論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Techniques for Mixed-mode Virtual Machines to Improve Overall System Performance
著者(和文)	緒方一則
Author(English)	Kazunori Ogata
出典(和文)	学位:博士（理学）, 学位授与機関:東京工業大学, 報告番号:乙第4128号, 授与年月日:2016年4月30日, 学位の種別:論文博士, 審査員:増原　英彦,南出　靖彦,渡邊　治,首藤　一幸,脇田　建
Citation(English)	Degree:, Conferring organization: Tokyo Institute of Technology, Report number:乙第4128号, Conferred date:2016/4/30, Degree Type:Thesis doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Techniques for Mixed-mode Virtual Machines to Improve Overall System Performance

インタープリター動的コンパイラ併用型仮想マシンのシステム性能改善技術

April 2016

Kazunori Ogata

緒方　一則

Abstract

This dissertation presents techniques to improve the performance of mixed-mode virtual machines (VMs), which use both an interpreter and a dynamic compiler. Such VMs are widely used for implementing runtime systems of programming languages and have good steady-state performance and short startup times.

Production systems can be characterized with various performance metrics other than the steady-state one, such as the startup time and response time to changes of program behavior, total performance of multiple programs in a system, and the time to debug a problem in a runtime system.

The first contribution of this dissertation is a technique to increase the execution speed of an interpreter. In a mixed-mode VM, the interpreter executes a piece of code until it is dynamically compiled, and most of the code in a newly started program and the code executed just after changes of program behavior are not compiled. We improved interpreter performance by up to 30% by eliminating redundant memory loads for fetching bytecode instructions.

The second contribution is a technique to reduce the inefficiency caused by a dynamic compiler's memory allocation pattern. Efficient memory usage helps to avoid thrashing and keeps the total performance of multiple programs good even under high loads. We performed in-depth analyses of Java memory usage and found a large amount of unused memory in the system library's free list. We reduced 78% of its physical memory consumption by explicitly releasing its page frames.

The third contribution is a technique to reduce the overhead of the trace-and-replay mechanism of a dynamic compiler. A dynamic compiler is very difficult to debug because simply rerunning the same program does not necessarily reproduce the problem. We devised a technique to implement a trace-and-replay functionality for the dynamic compiler by recording all inputs to the compiler with a small overhead. We utilized a process dump to reduce the amount of data recorded during the execution of a program.

Acknowledgements

It is a great pleasure to thank the people who made this dissertation possible.

First of all, I would like to express my great appreciation to Professor Hidehiko Masuhara, the chair of the dissertation committee, for his suggestions and support over a long time. Without his continuous help and patience, I could not have completed the dissertation.

I would also like to thank the dissertation committee members, Professor Osamu Watanabe, Professor Yasuhiko Minamide, Professor Ken Wakita, and Professor Kazuyuki Shudo, who spent a lot of their valuable time on the examination, and who gave me valuable comments and advice for improving the quality of this dissertation.

This dissertation is based on several research projects I did in IBM Research – Tokyo. Therefore, I am deeply indebted to my colleagues in the company. Hideaki Komatsu gave me a chance to move to the research division, and his advice and insights into system software research were very helpful. Toshio Nakatani, my former manager for nine years, gave me a lot of advice on pursuing research projects and writing papers. His keen comments greatly improved my capability as a researcher. Tamiya Onodera, my manager for nine years, gave me inspiration and advice on these projects. He was very helpful in making them happen and moving them forward. Kiyokuni Kawachiya gave me brilliant comments that improved the quality of my work in various areas. Moreover, my thanks go out to other members of IBM Research – Tokyo who developed the Java VM and the JIT compiler. The daily conversations with them gave me valuable ideas on the performance of Java runtime systems.

The work described in this dissertation would not have been possible without the state-of-the-art, industry-leading Java VMs and the JIT compilers. In addition to the team in IBM Research – Tokyo, I thank the teams involved in development of the IBM Java VMs and Java JIT compilers, namely IBM Java Technology Center in Hurley, UK, the IBM J9 team in Ottawa, Canada, and the IBM TR JIT compiler team in Toronto, Canada. Especially, our regular discussions with Andrew Low, Trent Gray-Donald, Peter Shipton, and Wen Hsin Chang of the J9 team were indispensable to the success of this work.

Contents

Abstract ..... **i**

Acknowledgements ..... **ii**

Contents ..... **iii**

List of Figures ..... **vii**

List of Tables ..... **ix**

1. Introduction **1**

1.1 Thesis Statement 1

1.2 Runtime Systems for Programming Languages 1

1.2.1 Runtime Systems Using Statically Compiled Machine Code 2

1.2.2 Runtime Systems Using Interpreters 2

1.2.3 Runtime Systems Using VMs 3

1.2.4 Production VMs and Their Performance Metrics 3

1.3 Approaches to Implementing Emulation Engines of VMs 4

1.3.1 Interpreter-based Emulation Engines 5

1.3.2 Compile-everything Style Emulation Engines 5

1.3.3 Mixed-mode Emulation Engines 5

1.4 Structure of Mixed-mode VM for Programming Languages 6

1.5 Performance of a Mixed-mode Production VM 9

1.5.1 Improving the Performance of an Interpreter 10

1.5.2 Reducing the Overhead of the Dynamic Compiler 10

1.5.3 Improving the Debugging Functionality with Little Overhead 11

1.6 Organization of This Dissertation 12

2. Java Programming Language **13**

2.1 Java Platform 14

2.2 Java Programming Model 15

2.3 Java Virtual Machine 16

2.3.1 Java Class File 16

2.3.2 Loading and Accessing Classes 17

2.3.3 Java Bytecode Instruction Set 18

2.4 Java Just-in-Time (JIT) Compiler 19

2.4.1 Optimization Techniques for Java JIT Compilers 20

2.5 Production Java VM 21
2.5.1 The Reference Java VM 21
2.5.2 IBM Development Kit for Java 21
3. Background 23
3.1 Bytecode Interpreter 23
3.1.1 Structure of a Naive Interpreter 23
3.1.2 Threaded Code Interpreter 23
3.2 Memory Management 24
3.2.1 Physical Memory Management by OSes 24
3.2.2 Memory Management APIs 26
3.3 Execution Models of User Programs and Static Compilers 27
3.3.1 Execution Models of a General User Program and a Static Compiler 27
3.3.2 Execution Models for Debugging a General User Program and a Static Compiler 28
3.3.3 Diagnostic Log File of Compilers 29
3.4 Java Programs Used for Measurements 31
4. Fast Bytecode Interpreters 33
4.1 Overview 33
4.2 Our Implementation 35
4.2.1 Base Interpreter 35
4.2.2 Stack Caching 35
4.2.3 Position-based Handler Customization 37
4.2.4 Position-based Speculative Decoding 39
4.3 Applicability 41
4.3.1 Architectural Aspects of PHC 41
4.3.2 Architectural Aspects of PSD 42
4.4 Performance Evaluation 42
4.4.1 Improvement by WT and DS 43
4.4.2 Improvement by PHC 45
4.4.3 Improvement by PSD 46
4.5 Analysis of Memory Access 47
4.5.1 Reduction of Memory Accesses 47
4.5.2 Increase in I-Cache Miss Ratios 48
4.6 Related Work 49

4.6.1 Memory access for stack operations 50
4.6.2 Interpreter dispatching 50
4.6.3 Trace-based compiler for implementing interpreters 52
4.7 Summary 52
5. Reduction of Java VM Memory Usage 53
5.1 Overview 53
5.2 An Anatomy of Non-Java Memory 55
5.2.1 Code Area 56
5.2.2 JVM Work Area 56
5.2.3 Class Metadata 57
5.2.4 JIT Compiled Code 57
5.2.5 JIT Work Area 57
5.2.6 Malloc-then-freed Areas 57
5.2.7 Management Overhead 58
5.2.8 Stack 58
5.3 Methodology to Measure Non-Java Memory 58
5.3.1 Our Approach 58
5.3.2 Gathering OS-level Memory Management Information 60
5.3.3 Gathering Memory Usage in Java VM 60
5.3.4 Computing Non-Java Memory Usage 60
5.4 Micro-Benchmarks 61
5.4.1 Micro-benchmark for the Class Metadata 61
5.4.2 Micro-benchmark for the JVM Work and Malloc-then-freed Areas 63
5.5 Macro-Benchmarks 64
5.5.1 WAS 7.0 Running Apache DayTrader 65
5.5.2 DaCapo 66
5.6 Reducing the Resident Set Size of the Malloc-then-freed Area 68
5.6.1 Run-to-run Fluctuation of the Resident Set Size of the Malloc-then-freed Area 68
5.6.2 Releasing Physical Pages to Reduce the Resident Set Size 69
5.6.3 Savings by Calling the Madvise System Call 70
5.6.4 Performance Impact of Calling the Madvise System Call 71
5.6.5 Discussion 72
5.7 Related Work 74

5.8 Summary 76
6. Replay Debugging of the Java JIT Compiler 79
6.1 Overview 79
6.2 Reproducing the Behavior of a JIT Compiler 81
6.2.1 Execution Models for a JIT Compiler 81
6.2.2 Difficulty in Debugging a JIT Compiler 82
6.2.3 Reproducing JIT Compilation Using a Mock Environment 83
6.3 Creating a Mock Environment for a JIT Compiler 84
6.3.1 Trace-and-replay Techniques for a JIT Compiler 84
6.3.2 Runtime Information as Inputs 85
6.3.3 Variable and Fixed Inputs 86
6.4 Overview of Replay Compilation 86
6.4.1 System Dump Based Approach 87
6.4.2 Log Structure 87
6.4.3 Building State-saving and Replaying Compilers 90
6.4.4 Replaying the Compilation 91
6.4.5 Further Reducing the Size of Logs 92
6.4.6 Discussion 93
6.5 Implementation 94
6.5.1 State-Saving Compiler 94
6.5.2 Replaying Compiler 95
6.6 Experimental Results 95
6.6.1 Replayed Compilation 96
6.6.2 The Size of a Log 96
6.6.3 Compilation Time 99
6.6.4 Execution Speed 100
6.7 Related Work 100
6.8 Summary 101
7. Conclusion 103
Improving the Performance of a Mixed-mode Bytecode Interpreter 103
Reducing the Memory Overhead of the JIT Compiler 104
Improving Debugging Functionality with Small Overhead 105
Bibliography 107

List of Figures

Figure 1.1 Components used to execute a program written in a VM-based programming language 7
Figure 2.1 Example of Java bytecode instructions 19
Figure 3.1 Execution model of user program and static compiler 28
Figure 3.2 Execution models when a user programs crashes and when a problem in the static compiler causes the user program to crash 29
Figure 3.3 Execution models when a user program or a static compiler is re-executed to generate diagnostic output 30
Figure 4.1 The base interpreter 36
Figure 4.2 The WT+PHC interpreter 38
Figure 4.3 The WT+PHC+PSD interpreter 40
Figure 4.4 Relative performance over the base interpreter 44
Figure 4.5 The bytecode handlers for iaload bytecode 45
Figure 4.6 The breakdown of memory accesses 46
Figure 4.7 The I-cache miss ratios 49
Figure 4.8 The relative I-cache miss ratios over the base interpreter 49
Figure 5.1 Breakdown of non-Java memory when Apache DayTrader is running on WebSphere Application Server 56
Figure 5.2 Correspondence between memory allocation paths and the eight categories of non-Java memory 59
Figure 5.3 Changes in non-Java memory due to repeated reflective invocation on x86 63
Figure 5.4 Changes in non-Java memory due to repeated reflective invocation on POWER 63
Figure 5.5 Change in non-Java memory due to allocating and freeing direct byte buffers on x86 64
Figure 5.6 Change in non-Java memory due to allocating and freeing direct byte buffers on POWER 64
Figure 5.7 Non-Java memory for WAS 7.0 running Apache DayTrader on x86 65
Figure 5.8 Non-Java memory for WAS 7.0 running Apache DayTrader on POWER 65
Figure 5.9 Results of DaCapo bloat on x86 67
Figure 5.10 Results of DaCapo bloat on POWER 67
Figure 5.11 Results for DaCapo bloat on x86 when the size of malloc-then-freed area is large 68
Figure 5.12 Non-Java memory breakdown for WAS 7.0 running Apache DayTrader on x86 when madvise() is called as the JIT work areas are freed 70

Figure 5.13 Non-Java memory breakdown for WAS 7.0 running Apache DayTrader on POWER when madvise() is called as the JIT work areas are freed 70
Figure 5.14 Non-Java memory breakdown for DaCapo bloat on x86 when madvise() is called as the JIT work areas are freed 71
Figure 5.15 Non-Java memory breakdown for DaCapo bloat on POWER when madvise() is called as the JIT work areas are freed 71
Figure 5.16 Relative performance of a Java VM that calls madvise() when freeing the JIT work areas compared to the Java VM without madvise() 72
Figure 5.17 Disk I/O rate and the amount of swapped data during execution of two WAS processes running Apache DayTrader when madvise() was not called 73
Figure 5.18 Disk I/O rate and amount of swapped data during execution of two WAS processes running Apache DayTrader when madvise() was called for the JIT work area 73
Figure 6.1 Execution model of a JIT compiler 82
Figure 6.2 Execution model for debugging a JIT compiler using a mock environment 83
Figure 6.3 Replay JIT compilation 88
Figure 6.4 An example of the code to get the input for the compiler 91
Figure 6.5 Reduction in the size of log with zlib and filter 97
Figure 6.6 Increase in compilation time without and with zlib compression 99

List of Tables

Table 4.1 Evaluated programs 43
Table 4.2 Execution environment 43
Table 4.3 Evaluated interpreters 43
Table 4.4 The average (maximum) performance improvement 44
Table 4.5 The categories of memory accesses 46
Table 4.6 The average (maximum) percentage of memory accesses of each category 47
Table 5.1 Categories of non-Java memory 56
Table 5.2 Execution environment for x86 62
Table 5.3 Execution environment for POWER 62
Table 5.4 Configurations to run DaCapo `bloat` for both x86 and POWER 66
Table 5.5 Execution environment for measuring disk I/O for swap-in and swap-out 72
Table 6.1 Types of input used by the Java JIT compiler and the values to be saved in a log 89
Table 6.2 Configurations of the tested machines 96
Table 6.3 Evaluated programs 96
Table 6.4 Comparison in size: diagnostic output vs. log 97
Table 6.5 Reduction in the number of logs 98
Table 6.6 Comparison in size: system dump vs. Java heap 98

1. Introduction

1.1 Thesis Statement

It has become popular to use virtual machines (VMs) as the runtime systems of programming languages. In particular, *mixed-mode VMs*, which use both an interpreter and a dynamic compiler, are widely used as they have good performance and portability.

The goal of this research is to improve the overall performance of a production system by increasing the efficiency of a mixed-mode VM. The production system is used for running enterprise applications, and thus, it is expected to perform well in various use cases and it needs to be reliable. For example, the startup time should be short. Moreover, as the production system often runs multiple applications, the memory usage of each application should be as efficient as possible. Some of the applications may be interactive, so their response time should be short. Reliability is often measured by the amount of downtime, and ease of debugging helps reduce downtime.

For improving the efficiency of a mixed-mode VM, we focus on reducing the execution time of the interpreted code, increasing the number of VMs runnable in a machine, and adding a debugging function to a dynamic compiler with an affordable overhead. Our research is based on three theses:

- The performance of a bytecode interpreter can be improved by avoiding redundant memory accesses for fetching bytecode instructions.
- A dynamic compiler may end up wasting large amounts of memory because its memory allocation pattern does not fit well with the least recently used (LRU) policy; the waste can be reduced by explicitly reclaiming physical memory based on the allocation pattern.
- The replay debugging of a dynamic compiler can be implemented with as small overhead as it can always be turned on in production environments by utilizing a process memory dump.

1.2 Runtime Systems for Programming Languages

A programming language is a means of describing a program in a human-readable form such as plain text. For running a program written in a programming language, it needs to be converted into machine code by a compiler or emulated by an interpreter.

A runtime system for a programming language is a set of components used when a program actually runs on a machine. It includes a set of libraries that can be used by the compiled machine

code, as well as an interpreter and a dynamic compiler, but it does not include a static compiler. There are three approaches to implementing a runtime system, and they are categorized by the components used for executing the programs.

1.2.1 Runtime Systems Using Statically Compiled Machine Code

This approach uses a static compiler to convert a program written in a program language into machine code prior to the execution of the program and runs the compiled code directly on the hardware. FORTRAN, C and C++ languages are typical examples of this approach.

The advantage of this approach is good execution performance. There is only a small runtime overhead because the program runs directly on the hardware. A static compiler can use time-consuming optimization techniques to generate highly optimized machine code because the time for compilation is not part of the execution time of the program.

The disadvantage is low portability. A compiler needs to be prepared for every platform. Furthermore, a set of optimizations effective on one platform may not be so effective on another, and thus, porting requires deep knowledge of the characteristics of the target platform.

1.2.2 Runtime Systems Using Interpreters

This approach uses an interpreter to emulate the operations described in a source program. LISP, BASIC, and many dynamic scripting languages are examples that take this approach, though some implementations of dynamic scripting languages take VM-based approaches.

The advantages of this approach are high portability and flexibility. An interpreter is portable because it can be implemented in a high-level language, and thus, it can be made to work on another platform simply by recompiling its source code. An interpreter is a flexible way of handling the dynamic features of programming languages, such as reflective computing, because the statements of the user program are evaluated when they are about to be executed.

The disadvantage of this approach is low execution performance because a naive interpreter simply repeats its interpretation of the program and pays an overhead, such as for decoding each statement, looking up the appropriate variable for a given name, and evaluating the programming constructs.

1.2.3 Runtime Systems Using VMs

Runtime systems using VMs are becoming popular means of designing new programming languages. The VM[1] defines its own virtual instruction set architecture (ISA), and a program is compiled into the virtual ISA code prior to execution. The virtual ISA code is then loaded into the VM and executed by emulating the virtual instructions. VM-based runtime systems have played major roles in the design of programming languages since Java [Gosling et al. 1996] made a VM-based implementation part of its specification [Lindholm and Yellin 1996]. Thereafter, many programing languages have taken this approach. For example, C# [ECMA 2006] is implemented on a CLI-based runtime system [ECMA 2012], and Ruby [IPA 2010] uses a VM-based runtime from version 1.9 [Sasada 2006]. Note also that VM-based runtime systems had been studied for a long time before Java was published in 1995, and include, for example, the UCSD p-code system [Nori et al. 1975] of PASCAL, the SECD machine [Landin 1964] of LISP, Smalltalk-80 [Deutsch and Schiffman 1984].

The advantage of VM-based runtime systems is the flexibility in implementing the VM. This flexibility stems from the virtual ISA, which is a well-defined interface to abstract the implementation of a VM and allows VM implementers to use any design that fits the target platform. Accordingly, VM implementers can improve portability, execution performance, or both. For example, they can use an interpreter if portability is important or if the target platform is a small embedded device. They can use a dynamic compiler if execution performance is important. The virtual ISA also helps improve the portability of the static compiler that converts the source code into virtual ISA code because the target platform of the compiler is the same regardless of the actual target hardware.

The disadvantage of this approach is the complexity of the runtime system because the VM itself is a complex component.

1.2.4 Production VMs and Their Performance Metrics

This research focuses on improving the performance and reducing the memory usage of production VMs, which are designed to be used in production systems. Examples of production VMs include Oracle's HotSpot VM [Sun Microsystems 2001] and IBM SDK for Java [Suganuma et al. 2000;

[1] Although there are two types of VM, namely a system VM and process VM [Smith and Nair 2005], we assume that this VM is a process one because process VMs are more common as runtime systems of programming languages.

