took less than 24 hours and less than 50 lines of code.

Appendix A
A Mini SableVM User Guide

This appendix lists a minimal set of commands to get *SableVM* up and running.

A.1 Getting and Compiling SableVM

SableVM can be downloaded from [Sabb]. The full distribution consists of three compressed `tar` archives:

- `sablevm-x.y.z.tar.gz`[1]: This file contains the source code of the *SableVM* virtual machine.
- `sablevm-class-library-x.y.z.tar.gz`: This file contains the source code of the Java class libraries developed by the GNU Classpath project, slightly modified for *SableVM*.
- `sablevm-native-library-x.y.z.tar.gz`: This file contains source code of the native C implementations of native class library methods, developed by the GNU Classpath project.

Here are the steps to compile and install *SableVM*:

1. Download the three files of the distribution.

[1] `x.y.z` stands for the version number.

2. Uncompress the `sablevm-x.y.z.tar.gz` file.

3. Read the README file.

4. Follow instructions in the INSTALL file.

A.2 Customizing SableVM

The procedure, for customizing and recompiling the *SableVM* virtual machine (not its class libraries) is the standard GNU procedure:

```
$ cd sablevm-x.y.z
$ ./configure --help
... /* many options shown */
$ ./configure [options]
$ make clean
$ make
$ make install
```

Here are the *SableVM* specific configuration options:

```
  --enable-debugging-features
                          Add compiler and runtime checks
  --disable-signals-for-exceptions
                          Do not use signals to detect some exceptions
                          (NullPointerException, ArithmeticException, etc.)
  --with-gc=TYPE          Use given garbage collector (none,copying)
  --with-obj-layout=TYPE  Use given object layout (bidirectional,traditional)
  --with-threading=TYPE   Use given interpreter threading flavor
                          (inlined,direct,switch)
```

A.2.1 Advanced Customization

Within the `configure.ac` file, there are two options which can be enabled by uncommenting the appropriate line[2].

```
dnl *** uncomment if you want to insert a magic value in every object instance for debugging ***
dnl AC_DEFINE(MAGIC,1,put "SableVM" in every instance)

dnl *** uncomment to print some statistics on VM exit ***
dnl AC_DEFINE(STATISTICS,1,print statistics on VM exit)
```

[2] The line-comment delimiter is: `dnl`.

The first option is very helpful for debugging garbage collectors, as it causes *SableVM* to insert a magic value in every object instance header, and to check that this value is not corrupted at key points, such as when a reference is pushed on the operand stack, or when garbage collection is done.

The second option adds various counters in *SableVM* and causes it to write a set of statistics to the standard output at the end of its execution.

A.3 Running SableVM

As long as the *SableVM* executable is located in one of the directories on the `PATH`, it can be started by simply typing:

```
$ sablevm --help
Usage: sablevm [OPTION]... CLASSNAME [ARGUMENT]...
  -c, --classpath="PATH"          set class path
  -p, --property="NAME=VALUE"     set system property
  -v, --verbose                   enable all verbose options
  -q, --quiet                     disable all verbose options
  -s, --verbose-class             enable verbose class loading
  -S, --no-verbose-class          disable verbose class loading
  -g, --verbose-gc                enable verbose garbage collection
  -G, --no-verbose-gc             disable verbose garbage collection
  -j, --verbose-jni               enable verbose JNI
  -J, --no-verbose-jni            disable verbose JNI
  -y, --copyright                 display copyright
  -Y, --no-copyright              do not display copyright
  -L, --license                   display license information and exit
  -V, --version                   display version information and exit

Help options:
  -?, --help                      Show this help message
  --usage                         Display brief usage message
$ sablevm HelloWorld
Hello world!
$ sablevm --classpath=hello2.jar HelloWorld2
Hello again, world!
$
```

By default, *SableVM* searches for *application classes* in the package directory tree rooted at the current user directory. The `--classpath` option can be used to explicitly specify a set of directories and `*.jar` archives to be searched. This parameter *does not* affect the search and loading of *bootstrap classes*.

A.3.1 Advanced Command-Line Options

Advanced command-line options can be specified through the `--property` option. System properties are used to specify the various parameters of internal *SableVM* modules, such as the garbage collector. The list of recognized system properties vary depending on the features compiled into *SableVM*.