Grcevski et al. 2004]. These VMs are regularly updated for improving performance and adding new features including debugging functions [Oracle 2008; IBM 2012].

The updates of these production VMs often aim at improving the overall performance of production systems, where customers' enterprise applications are executed. Note that the overall performance of production systems is measured not only by the peak throughput, but also by various metrics, such as the length of downtime and stability in performance. It is more important for production systems to reduce downtime and to avoid significant performance degradation in any usage scenario than to achieve higher peak performance, because these metrics directly affect the downtime and performance stability of customers' applications, which are expected to be always available and performing well. To improve these metrics, we need to improve the execution speed of the interpreter, as well as that of dynamically compiled code, so the system can achieve better performance even when most of application code is not compiled, such as the application startup phase.

It is also important to increase the number of VMs runnable in a machine because production systems, especially in cloud data centers, tend to run multiple VMs in a single machine. An approach for increasing the number of runnable VMs is to reduce the memory usage of each VM and keep the total working set size smaller than the amount of physical memory, otherwise thrashing severely hurts the system performance. Since a single machine of a production system can run more than a hundred VMs, a small memory overhead in each VM will be accumulated to a large amount as a whole, which could cause thrashing.

1.3 Approaches to Implementing Emulation Engines of VMs

A key component of a VM-based runtime system is the emulation engine [Smith and Nair 2005], which executes virtual ISA instructions. The components to emulate virtual ISA instructions are the interpreter and dynamic compiler. The dynamic compiler receives a piece of virtual ISA code and compiles it into machine code at runtime; then, the underlying hardware runs the generated machine code. The generated machine code is cached in memory, so that it can be reused later without recompiling the same virtual ISA code.

There are three approaches to implementing an emulation engine, depending on the set of components used for the implementation.

1.3.1 Interpreter-based Emulation Engines

VMs only using a bytecode interpreter have been studied for a long time; they include the UCSD p-code system [Ammann 1977] and some implementations of the FORTH programming language [ISO/IEC 1997].

The advantages of this approach are ease of development, fewer resource requirements, and fast startup upon initial execution of a piece of code. These advantages also help adjust the runtime system's behavior quickly when programs change their behavior.

The disadvantage is low execution performance in the steady state of computation intensive code. Many techniques [Bell 1973; Ertl 1995; Ertl and Gregg 2003] have been proposed to speed up the execution of interpreters, but it is still far slower than machine code generated by a dynamic compiler.

1.3.2 Compile-everything Style Emulation Engines

VMs using only a dynamic compiler have been studied as the runtime systems of the SELF programming language [Ungar and Smith 1987; Chambers and Ungar 1989] and in some implementations of Java, such as Jikes RVM[1] [Alpern et al. 2000] and Intel ORP [Cierniak et al. 2003].

The advantages of this approach are high steady-state performance and deterministic behavior of the compiler. The latter makes this approach well suited for compiler research because a change in the compiler directly affects the performance of the benchmark programs.

The disadvantages are higher resource consumption and a long delay in the initial execution of a piece of code.

1.3.3 Mixed-mode Emulation Engines

Implementations of emulation engines using both an interpreter and a dynamic compiler are now popular for VM-based programming languages. This approach is called *mixed-mode bytecode execution* [Agesen and Detlefs 2000], and we call a VM for mixed-mode bytecode execution *a mixed-mode VM*. In this approach, a dynamic compiler only compiles the frequently executed pieces of a program and an interpreter executes the rest of the program. Here, mixed-mode VMs for Smalltalk-80 [Deutsch and Schiffman 1984] and Java VMs have been extensively studied.

---

[1] The Jikes RVM Project is available at http://jikesrvm.org/

The advantages of mixed-mode bytecode execution are high execution performance in both the startup phase and steady state, less memory consumption, and greater flexibility in the design of a dynamic compiler. The memory usage is low because the amount of code to be compiled is much smaller than the entire program. The dynamic compiler has greater flexibility because it can focus on the code that is worth optimizing and the interpreter can execute the rest.

The disadvantages are the complexity and non-deterministic behavior. In a mixed-mode VM, the code to be dynamically compiled is selected on the basis of a runtime profile, and thus, the timing at which to compile a piece of virtual code may change from one execution of the user program to another. This non-deterministic aspect can make debugging difficult.

1.4 Structure of Mixed-mode VM for Programming Languages

This section describes a typical runtime system using a mixed-mode VM. Figure 1.1 illustrates the components and data structures in a mixed-mode VM, as well as the static compiler for the source code and the object code in virtual ISA compiled from the source code, to show the entire path of running a program. The rounded boxes in the figure correspond to components and the square boxes correspond to data structures. The components in italics are those this research focuses on. Shaded boxes show the code and data of a user program executed in the VM. Arrows show the flow of executing the program written in the programming language.

The following list describes each of the components and data structures in Figure 1.1. It also describes each of the corresponding components in a Java VM [Lindholm and Yellin 1996].

Source code

This is a file containing the source code of the user program described in programming language. This file corresponds to a Java source file, whose file name extension is `.java`.

Static compiler

This is a compiler that compiles the source code into the object code of the virtual ISA. It is often called a static compiler, in contrast with the dynamic compiler in the VM. In particular, it corresponds to the `javac` command included in the Java development kit (JDK)[1].

[1] http://java.sun.com/javase/

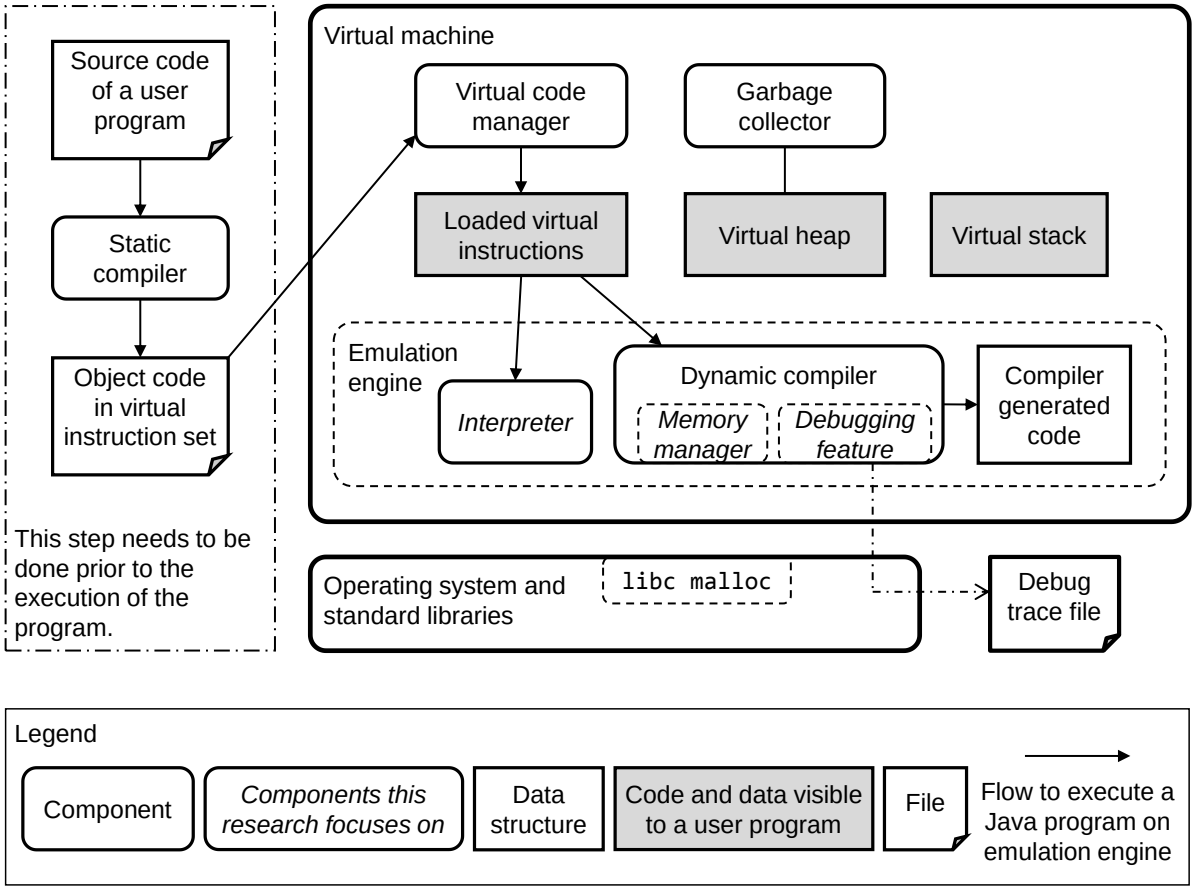


Figure 1.1. Components used to execute a program written in a VM-based programming language.

Object code in virtual ISA

This is a file containing the user program compiled into virtual ISA code. For a high-performance VM, the format of the virtual ISA is usually a bytecode that consists of a one-byte opcode and optionally one or more bytes of operands. This file corresponds to a Java class file, whose file name extension is `.class`.

Virtual code manager

This component loads the object code into memory and manages them so that the emulation engine can access the loaded code. For the VMs of object-oriented languages, this component also performs class hierarchy analysis, which helps reduce the overhead of virtual method invocation. It corresponds to a class loader, though some of its functions are implemented as library code in Java, instead of as part of the Java VM.

Loaded virtual instructions

Virtual instructions in the object code files are loaded into memory areas in the VM, so that emulation engine can access them.

Virtual heap

This is a memory area where all data created by the user program are allocated. This area corresponds to the Java heap.

Virtual stack

This is a memory area where stack frames of the user program are allocated. Local variables of the user program are also allocated here. For Java, this area corresponds to the Java stack. The virtual heap and virtual stack are memory areas that are writable from the user program.

Garbage collector

If the virtual heap uses automatic memory management, the garbage collector (GC) performs the automatic heap management tasks. The Java VM has a garbage collector, since the Java language specification requires the managed heap.

Emulation engine

This is the component to execute virtual instructions. This component includes the interpreter and dynamic compiler.

Interpreter

Modern high-performance VMs use mixed-mode bytecode execution and an interpreter to execute infrequently executed pieces of code in order to reduce the compilation time and memory consumption. OpenJDK, HotSpot VM, and IBM J9 Java VM use an interpreter, while Jikes RVM does not.

Dynamic compiler

A dynamic compiler receives a piece of frequently executed virtual instructions and generates machine code. The dynamic compiler may use runtime profile information to generate code that is more optimized for the currently executing user program and the current state of the VM. For Java, the dynamic compiler is often known as just-in-time (JIT) compiler.

Memory manager of dynamic compiler A compiler is a memory intensive program because it allocates a large number of data structures to every variable, operator, and other programming construct. For handling large numbers of memory allocations, a compiler often uses internal memory management functions, instead of directly calling standard libraries or system calls.

Debugging feature of dynamic compiler This component may be built on both a standard specification, such as JVMTI[1] for Java, and a vendor-specific one [Oracle 2008; IBM 2012]. These features are useful for application programmers to debug their program, as well as for VM developers to investigate operations in a user program that cause problems in the VM.

Compiler generated code This is a memory area that contains machine code instructions generated by the dynamic compiler.

1.5 Performance of a Mixed-mode Production VM

For improving the overall performance of a mixed-mode production VM, just improving steady-state performance is not sufficient because there are many performance metrics on which to judge production systems. That is, production systems are expected to be always available and perform well under any situation.

Startup time is one of such metric; a short startup time reduces service downtime at the beginning of a business and in the event the service needs to be restarted. Suganuma et al. reported that compiling every method causes an intolerably slow start up time [Suganuma et al. 2001]. Prompt responses to changes of program behavior is also important for keeping a system performing well.

Memory consumption of a production VM is another metric; here, production systems usually run multiple programs in a machine and the system should not start thrashing even if it has a heavy load. Note that the size of the memory tends to be the bottleneck to running many programs in modern production systems, especially in cloud datacenters.

To reduce downtime, the time used for debugging should be reduced because a mixed-mode VM is a complex runtime system. Administrators of production systems will not likely use mixed-mode VMs if they do not have efficient means of debugging.

---

[1] Java Virtual Machine Tool Interface (JVM TI). The specification is available at http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html

Following subsections explains the issues of these metrics in production environments.

1.5.1 Improving the Performance of an Interpreter

Interpreters have been used for a long time, and there are many studies on improving their performance. The major source of overhead in an interpreter is repeated operations, and the studies have tried to avoid or reduce them.

One of the repeated operations is indirect branches used to return from bytecode handler routines to the decode loop. The threaded code [Bell 1973] eliminates indirect branches by inlining the decode loop into every handler routine because frequent execution of indirect branches is a major bottleneck in an interpreter.

Redundant memory accesses are another source of overhead. For example, an interpreter of the stack machine frequently accesses the memory area for the stack in order to get operands. Dynamic stack caching [Ertl 1995] reduces the number of memory accesses by caching operands in the registers of the processor. It avoids runtime checks to keep track of the number of cached operands by preparing separate handlers customized for the number of cached operands.

Another source of redundant memory accesses is fetching bytecode instructions, whose format is often designed to be as short as possible. Since modern high-performance processors access memory in units of a word, which is typically 4 bytes or 8 bytes, the processor reads the same word repeatedly from memory to fetch bytecode instructions because a word typically contains multiple bytecode instructions. This research focuses on reducing the overhead caused by this redundancy.

1.5.2 Reducing the Overhead of the Dynamic Compiler

The sources of overhead in the dynamic compiler are the runtime profile and memory management. Note that we consider the time for compilation and optimization not to be overhead because it is a kind of investment for generating faster code. The overhead of runtime profiling is manageable by applying it adaptively; we can apply costly profiling, such as the path profile, only to the code that is frequently executed and further optimization will be beneficial. The overhead of memory management, however, is inevitable.

The memory management overhead can be broken down into overhead in execution time and overhead in memory. We focused on memory wastage because it reduces the overall performance of a production system by reducing the number of applications runnable in a machine. The overhead in

time, on the other hand, is not as severe a problem because the time needed for memory management is much smaller than the time for compilation.

In order to reduce the memory usage of a dynamic compiler, we need to be aware that the memory usage pattern of a dynamic compiler is different from typical patterns of other components in a mixed-mode VM in two ways. One is that it allocates a large amount of working memory and frees all of it at the end of compilation, and the other is that it runs intermittently. Since the memory manager of the system library keeps memory blocks in the free list when they are freed, this allocation pattern of the dynamic compiler puts large memory blocks into the free list. Unfortunately, most of the memory blocks will not be reused until the next compilation because they are too large to be reused by other components.

The virtual memory manager of the OS cannot reclaim the physical pages of the memory blocks freed by the dynamic compiler because most of them become freed more recently than the other memory blocks in the free list. Since the OS treats the free list as part of the process's memory, the lagging free list increases the memory usage of the VM and reduces the number of VMs runnable in a machine as well as the total performance of the system.

To reduce memory wastage in the free list, we focused on the internal memory manager in the dynamic compiler. The internal memory manager knows when a compilation finishes and the freed memory blocks are unlikely to be reused soon, and thus, it can easily identify when it should tell the OS that the physical memory pages of the freed block can be reclaimed.

1.5.3 Improving the Debugging Functionality with Little Overhead

Debugging a dynamic compiler is especially difficult because its dynamic optimizations cause it to behave non-deterministically. Simply re-executing the same program may not reproduce the problem. Here, a record-and-replay debugging would be an effective way to reduce the time needed to debug such non-deterministic programs. However, replay debugging entails a large overhead of recording the debugging information at runtime, and it increases the compilation time and memory usage.

This research focuses on an implementation of replay debugging with a small enough overhead for it to be always enabled in production environments. To reduce the overhead of replay debugging, we utilize a system dump, which is created when a program crashes, as much as possible to save data for replay debugging.

1.6 Organization of This Dissertation

The remaining chapters of this dissertation are organized as follows.

Chapter 2 details the architecture and implementation of the Java VM as an example of a VM-based programming language implementation, and Chapter 3 describes the background of this work.

The next three chapters show our approach to achieving a better balance in the trade-offs described in Chapter 1. Chapter 4 describes a technique to improve the execution performance of the bytecode interpreter, which reduces the number of memory loads for fetching virtual instructions and reduces the overhead of indirect branches by speculatively decoding virtual instructions. Chapter 5 gives a detailed breakdown of the memory usage in a Java virtual machine and proposes a management approach that reduces its memory footprint by taking account of the allocation pattern of the dynamic compiler. Chapter 6 describes a technique to implement a record-and-replay debugging feature for a dynamic compiler with a small overhead.

Chapter 7 concludes the dissertation.

2. Java Programming Language

This chapter describes the features of the Java programming language and its runtime system in terms of how they affect the design of mixed-mode VMs.

Java is an object-oriented programming language that uses a VM-based runtime system (Java VM). Java is developed by Sun Microsystems, Inc.[1], and the first beta version was published in 1995. It has been continuously improved by the addition of new language features and standard class libraries. It is specified by a set of three specifications: the Java language specification [Gosling et al. 1996], Java virtual machine specification [Lindholm and Yellin 1996], and Java API specification [Chan et al. 1998].

Sun Microsystems, Inc. also released a reference implementation of the Java VM, and it has since evolved into an open source version called OpenJDK[2]. Thanks to its improving performance and language features, Java has become one of the most widely used programming languages and is comparable in popularity to other major languages such as C and C++[3].

Java has the three major advantages:

Platform independence

Java is designed to be executed in a VM-based runtime system and the specification of the Java VM is precise enough to ensure portability. A rich set of standard APIs is helpful for ensuring portability by abstracting system resources, such as files and network sockets, in a platform-independent manner. Most of the development tools are also portable as they are written in Java.

Security

Java has built-in mechanisms to ensure security. The Java VM ensures the safety of the code through a combination of load time verification and runtime checking of data. It verifies code as to its structural correctness and type safety. Structural correctness ensures that the code never accesses other code or data out of its bound, such as by branching beyond the end of the code or accessing undefined local variables. The Java VM performs runtime checking, such as array

[1] Oracle Corp. acquired Sun Microsystems, Inc. in April, 2009.

[2] OpenJDK is available at http://openjdk.java.net/.

[3] TIOBE Software BV. TIOBE Programming Community Index for October 2014.
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (referenced on October 26, 2014).

bound checking and downcast checking, whenever it needs runtime data. Java also has a built-in policy-based security management system that allows users to decide which code can access which resources. The built-in security system allows remote code to be executed safely, and thus, it has helped Java become a platform for network computing.

High productivity

The Java programming environment includes wide range of standard APIs, such as those for networking and handling XML. Programmers can rely on the APIs and thereby reduce the amount of newly developed code. Garbage collection also helps improve productivity by freeing developers from having to deal with hard-to-debug memory leak problems.

The following sections describe the highlights of the Java language, the major components of the Java VM, and production Java VMs, especially the IBM implementation of the Java runtime system that we used for the evaluations.

2.1 Java Platform

Since Java covers a wide variety of target systems ranging from embedded devices to enterprise servers, there are three *editions* of Java platforms categorized by their size and system functionality. Note that the differences between these editions are in the sets of standard libraries, whereas the virtual ISAs are the same. Thus, the same implementation techniques can be used for all of these editions if the underlying hardware has sufficient resources.

In this research, we used the Java standard edition (Java SE) and Java enterprise edition (Java EE) [Bodoff et al. 2004] as example production systems.

Java micro edition (Java ME)

This edition is mainly for embedded devices, such as mobile phones, set-top boxes, and car navigation systems. It provides a limited set of APIs based on profiles prepared for each class of target device. Dynamic compilers are not common for Java ME platforms because of resource limitations. In such case, the execution performance of the interpreter becomes important.

Java standard edition (Java SE)

This is the fundamental edition of the Java platform. It includes all APIs for desktop and simple

server applications. The Java VM for Java SE usually uses a mixed-mode VM to achieve high steady-state performance and a short start-up time.

Java enterprise edition (Java EE)

This edition adds a rich set of additional libraries on top of the Java SE platform. These libraries are useful for implementing enterprise applications such as Web application servers and transactional applications.

2.2 Java Programming Model

Java is a statically-typed, class-based object-oriented programming language. It is designed to be simple and easy to learn [Gosling and McGilton 1996]. For example, classes only support single inheritance, and `interface` defines a common interface across classes, so it avoids the complexity caused by multiple inheritance.

Java has two language features that affect the behavior of a mixed-mode Java VM: dynamic class loading and built-in multi-threading. Dynamic class loading is a feature that lazily loads a class when it is actually used. This feature allows a Java program to specify the classes to load during its execution. A class can be unloaded when it becomes unused. Dynamic class loading is useful for implementing flexible middleware and software frameworks that need to replace classes without restarting, such as Web application servers[1] and the OSGi framework [OSGi 2003].

Built-in multi-threading lets programmers write a portable multi-threaded program simply by using standard APIs to create and synchronize threads. Thus, many server programs, such as Web application servers, have been developed in Java to easily exploit underlying symmetric multiprocessing (SMP) hardware. Java-based client programs are also common, especially for GUI-based programs such as eclipse IDE[2],because the event-driven programming model is well suited to multi-threaded programs.

These two features cause a dynamic compiler to behave non-deterministically because a dynamic compiler uses the internal states of a mixed-mode VM, such as a set of loaded classes, but they may

[1] Examples of production Web application servers are IBM WebSphere Application Server (http://www.ibm.com/software/webservers/appserv/was/) and Oracle GlassFish (http://glassfish.java.net/).

[2] Eclipse IDE is available at http://www.eclipse.org/

change from run to run. Although this is a reason why a dynamic compiler is effective, debugging the compiler may be very difficult as it tends to be hard to reproduce the problem.

2.3 Java Virtual Machine

The Java virtual machine is the VM-based runtime system for the Java programming language. Although there are implementations that support static compilation of a Java program into native binary and executing it without using Java VM[1], they are not common for Java SE platforms, because they put limitations on the programming model, such as no dynamic class loading.

The Java source code is compiled into Java class files by using a static compiler (`javac`) and loaded into the Java VM where it is executed. A Java VM is usually implemented as a process VM, and thus, a new VM process is started to execute a Java program. The Java VM loads the class of the main entry point and starts executing the virtual instructions (*Java bytecode*) from the entry point. It loads other classes when the program accesses a class that has not been loaded.

The structure of a Java VM is basically the same as that of the typical mixed-mode VM shown in Figure 1.1. The only difference is that the Java VM has a separate stack (*operand stack*) for holding the operands of bytecode because the Java VM is a stack machine.

The rest of this section describes the Java VM features that constitute relevant background on our research described in Chapter 4, 5, and 6.

2.3.1 Java Class File

This subsection explains the content of a Java class file to give a background for Chapter 6, which is on efficient techniques for recording debugging information during a JIT compilation.

A Java class file contains Java bytecode, constant values, and other elements as described below. A class file can be loaded from local disks, a remote computer through the Internet, or a memory area that contains dynamically generated class data.

A class file contains the following elements. Note that the largest elements in it are the bytecode and string values in the constant pool.

[1] GCJ is an example of a Java static compiler, and it is available at https://gcc.gnu.org/onlinedocs/gcj/

Meta data

This data describes the attributes of the class and the class file itself, such as the file format version number, name and access permission of the class, and name of the super class.

Constant pool

This data contains references to members and constants appearing in the source code. References include both internal and external ones. Constants include both literals and identifiers such as the names of classes, methods, and fields.

Member definition

This is the metadata on the members implemented in this class, that is, the fields, methods, and inner classes. The metadata includes the types and access permissions of the members. Character sequences of members' names are stored in the constant pool.

Bytecode

The virtual ISA instructions of the methods are implemented in a class. A class file only contains the bytecode of the methods implemented in the class. When an inherited method is called, the Java VM refers to the implementation in the super classes.

Debug information

A class file optionally contains the data for the debugger, such as the mapping between the bytecode instructions and the line number in the source code.

2.3.2 Loading and Accessing Classes

This subsection explains how Java VM handles dynamic class loading as background for Chapter 6, which is on efficient techniques for recording operations during a JIT compilation.

A class is loaded when it is accessed for the first time. The first access to a class occurs when a symbolic reference to the class is *resolved*. Class resolution is an operation in which the Java VM converts a symbolic reference into a binary form that directly points to the data structure of the referenced class.

The bytecode loaded into a Java VM is read-only, so that the load-time verification of the bytecode is kept valid throughout the execution of the program. Although the old version of the

reference Java VM, namely the one described in the first edition of the Java VM specification [Lindholm and Yellin 1996], modified the bytecode during execution, the modification still keeps the semantics of each bytecode instruction.

A class can be unloaded when it becomes unused. The Java VM uses class loaders to load and unload classes. A class loader loads classes one by one as they are accessed, but it unloads all of them at once. Classes loaded by a class loader become unloadable when none of the classes is used by any of the other classes loaded by other class loaders.

The operations for loading and unloading classes produce two properties that are useful for the dump-based debugging explained in Chapter 6. One is that the bytecode sequence when it is loaded from a class file and when it is saved into a dump file remains the same because it is read-only. The other is that a resolved reference will never go back to an unresolved state because the class pointed to by a resolved reference and the class that contains the reference must be unloaded at the same time.

2.3.3 Java Bytecode Instruction Set

This subsection explains the basics of the Java bytecode instruction set and its characteristics as background for Chapter 4, which is on efficient execution techniques for the bytecode interpreter.

Java bytecode is the ISA for the Java VM. It has numerous instructions for simple operations, such as loading and storing local variables and performing arithmetic operations, similar to the ISA for real processors. There are some instructions that are closely tied to the design of the Java language and Java VM. For example, there are separate instructions for object creation, access to a field in an object and an element of an array, and method invocation. It also has instructions to acquire and release a monitor and to throw an exception.

A Java bytecode instruction consists of one byte of opcode and optionally one or more bytes of operands, as shown in Figure 2.1. The Java bytecode instruction set is designed to reduce the code size. It has a stack architecture. Some operations have multiple opcodes for short and long forms to reduce the average code size. For example, bytecode to load a local variable of the integer type has five opcodes. Four opcodes for loading each of the first four local variables have a one-byte form, while the other opcode for loading other local variables has a two-byte form. The lengths of frequently appearing instructions are from one to three bytes, and the average length of Java methods is usually less than two bytes.

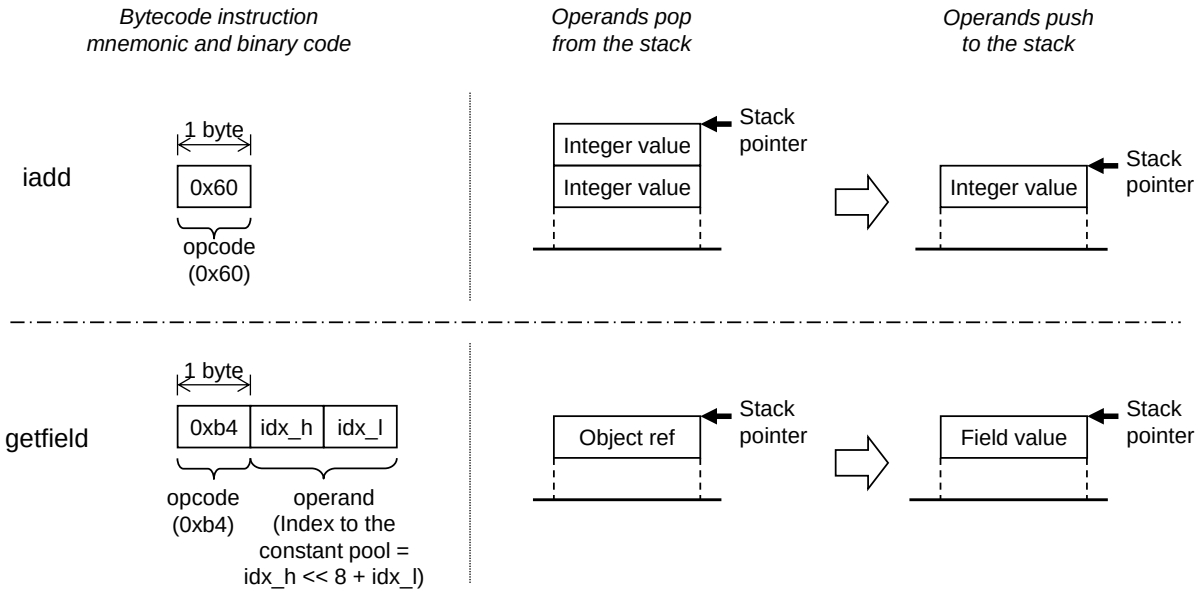


Figure 2.1. Example of Java bytecode instructions

Figure 2.1 shows the binary format and the operation of the operand stack of two bytecode instructions `iadd` and `getfield`. The left column of Figure 2.1 shows the binary code, and the middle and right columns show how each instruction gets operands from the operand stack and puts the result back on the stack.

`iadd` is an example of a simple Java bytecode instruction. It adds two integer values in the operand stack. Its binary code has a one-byte form, which consists of only the opcode. `iadd` pops two integer values from the operand stack, adds them, and pushes the result back to the stack.

`getfield` is an example of a Java bytecode instruction that is closely tied to Java language. It pops an object reference from the operand stack, loads a field variable of the object, and pushes it back to the stack. Its binary code has a three-byte form, which consists of one byte of opcode and two bytes of operands that form an index to the constant pool entry for the field definition. The Java VM resolves the constant pool entry if it is not resolved.

2.4 Java Just-in-Time (JIT) Compiler

The dynamic compiler for the Java VM is often called the Java Just-in-Time (JIT) compiler. The Java JIT compiler is an important component for improving the performance of a Java program. On the

other hand, it has issues in that it consumes a large amount of resources and is difficult to debug, as we will discuss in Chapter 5 and Chapter 6.

The JIT compiler converts a Java bytecode sequence, usually in units of Java methods, into optimized native code and stores it in memory for reuse later. In a mixed-mode Java VM, it selects a method to be compiled on the basis of the runtime profile, including the invocation count of the method, and usually compiles it asynchronously in a separate thread. Java JIT compilers use internal data of the Java VM for applying adaptive optimizations; these data are utilized for generating faster code customized for the current execution environment.

As mentioned above, the drawbacks of the JIT compiler are the increase in resource consumption and the difficulty debugging it. The compiler uses large amounts of CPU power and working memory. Although it is invoked intermittently and most of its working memory is released at the end of each compilation, its large peak usage defines the minimum requirement of the underlying system. The debugging issue is explained in Chapter 3 and Chapter 6.

2.4.1 Optimization Techniques for Java JIT Compilers

Many optimization techniques have been developed for dynamic compilers. They efficiently reduce the execution time of the generated code by using the information that is available only at runtime. The feedback-directed optimization [Arnold et al. 2000; Suganuma et al. 2000; Paleczny et al. 2001] is one such technique. It utilizes the internal states of the runtime system and applies a higher level of optimization, such as specializing the generated code to the current internal states. Devirtualization using class hierarchy analysis is another example of optimizations developed for dynamic compilers. It changes a virtual method invocation into a non-virtual one when there is only one implementation of the virtual method. It allows the single implementation to be inlined into the caller methods. Ishizaki, et al. [Ishizaki et al. 2000] proposed an efficient devirtualization for Java by implementing a mechanism that nullifies the optimization when the Java VM loads a class after the JIT compilation and the devirtualization becomes invalid.

The JIT compiler also utilizes the status of the constant pool resolution. For example, if a reference to an external method has already been resolved when the JIT compiler refers to it, the external method can be inlined safely because a reference is resolved only when it is verified accessible and because a resolved reference never becomes unresolved. For object-oriented languages, method inlining is a very effective optimization because a typical method is small and it is difficult to

find opportunities for making optimizations to small methods. Without being aware of the resolution status, the JIT compiler may need to give up some optimizations if it is difficult to handle error cases in which the reference is inaccessible.

Note that a mixed-mode Java VM can fully utilize optimizations based on the resolution status because the interpreter already resolves most of the frequently accessed references when the JIT compiler checks their resolution status.

2.5 Production Java VM

This research focuses on improving the performance of production Java VMs for executing enterprise applications. The reference implementation released by Oracle Corp. is an example of a production Java VM. Some other companies provide their own implementations of the Java platform as part of their products or as separate packages. For example, IBM Corp. provides IBM J9 Java VM [Grcevski et al. 2004], while Oracle Corp. provides JRockit Java VM [Oracle 2011], which was originally developed by BEA Systems, Inc.

2.5.1 The Reference Java VM

There are two implementations of the reference Java VM. Java 1.2 and earlier versions used the classic VM. The current implementation is HotSpot VM [Sun Microsystems 2001], and it is used in Java 1.3 and later versions. OpenJDK is its open source version, and it became the reference Java VM from Java 7. The HotSpot VM improves performance by using a mixed-mode VM and a JIT compiler with adaptive optimization.

HotSpot VM has two JIT compilers, a client compiler [Kotzmann et al. 2008] and server compiler [Paleczny et al. 2001], and it selects the appropriate one on the basis of the size of the underlying machine and command line option. The client compiler is intended for interactive desktop applications, and it is optimized for reducing the response time and start-up time. The server compiler is for server applications and is optimized for improving scalability and throughput in the steady state.

2.5.2 IBM Development Kit for Java

IBM Corp. uses its own implementation of the Java VM in its products. The Java VM is also available on the Internet for some platforms as a IBM development kit for Java. This research uses the IBM implementation of Java VM as an example of a production Java VM since it was the only production Java VM whose source code we were able to access when we started.

There are two implementations of the IBM Java VM. The older implementation is based on Sun Microsystems' classic VM with IBM's own improvements. This implementation was used until Java 1.4.2. Although the original classic VM was used up to Java 1.1, IBM maintained their Java VM until Java 1.4.2 by making it support new language features added to Java 1.2 and later versions, as well as by improving its performance and reliability. The JIT compiler was independently developed by IBM. Mixed-mode bytecode execution was implemented in the release for Java 1.2.2 and later.

For Java 5 and later, IBM switched from the Java VM and JIT compiler to another implementation called the IBM J9 Java VM [Grcevski et al. 2004] and TR JIT compiler [Stepanian et al. 2005]. It supports a wider variety of platforms, including 64-bit servers and embedded devices.

Both of the IBM Java VMs are mixed-mode VMs. The J9 Java VM also uses a JIT compiler with multiple-level optimization. A frequently executed method is first compiled at a lower optimization level. When a sampling profiler detects that the method is frequently executed, it is compiled at a higher optimization level. If the method is so frequently executed as to be worth being optimized even more, it is compiled again at an even higher level. This technique improves the overall performance of enterprise applications because only a small part is very frequently executed and most of the rest is executed frequently enough for JIT to compile, but not to optimize at higher levels. However, this implementation makes the JIT compiler very difficult to debug.

In this research, we used either of these two IBM Java VMs depending on the version of the Java language used to evaluate our techniques. The differences should not affect the effectiveness of our research because its findings are applicable to any implementation of the Java VM.

3. Background

This chapter describes the background on the three areas explained in Section 1.5.

3.1 Bytecode Interpreter

This section describes the structure of a naive interpreter, the bottleneck caused by it, and a common technique for alleviating the bottleneck and implementing a high-performance interpreter.

3.1.1 Structure of a Naive Interpreter

A bytecode interpreter repeatedly reads an opcode of a bytecode and calls *the bytecode handler* corresponding to the opcode. A bytecode handler is a piece of code prepared for every opcode, and it performs operations of the corresponding opcode. The size of the bytecode handler depends on the operations defined for each opcode. For a bytecode instruction set that has many primitive operations, like Java bytecode, most handlers take less than ten cycles to execute.

A naive interpreter uses *the decode loop*. The decode loop is a small piece of code that repeatedly reads bytecode instructions and calls the appropriate bytecode handler for processing the bytecode. Thus, the decode loop and bytecode handers are executed one after another.

Because of this structure, the naive interpreter results in frequent executions of indirect branches. It executes indirect branches twice for each bytecode instruction when the decode loop calls the bytecode handler and when the hander returns to the decode loop. Indirect branches cause pipeline stalls, and the stall cycles can be larger in number than the execution cycles of a simple handler. Although the branch prediction mechanism may be able to reduce the stall cycles when returning from the handlers, it cannot reduce those when calling them because it is impossible for the branch prediction hardware to predict the next bytecode instruction. Thus, frequent executions of indirect branches constitutes a large overhead.

3.1.2 Threaded Code Interpreter

Implementing a high-performance interpreter requires one to reduce the overhead of indirect branches for calling bytecode handlers. Threaded code [Bell 1973] is the most common technique for this purpose, and most high-performance bytecode interpreters use it.

Threaded code eliminates the indirect branches for returning to the decode loop by inlining it into every bytecode handler. The inlined decode loop directly jumps to the next handler at the end of each handler, and thus, the resulting interpreter jumps around the bytecode handlers.

For modern processors like superscalar processors, threaded code is an effective way of exploiting instruction-level parallelism (ILP) and improves performance. Because the decode loop has no dependency on the main path of the handler, the inlined decode loop can be executed in parallel with the main path, and thus, it can be executed on any free execution units that are available when the main path has low ILP or stalls while waiting for data from memory.

The inlined decode loop can calculate the address of the next handler in parallel with the execution of the handler, and thus, the address calculation could finish earlier than the execution of the handler. In that case, the instruction fetch unit of the processor could start fetching the instructions of the next handler and hide part of the stall cycles.

The original threaded code [Bell 1973] is sometimes called *direct threaded code* to distinguish it from variations. The variations can be categorized according to whether or not an interpreter requires conversion from a bytecode sequence into an intermediate representation. The direct threaded code and *indirect threaded code* [Dewar 1975] convert a bytecode sequence into a list of addresses of bytecode handlers or addresses of operands to bytecode handlers before executing the bytecode. *Token threading* [Ritter and Walker 1980], on the other hand, avoids this conversion step by calculating the address of bytecode handlers from the opcode values, such as by looking up a translation table.

3.2 Memory Management

This section describes the physical memory management technologies of OSes and APIs for memory management.

3.2.1 Physical Memory Management by OSes

Physical memory management is an important function of OSes, and modern multi-process OSes use virtual memory and paging for efficiency. An OS provides isolated address space (*virtual address space* or process's *virtual memory*) for each running process, divides the virtual address spaces and physical memory into areas of the same size (*pages*), and maps used pages in the virtual address

spaces (*virtual pages*) to physical memory pages (*page frames*). A typical page size is 4 Kbytes, although the size depends on the implementation of the OS and the type of processor.

The OS dynamically changes the mapping between virtual pages and page frames on the basis of how often virtual pages are used, so it can try to map all frequently used virtual pages to the physical memory. When the total amount of used virtual memory is larger than the amount of physical memory, the OS stores some of the virtual pages to disk (*swap out*). The swapped out page will be restored from disk to memory (*swap in*) when the virtual page is accessed again.