The current list of supported system properties is:

```
sablevm.boot.class.path:
  bootstrap class lookup directory

sablevm.boot.library.path:
  bootstrap native library lookup directory

sablevm.stack.size.min
sablevm.stack.size.max
sablevm.stack.size.increment:
  stack parameters

sablevm.classloader.heap.size.min
sablevm.classloader.heap.size.max
sablevm.classloader.heap.size.increment:
  class loader memory parameters

#if defined (_SABLEVM_NO_GC)

  sablevm.heap.size:
    maximum heap size

#elif defined (_SABLEVM_COPY_GC)

  sablevm.heap.size.min
  sablevm.heap.size.max
  sablevm.heap.size.increment:
    heap parameters

#endif /* defined (_SABLEVM_NO_GC) */

#if !defined(NDEBUG)

  sablevm.verbose.methods
  sablevm.verbose.instructions:
    verbose execution trace

#endif

Example:

  sablevm --property="sablevm.verbose.methods=true" HelloWorld
```

Additional system properties can be easily created using *m4* macros in the file `src/libsablevm/vm_args.m4.c`.

Appendix B
Alpha Port Diffs

This appendix lists the *unified diffs* `diff -u` of the *SableVM* port to the Debian GNU/Linux operating system on the *Alpha* processor.

B.1 jni_system_specific.h

```
--- src/libsablevm/include/jni_system_specific.h        6 Aug 2002 10:27:22 -0000       1.3
+++ src/libsablevm/include/jni_system_specific.h        15 Aug 2002 04:48:53 -0000      1.4
@@ -8,15 +8,17 @@
    u = unsigned intger,  s = signed integer, f = float, d = double
    8,16,32,64 = 8 bits, 16 bits, ...
    So, "u8" means an 8 bits unsigned integer. */
+
+/* alpha and i386 are identical here */
+
-#if (defined (__i386__) && defined (__GNUC__))
+#if ((defined (__alpha__) || defined (__i386__)) && defined (__GNUC__))

 #define JNICALL
 #define JNIEXPORT
```

B.2 system.h

```
--- src/libsablevm/system.h     6 Aug 2002 10:27:22 -0000       1.3
+++ src/libsablevm/system.h     15 Aug 2002 04:48:53 -0000      1.4
@@ -48,7 +48,7 @@

 */

-#if (defined (__i386__) && defined (__GNUC__))
+#if ((defined (__alpha__) || defined (__i386__)) && defined (__GNUC__))

 /* "inline" is now an official keyword since the latest C standard (1999).
    So, it is a reasonable assumption to expect a target compiler to
@@ -63,19 +63,36 @@
  *
  * I guess that on most architectures, an "unsigned int" is a "word".
  */
+
+#if defined (__i386)
+
 typedef _svmt_u32 _svmt_word;

 #define SVM_WORD_SIZE 4                /* size in bytes */
 #define SVM_WORD_BIT_COUNT 32  /* size in bits */

-/* FFI specific types */
-#define ffi_type_float32 ffi_type_float
-#define ffi_type_float64 ffi_type_double
-
 /* see comments at the head of this file */
 #define SVM_ALIGNMENT 4
 #define SVM_ALIGNMENT_POWER 2  /* 2 ^^ SVM_ALIGNMENT_POWER == SVM_ALIGNMENT */
 #define SVM_PAGE_SIZE 4096
+
+#elif defined (__alpha__)
+
+typedef _svmt_u64 _svmt_word;
+
+#define SVM_WORD_SIZE 8                /* size in bytes */
+#define SVM_WORD_BIT_COUNT 64  /* size in bits */
+
+/* see comments at the head of this file */
+#define SVM_ALIGNMENT 8
+#define SVM_ALIGNMENT_POWER 3  /* 2 ^^ SVM_ALIGNMENT_POWER == SVM_ALIGNMENT */
+#define SVM_PAGE_SIZE 8192
+
+#endif
+
+/* FFI specific types */
+#define ffi_type_float32 ffi_type_float
+#define ffi_type_float64 ffi_type_double

 /* Does ">>" behaves as a "signed" or "unsigned" shift when
    applied to a signed argument?  I personally think that the C
```

B.3 system.c

```
--- src/libsablevm/system.c     6 Aug 2002 10:27:22 -0000       1.3
+++ src/libsablevm/system.c     15 Aug 2002 05:17:13 -0000      1.5
@@ -5,7 +5,33 @@
  * modification of SableVM.                                  *
  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

-#if (defined (__i386__) && defined (__GNUC__))
+#if ((defined (__alpha__) || defined (__i386__)) && defined (__GNUC__))

 /*
 ----------------------------------------------------------------------
@@ -33,19 +59,45 @@
 _svmh_compare_and_swap (volatile _svmt_word *pword, _svmt_word old_value,
                        _svmt_word new_value)
 {
+  /* Yes, some inline assembly source code... Unfortunately, this
+     cannot be expressed in C. */
+
+#if defined (__i386__)
   /* On the ia32, cmpxchgl has a side effect.  When swapping fails,
      the following variable contains the value that is currently in
      *pword (presumably different from old_value). */
   _svmt_word current_value;
   _svmt_u8 result;

-  /* Yes, some inline assembly source code... Unfortunately, this
-     cannot be expressed in C. */
+/* *INDENT-OFF* */
   __asm__ __volatile__ ("lock\n\t"
                        "cmpxchgl %3, %1\n\t"
                        "sete %0"
                        :"=q" (result), "=m" (*pword), "=a" (current_value)
                        :"r" (new_value), "m" (*pword), "a" (old_value)
                        :"memory");
+/* *INDENT-ON* */
+#endif
+
+#if (defined (__alpha__))
+  register _svmt_word result, tmp;
+
+/* *INDENT-OFF* */
+  __asm__ __volatile__ ("1:  mb\n\t"                  /* make sure */
+                       "    ldq_l      %1,%4\n\t"      /* load *pword into tmp (reg,<= mem) */
+                       "    cmpeq      %1,%5,%0\n\t"   /* result = (*pword == tmp) */
+                       "    beq        %0,3f\n\t" /* nothing to do if they differ(0) - jump away */
+                       "    mov        %3,%1\n\t"      /* copy tmp<=new so that we don’t lose it */
+                       "    stq_c      %1,%4\n\t"      /* *pword = new_value (reg,=> mem) */
+                       "    beq        %1,2f\n\t"      /* store could fail! (%1 overwritten!) */
+                       "    mb\n\t"                   /* make sure */
+                       "    br         3f\n\t"         /* were done */
+                       "2:  br         1b\n\t"         /* goto "again" */
+                       "3:  nop"
+                       :"=&r" (result), "=&r" (tmp), "=m" (*pword)
+                       :"r" (new_value), "m" (*pword), "r" (old_value));
+/* *INDENT-ON* */
+#endif

   return result ? JNI_TRUE : JNI_FALSE;
 }
```

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. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, October 2000.

[AAC$^+$99] Bowen Alpern, Dick Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, and John J. Barton. Implementing Jalapeno in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 314–324. ACM Press, November 1999.

[ACFG01] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 108–124. ACM Press, 2001.

[ADG$^+$99] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the Conference on*

Object-Oriented Programming, Systems, Languages, and Applications, pages 207–222. ACM Press, November 1999.

[ADM98] Ole Agesen, David Detlefs, and J. Eliot Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 269–279. ACM Press, 1998.

[APC+96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pages 149–159. ACM Press, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., 1986.

[ATCL+98] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a Just-in-Time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290. ACM Press, 1998.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88.2, Digital – Western Research Laboratory, 1988.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268. ACM Press, June 1998.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[Cam] The Caml Language.
URL: <`http://caml.inria.fr/`>.

[Cla] Classpath.
URL: <`http://www.classpath.org/`>.

[CLS00] Michał Cierniak, Guei-Yuan Lueh, and James N. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 13–26, Vancouver, British Columbia, June 2000. ACM Press.

[Cox87] B. Cox. *Object-Oriented Programming: An evolutionary Approach*. Addison-Wesley, 1987.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70. ACM Press, October 1989.