3.2.1.1 Overhead of Frequent Swapping

The set of virtual pages frequently accessed during a short period is called *the working set* and its size is called *the working set size*. The virtual memory utilizes temporal locality for efficient physical memory management because the working set is usually much smaller than the total amount of virtual memory accessed by the program.

An OS performs frequent swap-ins and swap-outs if the size of the working set exceeds the size of physical memory. In such a situation, some pages in the working set need to be swapped out, even though they will soon be swapped in, because the pages are part of the working set and accessed frequently.

This situation is called *thrashing*, and it significantly degrades performance. An access to a swapped-out page takes orders of magnitude longer than an access to a page in physical memory because the process needs to wait until the accessed page is swapped in from disk and disk is much slower than memory. Such slow memory accesses occur frequently in thrashing situations.

3.2.1.2 Page Replacement Policy

It is desirable to swap out pages that are unlikely to be used in the near future. An issue in selecting pages to swap out is that the OS cannot know when each page will be needed in the future. Thus, the OS guesses it be following some rule or heuristic. This selection strategy is called *a page replacement policy*. Several policies have been proposed, such as FIFO, second chance, and the least recently used (LRU) policy [Silberschatz 2012]. The most common policy in modern OSes is LRU or pseudo-LRU, which simplifies the decision of least recentness.

The LRU policy swaps out the page that has experienced the longest time since its last access. This policy assumes that the pages unused for a long time are less likely to be accessed again anytime

soon and is further based on the assumption that the memory access patterns of programs have temporal locality.

Although the above assumption works well in most cases, it does not work well for some programs whose memory access patterns do not have temporal locality. For example, if a program repeatedly scans an array that is larger than the amount of physical memory, an array element once loaded into memory will be swapped out before it is accessed again. Processing Streaming data is another example where the above assumption fails, because each byte of streaming data is handled only once.

3.2.2 Memory Management APIs

Programs allocate and deallocate working memory areas during their execution. Programming environments provide APIs to allocate memory areas whose size is specified by the programs. This functionality is called *dynamic memory management*. Since Java VMs are usually developed in C or C++, efficient memory management is needed for good performance and low resource usage. In the programming model of C, the API allocates the requested memory area from a large memory area called *the heap* by dividing it into smaller areas. Unix operating systems also provide system calls for explicit allocation and deallocation of physical pages.

3.2.2.1 The Malloc-free API

The system library provides `malloc()` for allocating an arbitrary amount of memory and `free()` for deallocating it. Since an OS manages virtual memory in units of a page, `malloc()` allocates a large memory block and divides it into areas of the requested size. The remaining part of the memory block is pooled in a library for handling subsequent requests. Memory areas deallocated by `free()` are usually kept in a free list managed by the library and reused for handling requests in the future.

A widely used implementation is that of the GNU C library. For example, Linux systems use it as their system library. The GNU version of the implementation uses PTmalloc [Gloger 2006], which is based on Doug Lea's `malloc` [Lea 2000]. They call a large memory block an *arena*. Multiple arenas can be allocated to avoid thread contention. The library creates a new arena when multiple threads try to access the same arena.

A free list is a common way to reduce the number of memory management system calls, which incur a large overhead. However, there is a possibility that the memory areas in the free lists will

remain unused for a long time. In such case, the memory areas in the free lists would be wasted memory.

3.2.2.2 System Calls for Manipulating Pages

Modern OSes provide APIs for manipulating the mapping between virtual memory pages and page frames. For example, UNIX-like OSes provide three system calls: `mmap()`, `munmap()`, and `madvise()`. A program can use `mmap()` for asking the kernel to map page frames or memory-mapped file regions to virtual address space, and it can use `munmap()` to unmap the memory.

System libraries for dynamic memory allocation use these APIs internally. User programs can also use these APIs for making an efficient implementation that is aware of physical memory usage. Effective use of the API may avoid thrashing.

The `madvise()` system call gives hints to the kernel on how the programs will use a range of virtual memory. The available hints depend on the OS. In Linux, the `MADV_DONTNEED` hint is useful for reducing physical memory usage. It indicates that a range of virtual memory is not needed any more and allows the kernel to discard the content of the memory. The kernel can simply unmap the pages for an address range without swapping out the data to disk. Subsequent accesses to the released pages will succeed, but the pages will be zeroed out. Other OSes (such as FreeBSD) require the `MADV_FREE` and `MADV_DONTNEED` options for this effect. These system calls can avoid thrashing by explicitly unmapping unused memory to reduce the size of the working set.

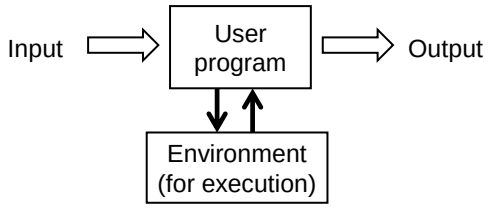
3.3 Execution Models of User Programs and Static Compilers

This section explains the execution models of general user programs and static compilers when they are executed normally and when they are debugged. We also explain the diagnostic output of the compilers because it is very useful for debugging.

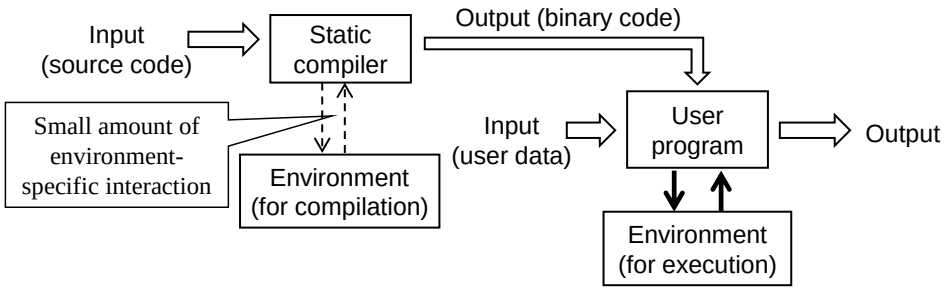
This research focuses on debugging deterministic programs because compilers usually operate deterministically. We assume the operation is executed in a single thread and its output depends only on its input and the interaction with the environment where it runs.

3.3.1 Execution Models of a General User Program and a Static Compiler

Figure 3.1 shows the execution models of a general program and a static compiler. A general user program produces its output based on the input and interaction with its environment, as shown in Figure 3.1 (a).



(a) Execution model of a user program



(b) Execution model of a static compiler

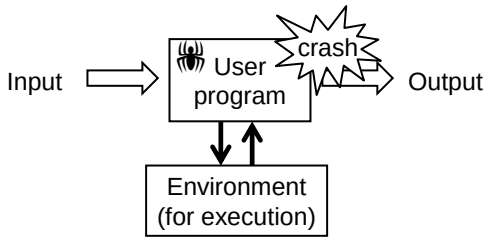
Figure 3.1. Execution model of user program and static compiler.

Figure 3.1 (b) describes the model of the static compiler. The difference between its output and a general user program is that the output of the compiler is binary code and it will be executed later using its own input and environment. The environment for executing the generated binary code is not necessarily the same as that for the compilation. Static compilers avoid using information that is specific to the environment because the generated binary code must be able to run in any environment.

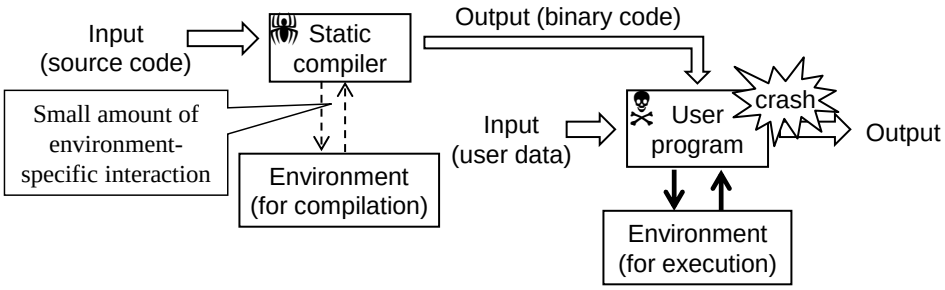
3.3.2 Execution Models for Debugging a General User Program and a Static Compiler

When a user program crashes, as shown in Figure 3.2 (a), the developers of the program will analyze the problem and fix it if it is in the user program. A user program can also crash because a problem in a compiler resulted in problematic binary code, as shown in Figure 3.2 (b). In this case, the developer of the user program will ask the compiler developers to debug the compiler. In either case, they need to reproduce the problem in order to analyze the program by checking the output or by tracing the execution with a debugger.

For debugging a general user program, the developer needs to reproduce the input to the program and the environment in which the program was executed. Debugging is usually easy when the



(a) Execution model when a user program crashes



(b) Execution model when a compiled user program crashes because a problematic static compiler generates the wrong binary code

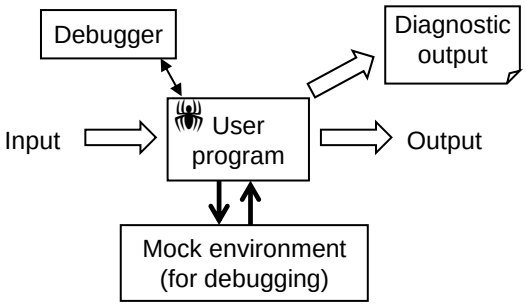
Figure 3.2. Execution models of (a) when a user programs crashes because of its own problem and (b) when a problem in the static compiler causes the user program to crash

problem is detected in the development phase, but it may be more difficult when the problem occurs in the customers' environment. A common approach to reproducing the problem when the developer cannot access the environment where the problem occurred is to create a mock environment that is sufficient for running a specific operation scenario, as shown in Figure 3.3 (a).

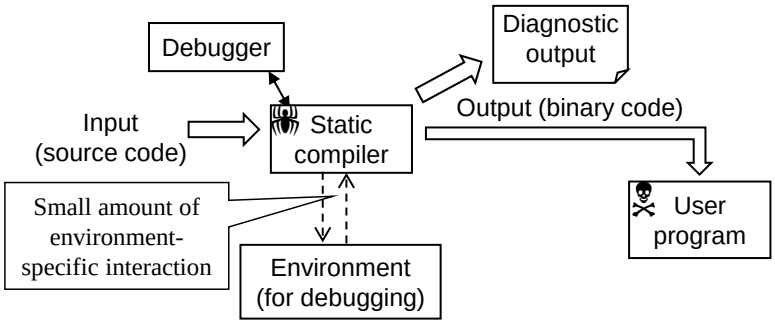
For debugging a static compiler, as shown in Figure 3.3 (b), compiler developers do not need to access the actual environment where the developer of the user program ran the compiler. Since a static compiler obtains only a limited amount of environment-specific information, its operations can be reproduced in a similar environment to the actual one in which the problem occurred.

3.3.3 Diagnostic Log File of Compilers

The compiler's diagnostic output plays a very important role in debugging because it contains all the details of what the compiler performed, including what optimizations were applied, and how each optimization transformed the code. This helps developers to analyze bugs in compilers, and they can often recognize a bug without re-executing the compiler in a debugger.



(a) Execution model for debugging a user program



(b) Execution model for debugging a static compiler

Figure 3.3. Execution models when a user program or a static compiler is re-executed to generate diagnostic output.

The diagnostic output usually contains the source code to be compiled, snapshots of intermediate representations after each compilation step, and generated machine instructions. Without such diagnostic output, it would be very difficult to associate each generated machine instruction with the source code. It may also contain detailed information on the decisions of each step, such as why a method was inlined, how hot paths were selected, and why a register was selected for spilling. Note that a Java program may be loaded from a remote machine through a network or from a dynamically generated memory block. Dynamic bytecode generation is becoming popular in the recent versions of Java[1], and there may be no corresponding external files for some compiled methods. The contents of those classes will never be available after they are unloaded. Thus, the diagnostic log file is indispensable for debugging Java JIT compilers. A log file contains very detailed information, so it is often large.

[1] For example, HotSpot Java VM 1.4 and later versions internally generate Dynamic Proxy Classes (http://java.sun.com/j2se/1.4.2/docs/guide/reflection/proxy.html) for improving the performance of reflective method calls and field accesses.

3.4 Java Programs Used for Measurements

This section briefly explains the Java programs used in the experiments of this research.

SPEC JVM98

A set of benchmarks to measure various performance characteristics of a single Java VM. It is created by Standard Performance Evaluation Corporation (SPEC). A detailed description is available at http://www.spec.org/osg/jvm98/.

SPECjbb2000

A benchmark to measure the performance of server-side Java by emulating the business processes of a wholesale company. It is created by SPEC, and further information is available at http://www.spec.org/osg/jbb2000/.

DaCapo

A benchmark suite intended for use by the programming language, memory management, and computer architecture communities. It consists of a set of open source real world applications, and the code is freely available at http://www.dacapobench.org/. Its characteristics are described in a paper [Blackburn et al. 2006].

Java Grande Forum Benchmark Suite

A benchmark suite created by Java Grand Forum (http://www.javagrande.org/). It had been freely available at http://www.epcc.ed.ac.uk/javagrande/, but at present only contact information is available at http://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite. Its characteristics are described in a paper [Mathew et al. 1999].

jBYTEmark

A set of micro benchmarks ported from the C version of BYTEmark. BYTEmark is the BYTE Magazine's benchmark for measuring the performance of CPUs and memory systems. It was a popular benchmark in the Java 1.1 era because improving the performance of emulation engines was an important research topic at the time.

Apache DayTrader benchmark sample

Apache DayTrader is a benchmark application to measure the performance of a Web application

server by emulating a stock trading system. It is built on Java EE technology. Further information is available at http://geronimo.apache.org/GMOxDOC20/daytrader.html

WebSphere Application Server

WebSphere Application Server is an IBM's production Java EE server, which supports the full specification of Java EE. Further information is available at http://www.ibm.com/software/webservers/appserv/was/

jigsaw

Jigsaw is the official World Wide Web Consortium (W3C) HTTP server written in Java. Further information and the program binary are available at http://www.w3.org/Jigsaw/

XML parser

XML Parser for Java was available at http://www.alphaworks.ibm.com/tech/xml4j, and we measured the time to parse sample XML files. The code was donated to Apache, but Xerces is not a direct successor of this code because it was re-implemented by Apache. XML4J is also included in the IBM SDK Java Technology Edition.

4. Fast Bytecode Interpreters

The performance of the interpreter is important even for high-performance virtual machines that employ just-in-time compiler technology, because there are advantages in delaying the start of compilation and in reducing the number of the target methods to be compiled. This chapter[1] describes three novel techniques of our Java bytecode interpreter, *write-through top-of-stack caching (WT)*, *position-based handler customization (PHC)*, and *position-based speculative decoding (PSD)*, which ameliorate these problems for the PowerPC processors. We show how each technique contributes to improving the overall performance of the interpreter for major Java benchmark programs on an IBM POWER3 processor.

4.1 Overview

Interpreters play an important role in many languages, and their performance is particularly critical for the popular language Java. The performance of the interpreter is important even for high performance Java VMs that employ Just-In-Time (JIT) compiler technology [Suganuma et al. 2000; Paleczny et al. 2001; Kotzmann et al. 2008] to boost the steady-state performance. Compiling every method causes an intolerably slow start up time and a huge memory footprint for the target application [Suganuma et al. 2001]. In order to improve the overall performance, there is more pressure to delay the start of compilation and reduce the number of the target methods to compile. Suganuma et al reported their JIT compiler compiles only about 20% of all the methods and interprets the rest [Suganuma et al. 2001].

Many techniques have been proposed to improve the performance of various interpreters [Bell 1973; Ertl 1995; Ertl and Gregg 2001; Gregg and Ertl 2001; Hoogerbrugge et al. 1999; Hoogerbrugge and Augusteijn 2000; Romer 1996], but none of them has fully addressed the issues of minimizing redundant memory accesses and the overhead of indirect branches inherently caused by interpreters on superscalar processors. These issues are especially serious for Java because its intermediate form, called *bytecode*, is typically one or a few bytes long and the execution routine corresponding to each bytecode, called *the bytecode handler (or the handler)*, has a short critical path length due to the low-level, stack-based semantics of Java bytecode.

[1] This chapter is based on my work presented at ASPLOS 2002 [Ogata et al. 2002].

Stack caching [Ertl 1995], caching a few stack operands in registers, is a common technique to improve the performance of an interpreter for a stack-based virtual machine language. For the Java bytecode interpreter, the number of stack operands frequently used is two or less, and thus caching top two operands on the operand stack using a state machine, called *the dynamic stack caching of two operands (DS)*, is the most popular approach. However, DS requires a state machine and three versions of customized bytecode handlers corresponding to each state. Our solution is to cache only the single operand at the top of the stack with 'the write through policy,' called *write-through top-of-stack caching (WT)*, and thus to use multiple versions of the bytecode handlers for a more effective optimization technique.

Frequent load operations for fetching a bytecode from memory are the major performance bottleneck for superscalar processors, because the hardware always performs word access for loading data from the data cache and misaligned load operations are expensive. Our solution is to prefetch a sequence of bytecodes in two registers, called *bytecode prefetch registers (BPRs)*, so that we can minimize the redundant memory accesses for fetching bytecodes. In addition, we create four versions of customized bytecode handlers, corresponding to each of the four possible bytecode positions in the BPRs, for every bytecode. We call this *position-based handler customization (PHC)*. Using a state machine, we can locate the current byte position without any runtime check in each handler.

Pipeline stalls caused by indirect branches for *dispatching* the bytecode handler are another performance bottleneck for superscalar processors, because the hardware branch prediction often misses when the interpreter branches to the next bytecode handler. Our solution is to reserve four registers, called *address pool registers (APRs)*, and to speculatively load into the APRs up to three candidate addresses of the handler for the bytecode following the next one. We call this *position-based speculative decoding (PSD)*. At the beginning of the currently running bytecode handler, we can immediately resolve the address of the next handler and specify it for the instruction fetch unit (IFU) as the branch target address to eliminate the pipeline stall cycles caused by the indirect branch.

In this chapter, we describe three novel techniques of our Java bytecode interpreter developed for the PowerPC processors to ameliorate these problems, and we show how each technique contributes to the overall performance of the interpreter for major Java benchmark programs on an IBM POWER3 processor [O'Connel and White 2000]. First, we add WT to the base interpreter. This improves the average (maximum) performance by 5.6% (11.8%) over the base interpreter. We also show that DS improves the average (maximum) performance by only 1.4% (3.5%) in comparison to

WT. On top of WT, we then add the support of PHC, which improves the average (maximum) performance by 25.7% (50.7%) over the base interpreter. Finally, we add the support of PSD, which cumulatively improves the average (maximum) performance by 30.3% (56.4%) over the base interpreter.

Among three techniques, PHC is the most effective one. We show that the main source of memory accesses is due to bytecode fetches and that PHC successfully eliminates the majority of them, while it keeps the instruction cache (I-cache) miss ratios small.

4.2 Our Implementation

This section describes our Java bytecode interpreter. We briefly describe our base interpreter first, and then provide the details on the three optimization techniques implemented on the base interpreter.

4.2.1 Base Interpreter

A naive interpreter repeats a sequence of *fetching*, *decoding*, *dispatching*, and *executing* bytecodes under the control of a small piece of the code, called *the decode loop*. *Fetching* is to load the target bytecode from memory to the register. *Decoding* is to resolve (find) the address of the corresponding bytecode handler for the target bytecode. *Dispatching* is to branch to the resolved address of the target bytecode handler. *Executing* is to process the target bytecode handler. A threaded code [Bell 1973] interpreter inlines the decode loop into every bytecode handler, which directly jumps to the handler for the next bytecode at the end.

Our base interpreter (Figure 4.1 (b)) is a threaded code (or a token threaded code [Ritter and Walker 1980]) interpreter implemented in an assembly language. The gray arrow in Figure 4.1 (b) shows the control flow to execute the first four bytecodes in a sample bytecode sequence of Figure 4.1 (a). Each handler fetches the operand of the current bytecode and the opcode of the next bytecode together in a single load instruction. Then it decodes the next opcode and executes the current bytecode using the operand. It avoids using a decode table by laying out the handlers at each 64-byte boundary in the order of the opcode, so that the handler can decode a bytecode by shifting the opcode to the left by six bits and adding the result to the base address of the handlers.

4.2.2 Stack Caching

An interpreter for a stack-based virtual machine language such as Java bytecode frequently accesses the operand stack in memory to push and pop its stack operands. Stack caching [Ertl 1995], caching a

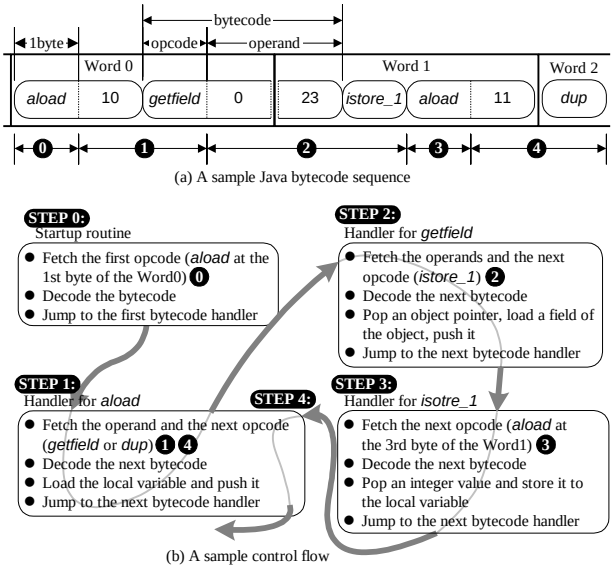


Figure 4.1. The base interpreter

few stack operands in registers, is one of the most common techniques to improve the performance of the interpreter. For the Java bytecode interpreter, the number of stack operands frequently used is two or less, and thus caching top two operands on the operand stack using a state machine, called *the dynamic stack caching of two operands (DS)*, is the most popular approach. However, caching a single operand at the top of the operand stack with the 'write-through policy', called *write-through top-of-stack caching (WT)*, is simple enough to get most of the performance improvement for the PowerPC processors, while unlike the DS interpreter we can avoid maintaining a state machine and preparing multiple versions of the bytecode handlers for stack caching.

The WT interpreter always holds in a pre-defined register, called *the top-of-stack (or stack) register*, the operand at the top of the operand stack. When a bytecode pushes the new operand, the bytecode handler keeps it in the stack register and also stores it into the operand stack in memory based on the write-through (store through) policy. When a bytecode pops the stack operand and the value in the stack register becomes invalid, the bytecode handler loads the next operand on the stack into the stack register. When the operand stack becomes empty, it loads the dummy operand into the stack register.

Although the WT interpreter seems to generate more store operations than the one with the write-back (store-in) policy, the superscalar processor can usually hide the latency when it jumps to the next bytecode handler. For the write-through policy, the bytecode handler can issue a load before a

store, while for the write-back policy it must issue a load after a store to reflect the stack register and thus delay the load. We chose the write-through policy to avoid this delay.

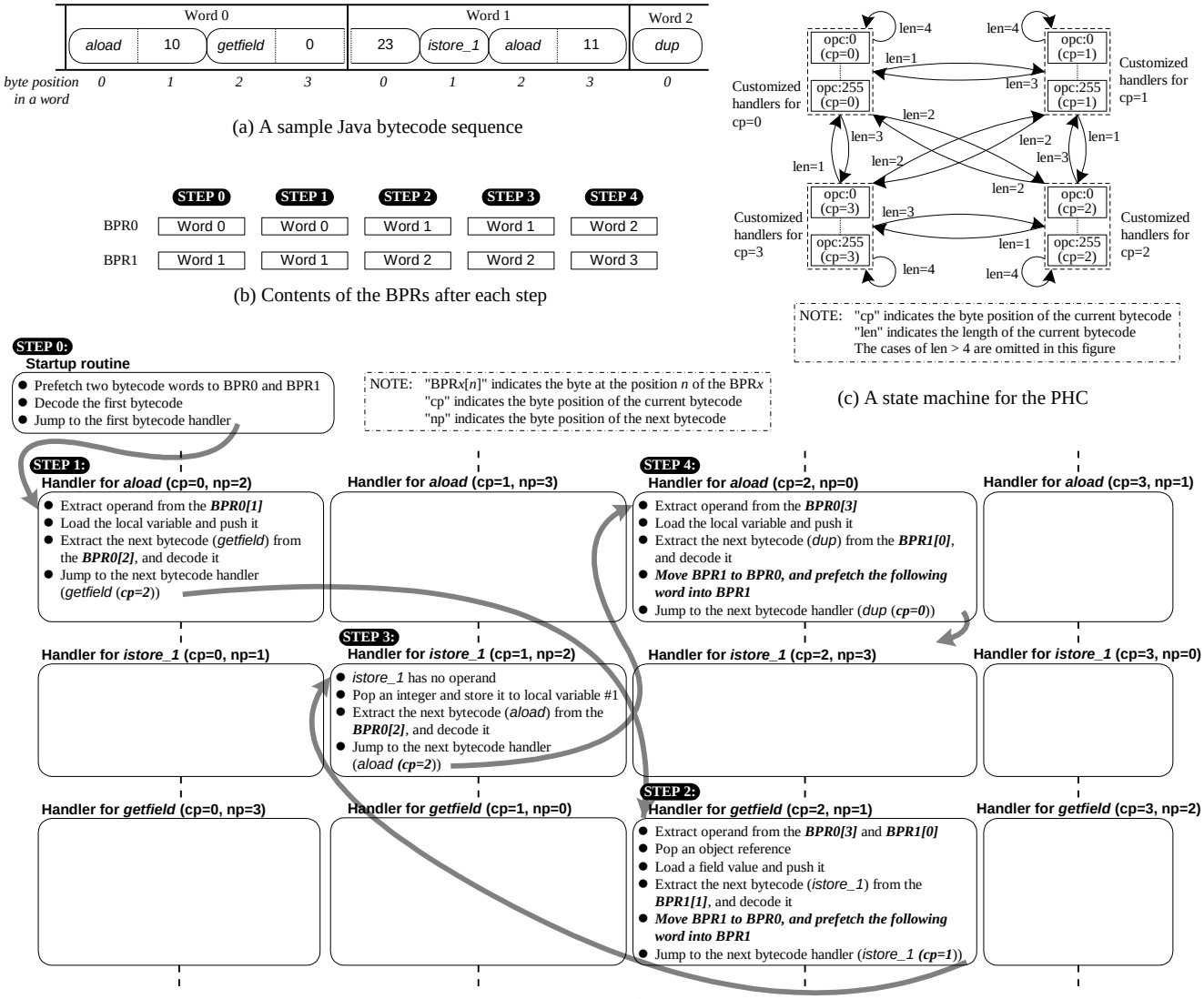
4.2.3 Position-based Handler Customization

The base interpreter issues a load instruction to fetch a new bytecode at the beginning of every handler. This operation is redundant if the target bytecode is short, because a load instruction fetches a full word from memory regardless of the size of the operand. It also requires two memory loads when the target bytecode crosses a word boundary.

One solution to reduce these redundant memory loads is to prefetch a sequence of bytecodes in registers, called *bytecode prefetch registers (BPRs)*, using word-aligned loads. In fact, reserving two registers is sufficient for Java bytecode. Since the beginning of the target bytecode is not always aligned on the word boundary, the bytecode handler must check the position of the bytecode in a word at runtime to locate its operand and the opcode of the next bytecode. Checking the position is expensive for the bytecode handler since the pipeline stalls on the conditional branches (up to five cycles on the PowerPC) could double the critical path length (typically 4 to 7 cycles) of the bytecode handler. An alternative [Hoogerbrugge and Augusteijn 2000] would be to shift the sequence of bytecodes in registers every time a handler accesses a new bytecode so that the position of the opcode in the register is always aligned to the left most position. However, this requires the runtime costs to perform shift operations. Furthermore, in order to refill the next word, it still requires runtime checking to see if the bytecode crosses a word boundary.

Our solution is to create four versions of customized bytecode handlers for every bytecode, each of which corresponds to the byte position of the currently executing bytecode, called *the current position (cp)*. We call this *position-based handler customization (PHC)*. Each handler can compute the byte position of its operands and the byte position of the next bytecode, called *the next position (np)*, from the cp and the offset from the cp without any runtime checking. Similarly, it can keep track of the cp using a state machine as shown in Figure 4.2 (c). By customizing the handlers, we can eliminate the following operations:

- Extraction of the lower two bits of the bytecode pointer to determine the cp (with an *andi* instruction, which takes one cycle),
- Determination of which of two BPRs is holding the operands of the executing bytecode (with a comparison and a branch, which take up to five cycles),



(a) A sample Java bytecode sequence

(b) Contents of the BPRs after each step

(c) A state machine for the PHC

(d) A sample control flow

Figure 4.2. The WT+PHC interpreter

- Computation of the shift count to extract the operands of the executing bytecode and the opcode of the next bytecode from the BPR (with a few arithmetic computations, which take a few cycles),
- Checking to see if shifting the BPRs and loading the following word are necessary (with a comparison and a branch, which take up to five cycles).

Figure 4.2 (d) shows how the customized bytecode handlers for the 32-bit PowerPC processor operate. The gray arrow in Figure 4.2 (d) shows the control flow to execute the first four bytecodes in

a sample bytecode sequence of Figure 4.2 (a). Here the customized operations unique to each version of the handler are written in ***bold-italic***. The handlers maintain two BPRs as shown in Figure 4.2 (b).

The disadvantage of PHC is the code expansion of the bytecode handlers, which decreases I-cache utilization. For our interpreter, the total size of the customized bytecode handlers is 64 Kbytes, but the working set of the frequently executed bytecode handlers is much less than half of that [Radhakrishnan 2001]. In our simple simulation using the bytecode trace, the I-cache miss ratio is expected to be less than 1.3% with a 16 Kbytes 4-way set associative cache or a 32 Kbytes 2-way set associative cache on the computation-intensive programs. We show the empirical results on the POWER3 processor in Section 4.5.

Since PHC is a kind of software pipelining techniques using prefetching, it has to reload the BPRs after every branch in bytecode, such as the execution of a conditional or unconditional branch bytecode, or method invocation and return, or handling Java exceptions. If the frequency of reloading the BPRs is high, the overhead for the restart will degrade the performance. However, since the frequency of the taken branch is usually around every 5 to 7 bytecodes [Radhakrishnan 2001], our technique can improve the performance of most of the programs as shown in Section 4.4.

4.2.4 Position-based Speculative Decoding

Each bytecode handler for the base interpreter executes an indirect branch at the end in order to jump to the handler of the next bytecode. The target address of this indirect branch is difficult to predict even with the branch prediction hardware such as the branch target buffer (BTB), because it is decided by the opcode of the next bytecode. The same opcode may not always follow the given opcode. In fact, in our experiment, 100 different opcodes followed *iload*, and the most frequent one was counted only for 6.8% of the total execution counts of *iload* instructions.

Speculative decoding based on the pipelined interpreter [Hoogerbrugge et al. 1999; Hoogerbrugge and Augusteijn 2000] solves this problem by overlapping the four stages of fetching, decoding, dispatching, and executing the bytecode, based on a software pipelining technique. It speculatively decodes a few candidates for the opcode of the bytecode following the next one, and the handler for the next bytecode can resolve and specify the branch target address for the IFU in several cycles before it executes the indirect branch. Many processors have a hardware mechanism to reduce the pipeline stall cycles on an indirect branch by specifying the branch target address for the IFU [Intel 2002; IBM 1994] in advance before the indirect branch is actually executed. For example, on

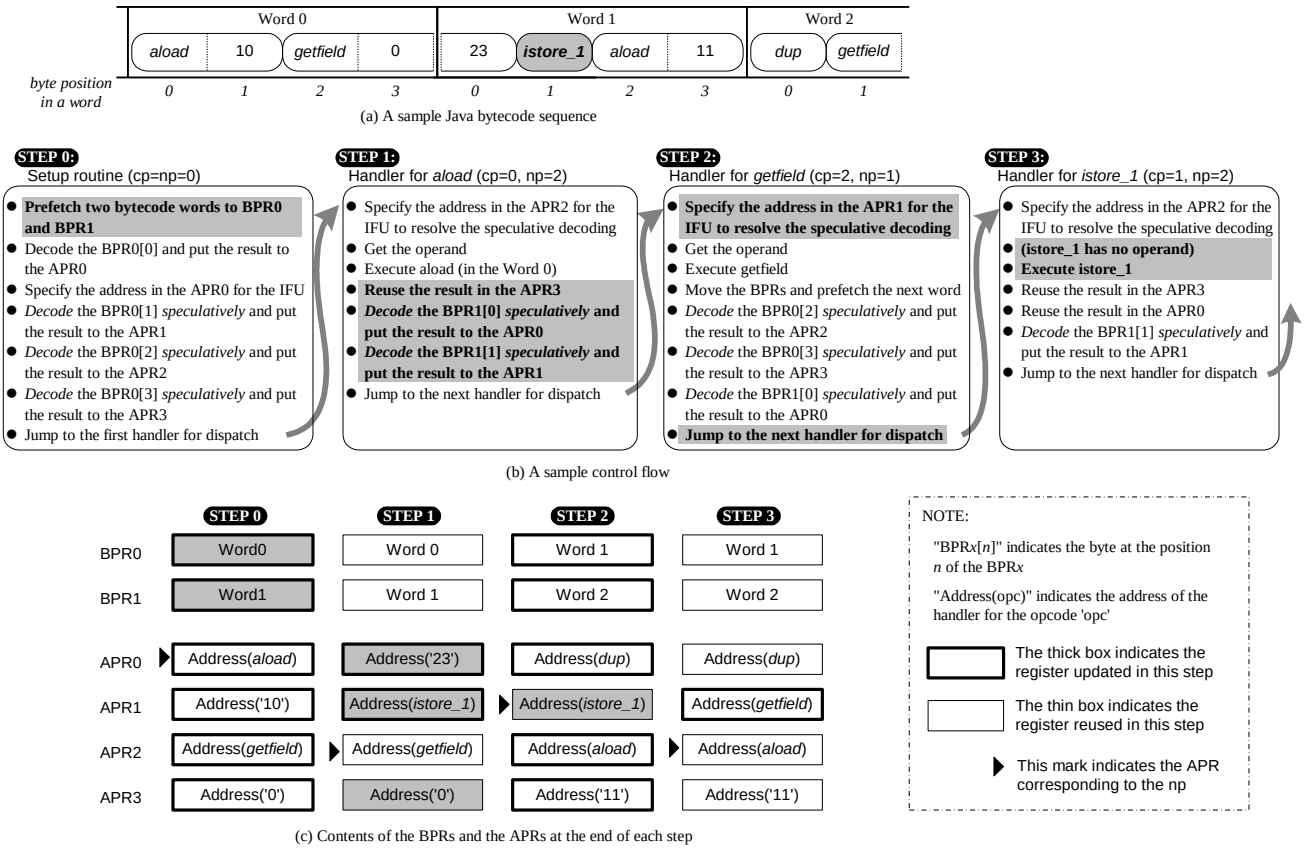


Figure 4.3. The WT+PHC+PSD interpreter

the PowerPC 604e processor [IBM and Motorola 1998], we can completely eliminate the pipeline stall cycles if we can set the branch target address to a special purpose register, either *the link register* or *the count register*, at least four cycles prior to the execution of the indirect branch. The Intel IA-64 processor [Intel 2002] has a similar mechanism.

The optimal number of candidates depends on the bytecode instruction set, and it is three for Java bytecode. The addresses decoded speculatively are held in registers for reuse to minimize the redundant decoding of multiple candidates for the following opcode. In order to avoid shifting the decoded addresses in three registers [Hoogerbrugge and Augusteijn 2000], our approach is to reserves four registers, called *address pool registers (APRs)*, to hold the decoded addresses. We call this *position-based speculative decoding (PSD)*. For a 32-bit processor, reserving four registers makes it possible to associate the byte position of the candidate for the opcode with each APR. Since our interpreter has four customized bytecode handlers corresponding to the cp, it can automatically select the valid APR based on the np.

As an example, Figure 4.3 (b) shows how the WT+PHC+PSD interpreter operates, using a sample bytecode sequence of Figure 4.3 (a). Figure 4.3 (c) shows the contents of the BPRs and the APRs at the end of each step of the execution in Figure 4.3 (b). The shaded operations, spread in several steps, all correspond to those to execute *istore_1* in a sample bytecode sequence. The setup routine (step 0) *prefetches* the bytecodes into BPR0 and BPR1 for PHC. The bytecode handler for *aload* (step 1) performs the *decode* stage, and it decodes three candidates for the opcode of *istore_1*, corresponding to the byte position 3 of BPR0 and the byte position 0 and 1 of BPR1. It decodes only the byte position 0 and 1 of BPR1, and it reuses the decoded result of byte position 3 from BPR0. The handler for *getfield* (step 2) then performs the *dispatch* stage, and it specifies the address of the next bytecode handler for the IFU at the beginning of the handler by obtaining the address from APR1. This should be sufficiently ahead in the execution cycle before the execution of the indirect branch for the next handler, and thus PSD completely eliminates the pipeline stall cycles. Finally, the handler for *istore_1* (step 3) executes the current bytecode.

4.3 Applicability

In this section, we discuss how we can apply our techniques to other architectures.

4.3.1 Architectural Aspects of PHC

The base interpreter usually needs at least six registers for the bytecode pointer, the frame pointer, the stack pointer, the local variable base pointer, and a few working registers. The WT interpreter needs one more register for the stack register. The WT+PHC interpreter needs to reserve two more general purpose registers for the BPRs. Therefore, PHC is less effective when it is applied to an architecture with a small number of general purpose registers, such as the Intel x86 processors.

For 64-bit processors, all the registers are eight bytes long and thus prefetching eight bytes is more efficient for PHC. In this case, eight versions of the customized bytecode handlers should be prepared for every bytecode, but refilling the BPRs should be delayed as much as possible because there is a higher chance of encountering a branch in the prefetched bytecodes. Considering both the code expansion caused by preparing eight versions of the bytecode handlers and the need for binary code compatibility with 32-bit processors, it is desirable to use 32-bit versions of the interpreters on the 64-bit processors.

4.3.2 Architectural Aspects of PSD

The WT+PHC+PSC interpreter needs to reserve four more general purpose registers for the APRs. For 64-bit processors, four registers are sufficient for the APRs. When eight versions of the customized bytecode handlers are prepared for every opcode, a pair of bytecode handlers for the same opcode corresponding to the same lowest two bits of the cp can access the same APRs.

The target processor architecture needs to have a hardware mechanism to specify the branch target address for the IFU in order to reduce pipeline stall cycles caused by indirect branches. For example, we can use a special instruction, such as *mtlr* or *mtctr* [IBM 1994] on the PowerPC or *mov br=gr,tag* [Intel 2002] on the Intel Itanium processor, or a delayed branch [Hoogerbrugge and Augusteijn 2000]. Smaller latency to specify the branch target address leads to greater improvement because it reduces the minimum critical path length of the handler when using this technique. Since a special instruction is required in the bytecode handlers, this technique needs an interpreter written in an assembly language or a compiler modified to generate the special instruction.