[DDD] The Data Display Debugger.
URL: <`http://www.gnu.org/software/ddd/`>.

[Deb] Debian-Alpha Mailing-List.
URL: <`http://lists.debian.org/debian-alpha/`>.

[Dri93] Karel Driesen. Selector table indexing & sparse arrays. *SIGPLAN Notices: Proc. 8th Annual Conf. Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 28(10):259–270, September 1993.

[Dri01] Karel Driesen. *Efficient Polymorphic Calls*. Kluwer Academic Publishers, 2001.

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

[Eng96] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pages 160–170. ACM Press, 1996.

[Ert] Anton M. Ertl. A portable Forth engine.
URL: <`http://www.complang.tuwien.ac.at/forth/threaded-code.html`>.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, December 1990.

[FM99] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 147–166. ACM Press, 1999.

[GCC] The GNU Compiler Collection (GCC).
URL: <`http://gcc.gnu.org/`>.

[GCJ] The GNU Compiler for Java (GCJ).
URL: <`http://sources.redhat.com/java/`>.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, 2000.

[Har] Harissa.
URL: <`http://www.irisa.fr/compose/harissa/harissa.html`>.

[Hot] HotSpot.
URL: <http://java.sun.com/products/hotspot/whitepaper.html>.

[HP96] John L. Hennessy and David A. Patterson. *Computer architecture (2nd ed.): a quantitative approach.* Morgan Kaufmann Publishers Inc., 1996.

[IKY$^+$00] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 294–310. ACM Press, 2000.

[Jik] Jikes.
URL: <http://oss.software.ibm.com/developerworks/opensource/jikes/>.

[JL96] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management.* Wiley, 1996.

[Kaf] Kaffe.
URL: <http://www.kaffe.org/>.

[Kra98] Andreas Krall. Efficient JavaVM Just-In-Time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 205–212. IEEE Computer Society Press, October 1998.

[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>.

[Mye95] Andrew C. Myers. Bidirectional object layout for separate compilation. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 124–139. ACM Press, October 1995.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming.* O'Reilly & Associates, Inc., 1996.

[NHCL98] Francois Noel, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages 1998.* IEEE Computer Society Press, April 1998.

[OK99] Tamiya Onodera and Kiyokuni Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 223–237. ACM Press, November 1999.

[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.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 146–161. ACM Press, November 1991.

[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.

[PW90] William Pugh and Grant Weddell. Two-directional record layout for multiple inheritance. *ACM SIGPLAN Notices*, 25(6):85–91, June 1990.

[Saba] SableCC, The Sable Research Group Compiler Generator.
URL: <http://www.sablevm.org/>.

[Sabb] SableVM.
URL: <http://www.sablevm.org/>.

[SAI$^+$90] Herbert Schildt, American National Standards Institute, International Organization for Standardization, International Electrotechnical Commission, and ISO/IEC JTC 1. *The annotated ANSI C standard: American National Standard for Programming Languages C: ANSI/ ISO 9899-1990.* Osborne/McGraw-Hill, Berkeley, California, USA, 1990. ISBN 0-07-881952-0.

[SLC99] James M. Stichnoth, Guei-Yuan Lueh, and Michał Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 118–127. ACM Press, 1999.

[Soo] Soot: A Java Optimization Framework.
URL: <http://www.sable.mcgill.ca/soot/>.

[SOT$^+$00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-In-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[SPE] SPECjvm98 Benchmarks.
URL: <http://www.spec.org/osg/jvm98>.

[Tob] Toba.
URL: <http://www.cs.arizona.edu/sumatra/toba/>.

[VH94] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In *Object-Oriented Programming, Proceedings of the 8th European Conference ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 432–449. Springer, July 1994.

[VHK97] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *OOPSLA '97 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 142–157. ACM Press, October 1997.

[VRCG$^+$99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON '99*, pages 125–135, 1999.

[Wir71] N. Wirth. The design of the PASCAL compiler. *Software Practice and Experience*, 1(4):309–333, 1971.

[YMP$^+$99] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioğlu, and Erik Altman. LaTTe: A Java VM Just-In-Time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 128–138. IEEE Computer Society Press, October 1999.