The target processor architecture needs to exploit instruction-level parallelism (ILP), available in superscalar and VLIW processors, to overlap the four stages: prefetching bytecodes into the BPRs, the speculative decoding of subsequent bytecodes for the bytecode following the next one, specifying the address of the handler of the next bytecode for the IFU, and the execution of the current bytecode.

For the PowerPC processors, two integer units and two of the other execution units (i.e., a load-store unit, a branch unit, and a multi-cycle integer unit) should be available in each cycle to decode the candidates without runtime overhead. For a processor without sufficient hardware resources, it may be necessary to decrease the number of candidates for the opcode.

4.4 Performance Evaluation

We evaluated our techniques using the programs described in Table 4.1. Table 4.2 describes the execution environment. The POWER3 processor [O'Connel and White 2000] is one of the implementations of the 64-bit PowerPC architecture. In order to isolate the effectiveness of each of our techniques, we modified the interpreter of the IBM Developer Kit for AIX, Java Technology Edition[1], as described in Table 4.3. The product version of the Java VM implements the combination of WT, PHC, and PSD. This virtual machine runs in 32-bit mode even on the 64-bit processor in

[1] IBM Developer Kit, Java Technology Edition is available at http://www.ibm.com/developerworks/java/jdk/

Table 4.1. Evaluated programs

Program	Description
SPECjvm	The elapsed time of the benchmarks in SPECjvm98. The result labeled "Geom. Mean" is the geometric mean of the SPECjvm98 results. The "controls" parameter was changed to 10.
jBYTEmark	The jBYTEmark index.
JavaGrande	The elapsed time for the Java Grande Forum Benchmark Suite [Mathew et al. 1999]. This is a benchmark suite consisting of three sets of benchmarks, and we used Section 3.
SPECjbb	The overall score of the SPECjbb-2000 benchmark.
XMLparser	The elapsed time to parse an XML file using the XMLparser for Java.

Table 4.2. Execution environment

CPU	POWER3, 400 MHz
I-Cache	32 Kbytes, 128-way set associative, 128 bytes / line, 2 banks
Memory	768 Mbytes
OS	AIX 4.3.3
Java VM	IBM SDK Java edition 1.3.0

Table 4.3. Evaluated interpreters

Interpreter	Description
Base	The base interpreter.
WT	The base interpreter with WT. This interpreter caches only integer operands.
DS	The base interpreter with DS. This interpreter has three customized handlers for the cache states: caching none, caching top-of-stack, and caching two operands. This interpreter caches only integer operands.
WT+PHC	The base interpreter with WT and PHC.
WT+PHC+PSD	The base interpreter with WT, PHC, and PSD. This interpreter is equivalent to the product version except for the arrangement of the bytecode handlers.

order to be compatible with the 32-bit operating environment, and the interpreter is implemented for 32-bit processors. The JIT compiler is disabled in all of the evaluations.

Figure 4.4 shows the relative performance over the base interpreter for each program. Taller bars indicate better performance. Table 4.4 shows the summary of the average (maximum) performance improvement over each interpreter on the left-most column for all the programs.

4.4.1 Improvement by WT and DS

The average (maximum) performance improvement by the WT interpreter is 5.6% (11.8%) over the base interpreter. The average (maximum) difference in improvement between the WT and DS interpreters is 1.4% (3.5%). The small gain by the DS interpreter proved that WT is sufficient for out-of-order processors with short-latency memory loads, such as the PowerPC.

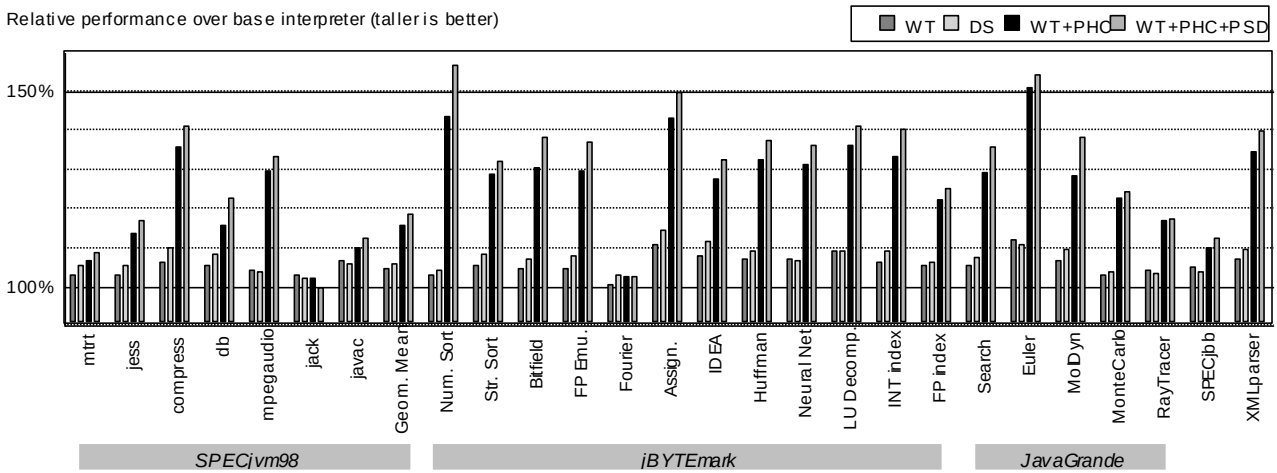


Figure 4.4. Relative performance over the base interpreter

Table 4.4. The average (maximum) performance improvement over each interpreter on the left-most column

	WT	DS	WT+PHC	WT+PHC+PSD
Base	5.6% (11.8%)	7.2% (14.3%)	25.7% (50.7%)	30.3% (56.4%)
WT		1.4% (3.5%)	19.0% (39.2%)	23.3% (51.8%)
WT+PHC				3.4% (9.1%)

We can think of two reasons for the small gain. First, most of frequently executed bytecodes do not use more than one stack operand. In fact, the average dynamic execution count of those bytecodes that use more than one stack operand is 21%. Second, reducing the bytecode execution path by more aggressive stack caching may not be beneficial when the inlined decode loop takes longer than that. As an example, Figure 4.5 shows the execution cycles for three different bytecode handlers for the iaload bytecode of the base, WT, and DS interpreters, respectively, on the POWER3 processor. All of these interpreters check for Java exceptions using trap instructions in the same manner as our JIT compiler does [Ishizaki et al. 1999], and all rely on the fact that the memory address 0 is readable under AIX. The WT interpreter can save one execution cycle by using stack caching, while the DS interpreter cannot reduce the number of execution cycles beyond the WT interpreter. This is because the inlined decode loop (shaded operations in Figure 4.5) dominates the critical path of the handler for the DS interpreter.

The average (maximum) improvement by the WT+PHC interpreter is 25.7% (50.7%) over the base interpreter. Although the DS+PHC interpreter may seem to achieve more improvement than the WT+PHC interpreter does, it is impractical if we consider the code size expansion. Since the DS

```
# The bytecode handler for iaload of the base interpreter
#cycle-1
   FetchNextBytecode      # Fetch the opcode of the next bytecode
   lwz    r4,0(sp)        # Load array index
#cycle-2
   lwz    r3,4(sp)        # Load array object reference
#cycle-3
   DecodeBytecode         # Decode the opcode of the next bytecode
   slwi   r5,r4,2         # Compute byte offset in the array
#cycle-4
   SpecifyTargetAddress   # Specify the address of the next bytecode handler
   lwz    r6,ArraySizeOff(r3)  # Load array length from the object header
   twnei r3,0             # Check for NullPointerException
   addi   r3,r3,ObjHdrSize # Compute the address of the array
#cycle-5 : stall
#cycle-6
   twlge r4,r6            # Check for ArrayIndexOutOfBoundsException
   lwzx  r3,r3,r5         # Load the array element
#cycle-7 : stall
#cycle-8
   stwu   r3,4(sp)        # Store the result and move stack pointer
   MoveBytecodePC         # Advance the bytecode pointer for the next bytecode
   Dispatch               # Jump to the next handler
```

```
# The bytecode handler for iaload with of the WT interpreter
#    tos : the register to cache top-of-stack
#cycle-1
   FetchNextBytecode      # Fetch the opcode of the next bytecode
   lwz    r3,4(sp)        # Load array object reference
#cycle-2
   slwi   r5,tos,2        # Compute byte offset in the array
#cycle-3
   DecodeBytecode         # Decode the opcode of the next bytecode
   lwz    r6,ArraySizeOff(r3)  # Load array length from the object header
   twnei r3,0             # Check for NullPointerException
   addi   r3,r3,ObjHdrSize # Compute the address of the array
#cycle-4
   SpecifyTargetAddress   # Specify the address of the next bytecode handler
#cycle-5
   twlge tos,r6           # Check for ArrayIndexOutOfBoundsException
   lwzx  tos,r3,r5        # Load the array element
#cycle-6 : stall
#cycle-7
   stwu   tos,4(sp)       # Store the result and move stack pointer
   MoveBytecodePC         # Advance the bytecode pointer for the next bytecode
   Dispatch               # Jump to the next handler
```

```
# The bytecode handler for iaload with of the DS interpreter
#    tos : the register to cache top-of-stack
#    nos : the register to cache next-of-stack
#cycle-1
   FetchNextBytecode      # Fetch the opcode of the next bytecode
   lwz    r6,ArraySizeOff(nos)  # Load array length from the object header
   slwi   r5,tos,2        # Compute byte offset in the array
   twnei nos,0            # Check for NullPointerException
#cycle-2
   addi   nos,nos,ObjHdrSize # Compute the address of the array
#cycle-3
   DecodeBytecode         # Decode the opcode of the next bytecode
   twlge tos,r6           # Check for ArrayIndexOutOfBoundsException
   lwzx  tos,nos,r5       # Load the array element
#cycle-4
   SpecifyTargetAddress   # Specify the address of the next bytecode handler
#cycle-5 : stall
#cycle-6 : stall
#cycle-7
   MoveBytecodePC         # Advance the bytecode pointer for the next bytecode
   Dispatch               # Jump to the next handler
```

Figure 4.5. The bytecode handlers for iaload bytecode

interpreter needs to prepare three customized versions of every bytecode handler, the DS+PHC interpreter would need twelve versions of each handler. This leads to a total size of 192 Kbytes of the handlers, the working set of which may not fit into the 32 Kbytes I-cache of the POWER3 processor. Therefore, considering the code size expansion, we believe the WT+PHC interpreter is more effective in improving the overall performance of the interpreter.

4.4.2 Improvement by PHC

The average (maximum) performance improvement by the WT+PHC interpreter is 19.0% (39.2%) over the WT interpreter. It improved the performance of the interpreter in many of these programs by more than 10%. These results show the effectiveness of PHC.

PHC is more effective when the bytecode handlers reload the BPRs less frequently. It is effective for the programs, such as “Num. Sort” and “FP Emu.”, which execute short bytecodes frequently. It is also effective for the programs, such as “mpegaudio”, Neural Net”, “LU Decomp.”, and “Euler”, which do not frequently branch. On the other hand, it is less effective for the programs, such as “jack”, which throw Java exceptions frequently or those, such as “Fourier”, which invoke native

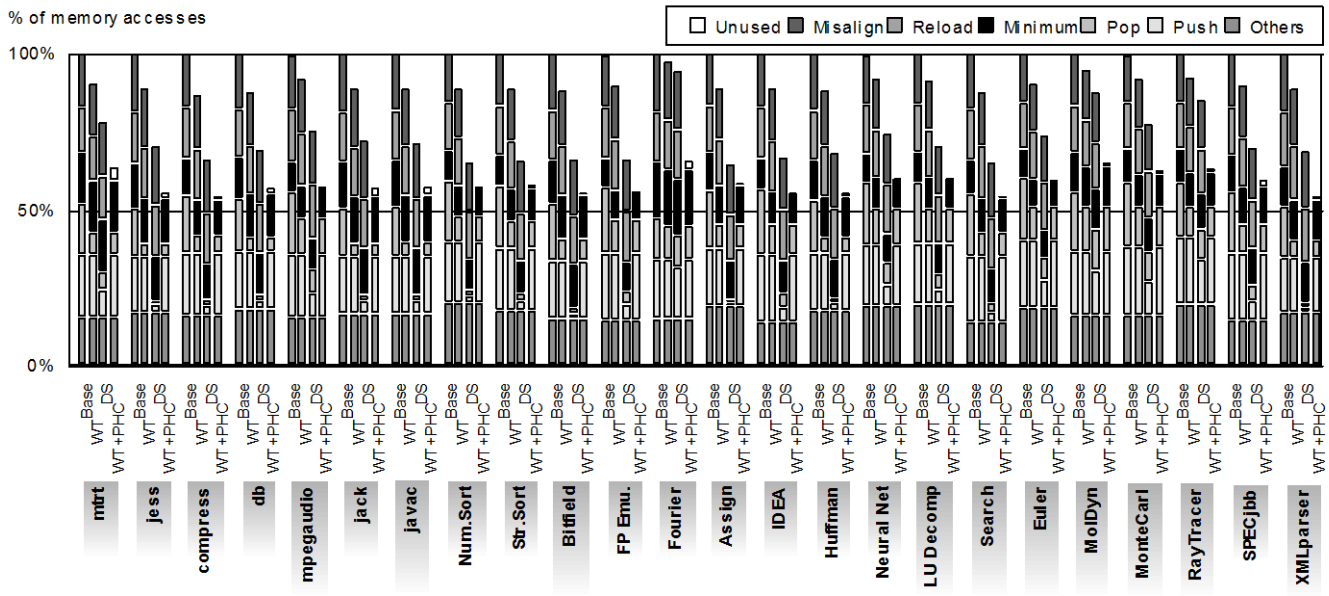


Figure 4.6. The breakdown of memory accesses (the total number of memory accesses by the base interpreter = 100%)

Table 4.5. The categories of memory accesses

Category	Description	Access type
Unique to some interpreters		
Misalign	The total number of memory accesses requested for the following word due to a misaligned load.	Bytecode fetch made by the base, WT, and DS interpreters
Reload	The total number of memory accesses requested redundantly for the word already loaded earlier.	
Unused	The total number of memory accesses whose results are unused for execution because of branches.	Bytecode fetch made by the WT+PHC(+PSD) interpreter
Common across all the interpreters		
Minimum	The total number of memory accesses requested for the minimum number of words required for fetching each bytecode.	Bytecode fetch
Pop	The total number of memory accesses for popping stack operands.	Load for a stack operand
Push	The total number of memory accesses for pushing stack operands.	Store for a stack operand
Others	The total number of memory accesses to other memory areas, such as local variables, instance variables, and class variables.	Load and store for stack frames and objects

methods frequently. In these programs, reducing the critical path length of the bytecode handlers does not improve much of the performance because the bytecode execution is small in the total execution time.

4.4.3 Improvement by PSD

The average (maximum) performance improvement by the WT+PHC+PSD interpreter is 3.4% (9.1%) over the WT+PHC interpreter. It led to performance improvement for many of the programs. These results show the effectiveness of PSD.

Table 4.6. The average (maximum) percentage of memory accesses of each category over the total number of memory accesses by the base interpreter

	Base	WT	DS	WT+PHC(+PSD)
Bytecode fetch	45.0% (54.2%)			**13.6% (22.5%)**
Pop	18.3% (21.0%)	**8.5% (14.5%)**	**4.7% (13.2%)**	**8.5% (14.5%)**
Push	19.8% (22.0%)		**5.9% (16.1%)**	19.8% (22.0%)
Others	16.9% (20.1%)			

PSD is more effective for the programs, such as “Num. Sort”, and “MolDyn”, where the inlined decode loops dominate the critical path length of the handlers. On the other hand, it is less effective for the programs, such as “jack”, which throw Java exceptions frequently. Frequent decoding of new candidate opcodes into APRs caused by the Java exceptions led to the poor performance.

4.5 Analysis of Memory Access

In this section, we show how many memory accesses each optimization reduces and how small it keeps the I-cache miss ratios. We used the same programs and interpreters as used for the performance evaluation.

4.5.1 Reduction of Memory Accesses

Figure 4.6 shows the breakdown of the memory accesses of each category normalized by the total number of memory accesses made by the base interpreter for each program. Table 4.5 summarizes all the categories of memory accesses, some of which are unique to some interpreters and others of which are common across all the interpreters. Table 4.6 shows the summary of the average (maximum) percentage of memory accesses of each category over the total number of memory accesses made by the base interpreter for all the programs. The memory accesses reduced by each interpreter are written in **bold**.

We do not show the results of the WT+PHC+PSD interpreter because they are the same as those of the WT+PHC interpreter. This is because both interpreters request the same number of memory accesses to load the BPRs, and they refill the next word into BPR1 when they execute the last bytecode in BPR0. They require no memory access for decoding bytecodes in the APRs.

The average (maximum) reduction of memory accesses for “Pop” by the WT interpreter is 53.6% (73.5%) over the base interpreter, while the WT interpreter does not reduce any memory accesses for “Push”. The average (maximum) reduction of memory accesses for “Pop” and “Push” by the DS interpreters is 74.2% (94.5%) and 71.1% (94.4%) over the base interpreter, respectively.

Since the DS interpreter uses the write-back policy, it reduces memory accesses for “Push” by avoiding the stores of intermediate results that are used by the next bytecode. Because it caches up to two operands and because it does not load a dummy operand when the stack becomes empty, it reduces more memory accesses for “Pop” than the WT interpreter. However, reducing memory accesses for stack operands does not always help improve much of the performance, as discussed in Section 4.4.1. The DS interpreter reduces more than 15% of memory accesses over the WT interpreter on most of these programs, while it improves the average (maximum) performance only by 1.4% (3.5%) over the WT interpreter.

The average (maximum) reduction of memory accesses for fetching bytecode by the WT+PHC interpreter is 70.2% (77.4%) over the WT interpreter. It completely eliminates both “Reload” and “Misalign”, while it adds “Unused”. Owing to PHC, the WT+PHC interpreter consistently reduces memory accesses even for the programs, such as “Fourier” and “MolDyn”, where stack caching is less effective. This is because these programs frequently use stack operands of floating-point type, which are not cached by the current implementation. Caching both integer and floating-point operands by the DS interpreter seems to be impractical because it would need thirteen versions of each handler (to cache all combinations of the types of two stack operands).

4.5.2 Increase in I-Cache Miss Ratios

Figures 4.7 and 4.8 show how small these optimization techniques keep the I-cache miss ratios. These miss ratios are equal to the number of I-cache misses per the number of instructions completed in the processor. We counted the numbers of I-cache misses and completed instructions using the hardware performance monitor [1] of the POWER3 processor. The average (maximum) miss ratio by the WT+PHC interpreter is still 0.13% (1.00%), while the average (maximum) increase in the relative miss ratio by the same interpreter is 1.8 (6.1) times over the base interpreter. The average (maximum)

[1] HPM Tool Kit was available at http://www.alphaworks.ibm.com/tech/hpmtoolkit/, but it has graduated from alphaWorks. The URL of the project page is http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2765, and the user's guide is available at http://researcher.watson.ibm.com/researcher/files/us-hfwen/HPM_ug.pdf

Figure 4.7. The I-cache miss ratios

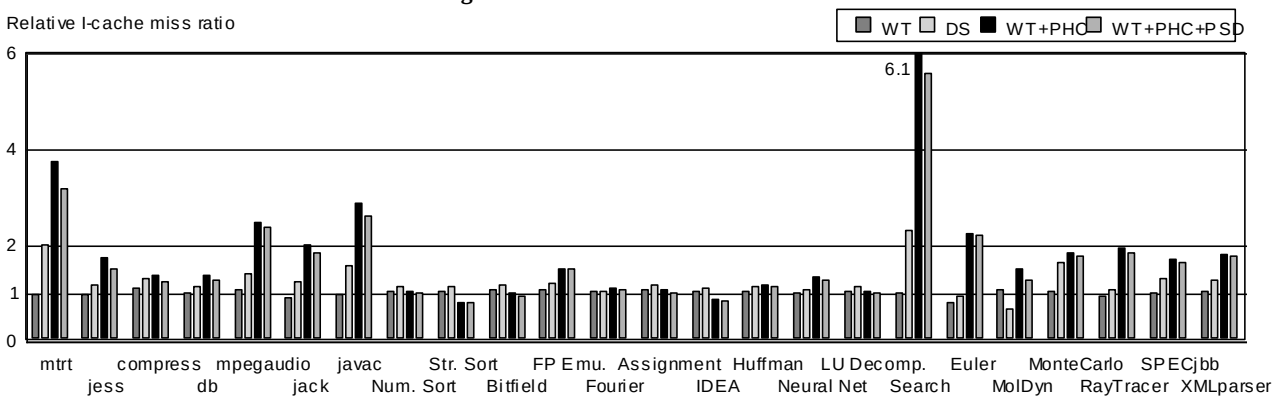


Figure 4.8. The relative I-cache miss ratios over the base interpreter

miss ratio by the WT+PHC+PSD interpreter is still 0.12% (0.92%), while the average (maximum) increase in the relative miss ratio by the same interpreter is 1.7 (5.6) times over the base interpreter. These increased cache misses led to the performance degradation of “jack”.

Since the DS interpreter also customizes the handlers, it slightly increased the miss ratios. The average (maximum) miss ratio is 0.10% (0.62%), while the average (maximum) increase in the relative miss ratio is 1.3 (2.3) times over the base interpreter. We suspect that the increased cache misses caused the DS interpreter to be less effective for “jack” and “javac” than the WT interpreter.

4.6 Related Work

We summarize related work on improving interpreter performance from three viewpoints: memory access for operand stack operations, bytecode handler dispatching, and a new trend of using trace-based compiler for implementing interpreters.

4.6.1 Memory access for stack operations

Stack caching [Ertl 1995] is a technique to reduce memory accesses for pushing and popping stack operands. In stack languages, handlers frequently access stack operands near the top of the stack. Because a stored operand is likely to be used in the next handler, caching operands near the top of the stack reduces the latency to deliver the data from memory. Dynamic stack caching [Ertl 1995] is an efficient technique to keep track of the number of cached operands using a state machine that changes states based on the stack operations. It dynamically changes the number of cached operands instead of loading data from the stack to fill in the cache of a fixed size. It uses a state machine to keep track of the number of cached operands and changes states based on the stack operations. By preparing customized bytecode handlers for each state, the technique can keep track of the states efficiently.

However, reducing the number of stores does not affect much of the performance on out-of-order processors because the processor can execute the following instructions even when a store for pushing an operand stalls. For those processors that can load a stack operand from the data cache in one or two cycles, caching only the top of the stack is sufficient for exploiting most of the benefit from stack caching. As shown in Section 4.4, our experiment shows that the performance improvement by DS is very small in most of the programs in comparison to WT, and that reducing redundant memory accesses for fetching bytecode is much more effective.

Peng et al. implemented a direct threading interpreter that caches a fixed number of stack slots [Peng et al. 2004]. Their interpreter uses the static stack caching [Ertl 1995] and puts the code for adjusting the number of cached stack entries at the beginning of each handler, so it can avoid code expansion by sharing the handlers among all stack states. It is also helpful to reduce redundant stack operations based on the stack operations of the hander. Although their interpreter still performs a few redundant memory accesses and register copying, it improved performance by 14% on average on Intel Xscale processor [Intel 2003].

4.6.2 Interpreter dispatching

A *pipelined interpreter* [Hoogerbrugge et al. 1999; Hoogerbrugge and Augusteijn 2000] with speculative decoding performs fetching, decoding, and dispatching bytecode in parallel with bytecode execution using a software pipelining technique. They implemented a pipelined interpreter with speculative decoding on a VLIW processor. Their bytecode handler speculatively decodes a few

bytes that could be candidates for the opcode of the bytecode following the next one, and then the next handler selects the right one to resolve the address of the next handler.

This approach reuses some of the decoded results to reduce the runtime overhead to decode multiple opcodes. However, their implementation needs up to two move instructions to reuse the decoded results in registers. Though extra register move instructions can be performed for free by using idle hardware resources on VLIW processors, this technique may slow down the bytecode handlers on superscalar processors because of their limited hardware resources.

Berndl et al. proposed context threading [Berndle et al. 2005] that utilizes the branch prediction facilities of modern processors. Their interpreter uses the subroutine threading, which calls bytecode handlers and returns back to the decode loop after finishing the handlers' job, to utilize the return stack. The return stack is a branch predictor that eliminates the pipeline stalls on return instructions by recording the return address when call instructions are executed. Their threaded code inlines the handler of branch bytecode, so branch instructions in bytecode can be visible to the branch prediction hardware as native branch instructions.

An interpreter with superinstruction [Ertl and Gregg 2003] eliminates the decode loop by creating a combined handler for a bytecode sequence that appears frequently. When the translator of the dynamic threaded code detects a frequently appearing sequence, it dynamically generates the combined handler of the sequence and uses the combined handler as the converted threaded code. This technique improves the interpreter performance by up to three times. They further improved their interpreter by adding static stack caching [Ertl and Gregg 2004].

This research used token threading because it directly execute bytecode sequence, and thus, there is no overhead of the conversion from bytecode to the threaded code. The direct threaded code, however, requires the conversion, and it results in increasing startup time and memory usage. For an interpreter of a mixed-mode VM, shorter startup time is more valuable than better steady state performance because we can use the JIT compiler to achieve much better steady state performance.

The amount of overhead of the indirect branches for dispatching depends on the processor architecture. When our original paper was published in 2002, many processors just had started using deeper processor pipeline to increase the clock frequency, and it resulted in higher overhead of indirect branches. However, Rohou et al. reported that the indirect branch overhead was continuously decreased in three recent microarchitectures of the Intel x86 processors [Rohou et al. 2015].

Interpreters tend to cause higher cache miss ratio and increase overhead of dispatching because it randomly jumps around the bytecode handlers, which are relatively small code blocks. McCandless and Gregg reduced cache misses by clustering bytecode handlers to improve temporal locality [McCandless and Gregg 2011]. They create an opcode profile graph and find good handler layout using algorithms for solving graph problems.

4.6.3 Trace-based compiler for implementing interpreters

Many implementations of dynamic languages use an interpreter. It is partially because the cost for developing a dynamic compiler is high, and also because the dynamic typing often makes it difficult to generate optimized code. For the runtime system of dynamic languages, it became common to write an interpreter in a VM-based programming language and use its trace-based compiler to generate the native code of the interpreter.

The trace compiler of PyPy [Bolz et al. 2009] is commonly used for this purpose. It was originally developed for compiling the Python interpreter, but it is improved to be able to generate efficient code for any interpreter written in RPython. Yermolovich et al. proposed a mechanism for the interpreter to communicate with the trace compiler, so the compiler can break traces at the points where interpreter's behavior changes [Yermolovich et al. 2009].

Trace compiler is not limited to dynamic languages. YETI [Zaleski et al. 2007] implemented an interpreter with trace compiler for Java. It gradually extends the range of a translated code block from a single bytecode to a trace of an extended basic block, which is a single-entry multiple-exit code sequence, and then it forms a complex trace by linking the traces.

4.7 Summary

We have described three major optimization techniques of our Java bytecode interpreter developed for the PowerPC processors. We showed how each technique contributes to the overall performance of the interpreter for major Java benchmark programs on an IBM POWER3 processor. We also analyzed how many memory accesses each technique reduces and how small it keeps I-cache miss ratios. Using all three techniques, we improved the average (maximum) performance by 30.3% (56.4%) over the base interpreter. Among three techniques, PHC was the most effective one. We showed that the main source of memory accesses is due to bytecode fetches and that PHC successfully eliminates the majority of them, while it keeps the I-cache miss ratios small.

5. Reduction of Java VM Memory Usage

A Java application sometimes consumes large amount of memory. This is usually because it created large number of objects in the Java heap. However, a Java application can increase its memory usage when it consumes the memory used by Java that is not in the Java heap. We call this area *non-Java memory*. For example, the non-Java memory becomes big when the Java VM (JVM) loads many classes.

This chapter[1] presents a quantitative analysis of non-Java memory. We studied the use of non-Java memory for a wide range of Java applications, including the DaCapo benchmarks and Apache DayTrader. It showed that a Java application can consume a considerable amount of non-Java memory. Our study is based on the IBM J9 Java Virtual Machine for Linux. Although some of our results may be specific to this combination, we believe that most of our observations are applicable to other platforms as well.

5.1 Overview

A Java application sometimes raises an out-of-memory exception. This is usually because it has exhausted the Java heap. A large application may use gigabytes of Java heap due to memory leaks or bloat [Mitchell and Sevitsky 2007]. With varying degrees of sophistication, many tools are available for analyzing the Java heap and for debugging the out-of-memory exceptions [Mitchell and Sevitsky 2003; Sun Microsystems 2008b].

However, a Java application can sometimes raise an out-of-memory exception because it has exhausted *'non-Java' memory*, the memory region outside the Java heap. For example, this can happen when it attempts to load too many classes into the virtual machine. Although running out of non-Java memory is rare compared to running out of the Java heap, a typical Java application actually consumes a considerable amount of non-Java memory. As we will show later, the non-Java memory usage is as large as the Java heap for more than half of the DaCapo benchmarks [Blackburn et al. 2006] when the heap sizes are twice the minimum size required for each benchmark.

A Java VM uses non-Java memory for various purposes. It holds shared libraries, the *class metadata* for the loaded Java classes, the just-in-time (JIT) compiled code for Java methods, and the

[1] This chapter is based on my work presented at OOPSLA 2010 [Ogata et al. 2010].

dynamic memory used to interact with the underlying operating system. Interestingly, modern virtual machines tend to use more and more non-Java memory. For instance, beginning with Version 1.4.0, Sun's HotSpot Virtual Machine [Sun Microsystems 2002] optimizes reflective invocations [Sun Microsystems 2002] by dynamically generating classes, which consumes non-Java memory. The same version also introduced *direct byte buffers* to improve I/O performance[1]. These buffers typically reside in non-Java memory. Out-of-memory exceptions can result from such implicit use of non-Java memory, even though Java programmers are often unaware of the specifics of such overhead. For testing, we used a micro-benchmark that repeatedly allocates and deallocates direct byte buffers with multiple threads in three implementations of the Sun HotSpot Java VM, the IBM J9 Java Virtual Machine [Grcevski et el. 2004], and the Jikes RVM[2] [Alpern et al. 2000]. We confirmed that this micro-benchmark caused out-of-memory errors (or segmentation fault crashes) in tens of seconds, even though we allocate sufficiently large Java heaps. (For the Sun HotSpot VM, we also allocated a large amount of memory reserved for direct byte buffers.)

This chapter presents a quantitative analysis of non-Java memory. To do this, we built a tool called Memory Analyzer for Redundant, Unused, and String Areas (*MARUSA*), which gathers memory statistics from both the Java VM and the operating system, using this data to visualize the non-Java memory usage. We modified the IBM J9 Java VM for Linux to efficiently gather fine-grained, JVM-level statistics.

We studied the usage of non-Java memory for a wide range of Java applications, including the DaCapo benchmarks and WebSphere Application Server running Apache DayTrader. We ran them with the modified IBM J9 Java VM under Linux. Note that the use of non-Java memory inevitably depends on both the Java VM and the operating system. Although some of our results may be specific to our JVM and Linux, we believe that most of our observations are relevant to other platforms. More specifically, in this research, we focus on the Java Standard and Enterprise Editions (Java SE and EE), rather than the Java Micro Edition (Java ME). Today the majority of Java VMs for Java SE and EE are written in C and C++, run on general-purpose operating systems, and include adaptive JIT compilers with multiple optimization levels [Grcevski et el. 2004; Sun Microsystems 2002; Oracle 2011]. We believe that our observations are also substantially relevant to these platforms.

---

[1] The direct byte buffer can be allocated using the java.nio.ByteBuffer.allocateDirect() method. (http://docs.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html)

[2] The Jikes RVM Project. Available at http://jikesrvm.org/

Contributions in this chapter are:

- We quantitatively analyzed the usage of non-Java memory for a variety of Java programs, including the DaCapo benchmarks and WebSphere Application Server running Apache DayTrader. We ran them on a modified version of IBM's production virtual machine for Linux on x86 [Intel 2009] and POWER [IBM 2007] processors, and divided non-Java memory into eight components, such as class metadata, JIT-compiled code, and JIT work areas. We measured the amount of *resident* memory these components consume over a period of time.
- We found that non-Java memory usage exceeds the Java heap for more than half of the DaCapo benchmarks when the heap size was set to be twice as large as the minimum heap size necessary to run each benchmark.
- We found that, in all of the programs studied, the JIT work area fluctuates greatly, while memory usage for the remaining components stabilizes soon. This is because the JIT compiler from time to time demands significantly more memory for its work area when compiling methods at aggressive levels of optimization.
- We observed that the behaviors of the libc memory management system (MMS), the malloc and free routines, have a strong impact on the usage of non-Java memory. Typically, a JVM-level MMS is built on top of the libc MMS, which in turn is built on top of the OS-level MMS. Even if the JVM-level MMS returns a chunk of memory to the libc MMS, this may not lead to reduce resident memory size, since the libc MMS may fail to return it to the OS-level MMS.
- We evaluated a technique to effectively manage memory by directly telling the OS-level MMS to remove memory pages even when libc MMS fails to remove it.

5.2 An Anatomy of Non-Java Memory

Figure 5.1 shows a breakdown for the non-Java memory of an enterprise Java application, WebSphere Application Server (WAS) running Apache DayTrader for 9 minutes. About 210 MB of non-Java memory was used, which is almost the same as the default WAS Java heap size, 256 MB (not shown in Figure 5.1). However, Java programmers are typically unaware of such situations.

For deeper quantitative analysis, we divided the non-Java memory into eight categories. Table 5.1 summarizes these categories and their typical data types. This section describes each of these memory areas. In the example in Figure 5.1, five categories consume most of the non-Java memory.

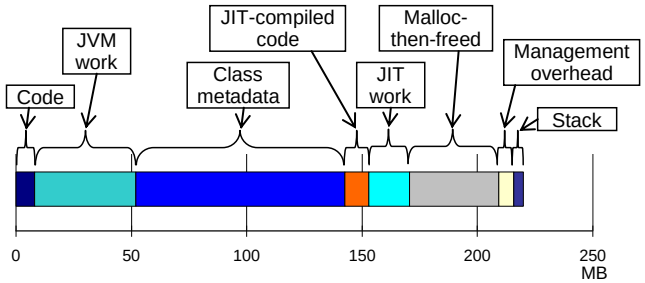


Figure 5.1. Breakdown of non-Java memory when Apache DayTrader is running on WebSphere Application Server. This is the annotated output from MARUSA, showing the resident set size but the Java heap.

Table 5.1. Categories of non-Java memory.

Category	Typical data
Code area	• Code loaded from the executable files • Shared libraries • Data areas for shared libraries
JVM work area	• Work area for the JVM • Areas allocated by Java class libraries
Class metadata	• Java classes
JIT compiled code	• Native code generated by the JIT • Runtime data for the generated code
JIT work area	• Work areas for the JIT compiler
Malloc-then-freed area	• The areas that were once allocated by malloc(), then free()ed, and still residing in memory (typically held in a free list)
Management overhead	• The unused portion of a page where only a part of a page is used, or the area used to manage an artifact, such as the malloc header
Stack	• C stack • Java stack

5.2.1 Code Area

Code area memory holds the native code from executable files and libraries, and the data loaded from shared libraries. This area does not include any of the code generated by the JIT compiler. The size of code area increases when the code or data in an executable file or a library is loaded and actually used.

5.2.2 JVM Work Area

JVM work area memory holds the data used by the JVM itself and the memory allocated by a Java class library or user-defined JNI methods. The memory used for direct byte buffers is an example of memory allocated by a Java class library. This area does not include class metadata or the other JIT-

related areas. The size of this area increases when the JVM needs more working storage or when a Java application allocates more memory through a Java class library.

5.2.3 Class Metadata

Class metadata is a memory area for the data loaded from Java class files, such as bytecode, UTF-8 literals, the constant pool, and method tables. The JVM creates metadata upon loading a Java class. While there is no explicit allocation for this in Java applications, using a class is not free, but does require some memory. This overhead memory can become significant for large applications using thousands of classes.

5.2.4 JIT Compiled Code

JIT compiled code memory area stores native code generated by the JIT compiler and the data for the generated code. The size of this area increases as the JIT compiler compiles more methods. Some JIT compilers can recompile methods to optimize them more aggressively and generate new versions of the compiled code, which usually consume even more memory. If a JIT compiler supports unloading of the generated code, the size of this area can decrease.

5.2.5 JIT Work Area

JIT work area memory contains data used by the JIT compiler, such as the intermediate representations of a method being compiled. The size of this area increases when the intermediate representation is large (perhaps as methods are inlined) or when the JIT does aggressive optimizations. The size of this area decreases when the compilation of a method is completed, though some of the data may remain in memory for inter-procedural analysis or profiling. Note that the JIT compiler can use aggressive optimizations depending on the amount of available work area memory, so the JIT compiler will function correctly even when it is unable to allocate the desired amount of work area memory.

5.2.6 Malloc-then-freed Areas

Malloc-then-freed memory areas are allocated using `malloc()` by the JVM or JIT, and then deallocated using `free()`. The malloc library typically manages such areas by holding them in a free list or returning them to the OS. If held in the free list, then the deallocated memory resides in the non-Java memory in this malloc-then-freed area. If returned to the OS, then the deallocated memory can be

removed from the process memory. Therefore, the size of this non-Java memory depends on how the standard C library (libc) and OS handle deallocated memory.

We include malloc-then-freed areas as part of the non-Java memory, since it remains in the resident memory of the process and consumes actual memory pages. This area sometimes becomes quite large, as shown in Figure 5.1. Note that this large malloc-then-freed area is not a unique problem for JVMs, but can also affect traditional C programs.

5.2.7 Management Overhead

Management overhead memory is implicitly used by OS or system libraries to manage process memory. A malloc header is an example of this kind of data. The unused parts of allocated pages are also included in this category.

5.2.8 Stack

Stack memory area is used for the Java stack and the C stack. We combined these stacks into the same category because both can be used to store the stack frames of Java methods corresponding to the implementation of the JVM. The size of this area increases when many stack frames are allocated in nested calls, when a stack frame contains many local variables, or when threads are created.

5.3 Methodology to Measure Non-Java Memory

This section describes the analysis methodology used to divide the non-Java memory into these eight categories.

5.3.1 Our Approach

The philosophical key to our memory analysis is to fully identify the usage of the resident memory of a JVM process based on these eight categories (plus the Java heap). The underlying OS manages the address ranges of a process's resident memory, while the JVM controls the actual memory usage. Thus, we need to gather memory management information at both the OS and JVM levels. We use three steps to categorize non-Java memory:

1. Gather OS-level information to enumerate all of the memory ranges owned by a JVM process and identify the attributes of each range.

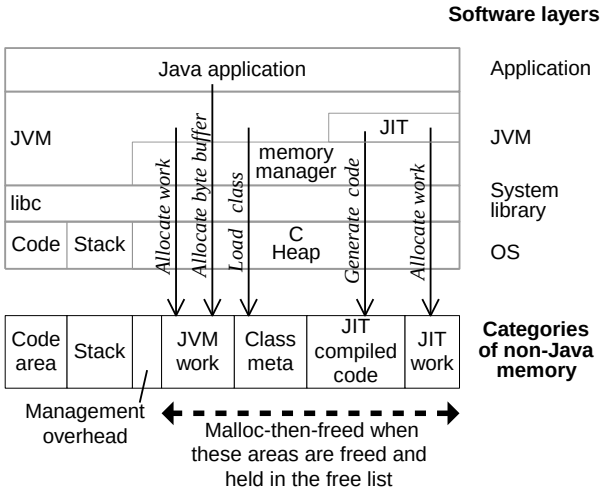


Figure 5.2. Correspondence between memory allocation paths and the eight categories of non-Java memory.

2. Gather JVM-level information to identify the use of each area based on the component that allocated it.

3. Combine these two levels of information and summarize the data using the eight categories.

Large modern programs, including JVMs, may have their own internal memory managers, which allocate chunks of memory from a pool, dividing them into smaller pieces to handle memory allocation requests from other components. Therefore, we also need to identify each component that requested memory from the internal memory manager. Tracing only the memory allocation API calls, such as `malloc()` and `free()`, is insufficient to identify the memory usage in such a program because it only captures the operations of the internal memory manager, without identifying how the pool is used by those components.

Figure 5.2 shows examples of the correspondence between the memory allocation paths and the eight categories of non-Java memory. Since a memory requestor at a higher layer has more detailed knowledge about how the memory is used, we need to gather information from all layers and combine it carefully, avoiding duplication.

For that purpose, we built a tool called MARUSA, which gathers two levels of memory information, interprets it, and then visualizes the breakdown of non-Java memory usage. Our tool can also analyze the Java heap area [Kawachiya et al. 2008], though we focus on non-Java memory in this research.

5.3.2 Gathering OS-level Memory Management Information

We first need to know the sizes and attributes of the memory blocks assigned to the JVM process. These attributes typically include access permission, mapped file flag, and the file path if the memory is mapped to a file, though the specific attributes available depend on the OS.

In this study, we focus on the resident set size of process memory, where physical memory is assigned. Therefore, we also need to gather information on which of the pages in the memory blocks of the process have physical pages.

Under Linux, MARUSA uses `maps` in the `/proc` file system to gather address ranges and their attributes. For Linux kernels version 2.6.25 or later, we can collect the physical page states using `pageinfo` in the `/proc` file system. For older kernels, we can use a kernel module included in the open source software `exmap`[1].

5.3.3 Gathering Memory Usage in Java VM

If the JVM provides detailed information about its memory usage for debugging the JVM, we can use it to categorize non-Java memory. If the information is insufficient, we need to add probes to the JVM by using plug-ins or by modifying JVM source code.

MARUSA uses a mix of these approaches. We use debugging information from the IBM J9 Java VM to get the sizes of the class metadata and the JIT compiled code, and we modified IBM JVM to gather detailed information about memory allocations and deallocations, including requests to the internal memory manager. This fine-grained data allows us to capture full information regarding memory usage of the JVM work area.

5.3.4 Computing Non-Java Memory Usage

To combine both OS-level and JVM-level information, the MARUSA analyzer uses a map structure that holds all of the gathered information for each memory byte in the JVM process. This map uses the virtual address of each byte as a key to combine the information gathered from different sources. We call this map *the memory attribute map*. For example, it can identify that a memory byte was

[1] The project page of exmap memory analysis tool is available at http://www.berthels.co.uk/exmap/ and its code has been moved to https://github.com/jbert/exmap

allocated using malloc() by the internal memory manager for loading class metadata, and that it is in a page that is allocated in physical memory.

To compute the breakdown of the non-Java memory usage, MARUSA counts the bytes with the same memory attributes. MARUSA uses a prioritized list of attributes to avoid counting any bytes twice. It first sums the bytes with the highest priority, and then sums the bytes with the second highest priority among the bytes still uncounted, and so on. We can create other views of the memory breakdown by changing the ordering of the list.

5.4 Micro-Benchmarks

This section describes the relation between the size of the non-Java memory and the operations in Java programs. Although this correspondence depends on the implementation of the Java VM, many other implementations of the Java VM should show similar trends. Actually, we measured the total resident set size of the Sun HotSpot JVM process running the same micro-benchmarks using the ps command, and confirmed that the resident set size followed the same trend as that of the IBM J9 Java VM.

We developed several micro-benchmarks to analyze non-Java memory, and evaluated them using the IBM J9 Java VM for Java 6 in Linux on x86 and POWER machines. Tables 5.2 and 5.3 describe our measurement environment.

In these measurements, we show the size of the non-Java memory where physical memory is actually allocated. Since no memory was swapped out during these measurements, this is the same as the RSS (Resident Set Size) of each JVM process after subtracting the size of its Java heap area.

5.4.1 Micro-benchmark for the Class Metadata

The first micro-benchmark shows how the size of the class metadata changes when reflective method invocation is heavily used. We created a micro-benchmark that invokes a getter and a setter for each of 6,000 fields by using a java.lang.reflect.Method object for each of them. We measured the memory usage when these 12,000 methods were invoked 10 times, 100 times, and 2,000 times.

Figure 5.3 shows the results for this micro-benchmark on x86. The class metadata area was 3.9 MB when each method was invoked 10 times and 21.8 MB when each method was invoked 100 or 2,000 times. This is because the JVM dynamically generated a method for each Method object to optimize the reflective invocations, and loaded those generated methods and their containing classes

Table 5.2. Execution environment for x86.

Hardware environment	
Machine	IBM BladeCenter LS21
CPU	Dual-core Opteron (2.4 GHz), 2 sockets
RAM size	4 GB
Software environment	
OS	SUSE Linux Enterprise Server 10.0
Kernel version	2.6.16
Java VM	IBM J9 Java VM for Java 6 (SR7), 32bit

Table 5.3. Execution environment for POWER.

Hardware environment	
Machine	IBM BladeCenter JS21
CPU	Dual-core PowerPC 970MP (2.5 GHz), 2 sockets
RAM size	8 GB
CPU and memory allocated to the tested virtual machine	
CPU	2 CPUs
Memory	2 GB
Software environment	
OS	RedHat Enterprise Linux 5.4
Kernel version	2.6.18
Java VM	IBM J9 Java VM for Java 6 (SR7), 32bit

[Sun Microsystems 2002]. For our micro-benchmark, this generated 12,002 classes for the tests with 100 and 2,000 invocations, while only two classes were generated for the test with 10 invocations. These two classes were always generated by a Java class library.

When each method was invoked 2,000 times, the size of JIT compiled code grew from 0.8 MB to 12.3 MB. This is because the methods in the generated classes were JIT compiled after they were invoked many times.

The total memory increase in the class metadata and the JIT compiled code was 29.2 MB. This extra memory consumption caused by reflective invocation is 43% of the resident set size when reflective invocation was used 2,000 times, but many programmers do not worry about such a large amount of overhead. In addition, since this memory consumption is a result of optimization by the JVM, a Java program may suddenly raise an out-of-memory error even though it has been running without problem for a while. As modern programs are becoming more dynamic and reflective method invocations are more heavily used for their flexibility, the likelihood of such errors is increasing and programmers need to monitor their use of non-Java memory.

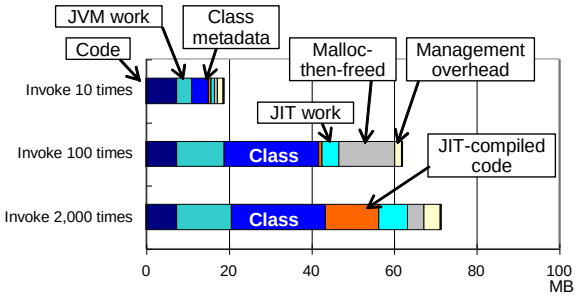


Figure 5.3. Changes in non-Java memory due to repeated reflective invocation on x86.

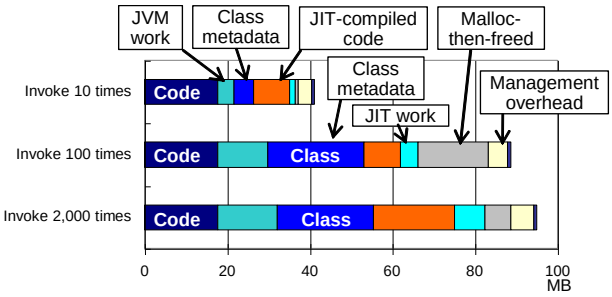


Figure 5.4. Changes in non-Java memory due to repeated reflective invocation on POWER.

Figure 5.4 shows the results for this micro-benchmark on POWER. The growth trend of the resident set size was the same as for x86. However, the code areas and the JIT-compiled code areas were notably larger.

The reason for the larger code areas was the difference in the base page size[1]. In RedHat Enterprise Linux 5 for POWER, the base page size is 64 KB. This change can improve performance by reducing the number of TLB misses [Talluri and Hill 1994], but may increase memory usage because of internal fragmentation.

The larger JIT-compiled code area was due to a difference in the implementations. Since the size when allocating a new chunk of memory for JIT-compiled code is larger in the POWER implementation, the initial size of this area is larger than for x86, and it grows in larger steps.

5.4.2 Micro-benchmark for the JVM Work and Malloc-then-freed Areas

Next we studied how the size of the JVM work area changes due to the allocations of direct byte buffers. We created a micro-benchmark that allocates and deallocates a specified number of direct byte buffers. For these measurements, the size of each byte buffer was set to 32 KB, the Java heap size was set to 8 MB, and no allocation failure GC occurred during the test runs. Figure 5.5 shows the memory usage on x86 when 10,000 direct byte buffers were created and then garbage collected by invoking `System.gc()`.

[1] Peter W. Wong and Bill Buros. A Performance Evaluation of 64KB Pages on Linux for Power Systems. https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Welcome%20to%20High%20Performance%20Computing%20%28HPC%29%20Central/page/A%20Performance%20Evaluation%20of%2064KB%20Pages%20on%20Linux%20for%20Power%20Systems

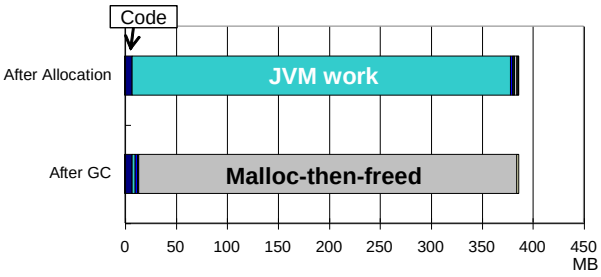


Figure 5.5. Change in non-Java memory due to allocating and freeing direct byte buffers on x86.

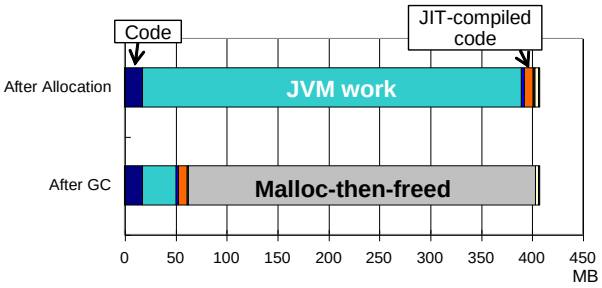


Figure 5.6. Change in non-Java memory due to allocating and freeing direct byte buffers on POWER.

Although the Java heap was as small as 8 MB, the JVM work area was 354.3 MB, and 351.7 MB in that JVM work area was used as memory for the actual buffers of the direct byte buffers. This large memory consumption could cause an unexpected out-of-memory error because it is invisible to Java programs and Java debugging tools.

The memory consumption after garbage collection was unchanged because the malloc-then-freed area increased by 350 MB, while the size of the JVM work area was reduced by 352 MB. This suggests that all of the memory for the direct byte buffers was retained in the free list, even though the Java programs and the JVM were unaware of its existence in the process memory. This memory in the malloc-then-freed area is also invisible to Java programs and other tools, and thus, it can cause problems for Java programmers and system maintainers because of unexpectedly high memory consumption.

Figure 5.6 shows the results for the same scenario on POWER. The size of the JVM work area was 355.2 MB. The memory used for the actual buffers of the direct byte buffers was exactly the same as the memory used on x86, because the Java program specified the amount of memory.

However, the JVM work area after garbage collection was 31.4 MB, while it was reduced to 2.7 MB on x86. This is because about 900 direct byte buffer objects still remain in Java heap even after garbage collection, while the direct byte buffers were completely collected on x86.

5.5 Macro-Benchmarks

This section shows our experimental results using larger programs. We evaluated WebSphere Application Server (WAS) 7.0 running Apache DayTrader and the DaCapo benchmarks. For DaCapo, we present and discuss only the results of the benchmark named `bloat`, because the other programs showed the similar trends in non-Java memory use.

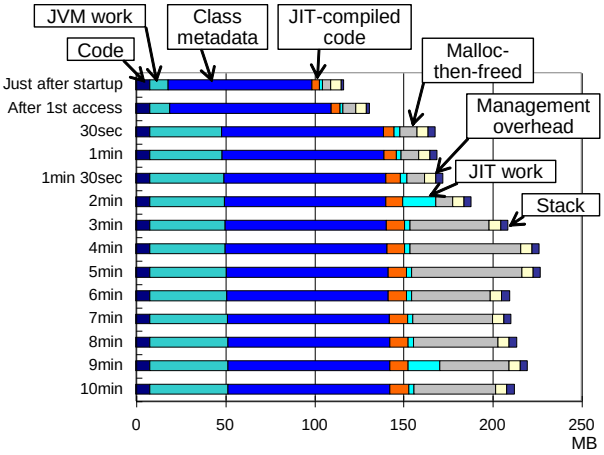


Figure 5.7. Non-Java memory for WAS 7.0 running Apache DayTrader on x86.

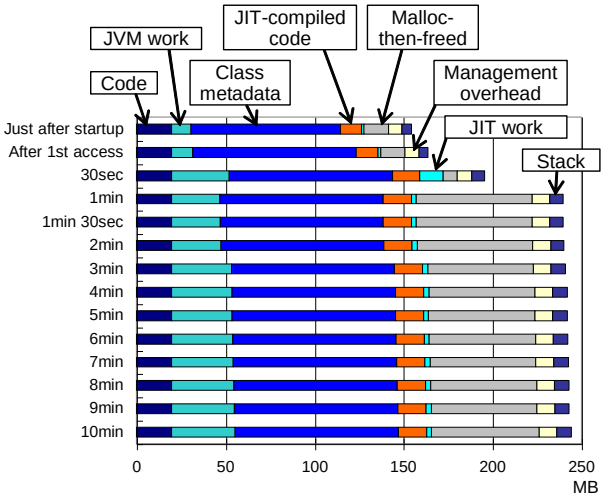


Figure 5.8. Non-Java memory for WAS 7.0 running Apache DayTrader on POWER.

The hardware environments for these measurements were the same as shown in Table 5.2 for the micro-benchmarks.

5.5.1 WAS 7.0 Running Apache DayTrader

Figure 5.7 shows how the non-Java memory use changes during the execution of Apache DayTrader in WAS 7.0 on x86. This graph shows the non-Java memory at fourteen points in a single invocation of the server: (1) just after starting the server, (2) after the first access to the scenario page of the DayTrader application, and then (3-14) at 12 times up to 10 minutes while DayTrader is accessed by a load generator using 30 threads. Note that the measurement intervals are not equal. The maximum heap size was set to 256 MB, but the Java heap area is not shown in the graph.

In this application, the class metadata was the largest memory area just after startup, and the JVM work area increased by 27.4 MB to 37.8 MB at 30 seconds. The cause of this increase was the memory for the direct byte buffers. Then the malloc-then-freed area grew, and these three areas became the major areas in the non-Java memory. The JIT work area occasionally became large, but it was small at many of the measurement points. We will discuss the JIT work and malloc-then-freed areas in Section 5.2.

Figure 5.8 shows the same scenario on POWER. For these measurements, we only used 20 threads on the load generator because this POWER machine was slower than the x86 machine, but the CPU utilization was still more than 90% during the measurements.

Table 5.4. Configurations to run DaCapo bloat for both x86 and POWER.

DaCapo configuration	
Version	2006-10-MR2
Measured benchmark	bloat
Workload size	default
Number of iteration	1
Java VM configuration	
Java heap	13 MB

The size of the JVM work area grew up to 30 seconds, but decreased at 1 minute and grew again at 3 minutes. In this execution, these fluctuations are caused by the combination of two memory allocation activities. One is the increase of the direct byte buffers, which were more numerous at 30 seconds and at 3 minutes, using 13.2 MB and 5.1 MB, respectively. The other is the allocation of temporary data structures at 30 seconds and their deallocation at 1 minute, which resulted in shrinking the JVM work area.

The stack area is also 4 MB larger than on x86. The reason is the larger base page size, which increased the unused memory in the pages allocated for the stack. In this measurement, WAS ran 155 threads. A JVM typically allocates at least one separate page as the stack for each thread, so that it can use guard pages to detect stack overflows. This means that threads whose largest stacks are small will not use memory efficiently.

5.5.2 DaCapo

We analyzed the non-Java memory use of the DaCapo benchmarks. We will only discuss the results for bloat since the other benchmarks showed similar trends in their non-Java memory use. Table 5.4 describes the configurations of the DaCapo benchmark and the Java heap size. We measured the memory use at 20 points in a single execution of the benchmark to see how the non-Java memory changed during the execution of a single iteration of the benchmark.

Figure 5.9 shows how the size of the non-Java memory changes during the execution of bloat on x86. The vertical axis is the percentage of the total object allocations in the benchmark. For example, the first bar shows the memory usage when the JVM had allocated objects whose cumulative size was 5% of the total allocation in bloat, which was about 990 MB. We call this point the 5% *allocation point*. As shown in Table 5.4, the maximum heap size was set to 13 MB for this benchmark. The non-Java memory consumption was much larger than the Java memory.

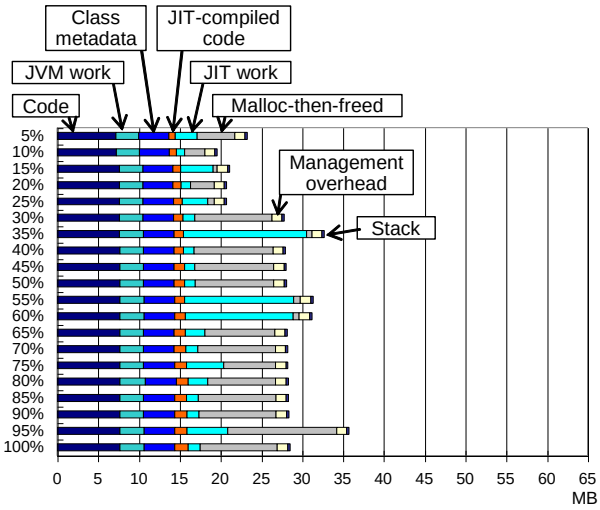


Figure 5.9. Results of DaCapo bloat on x86.

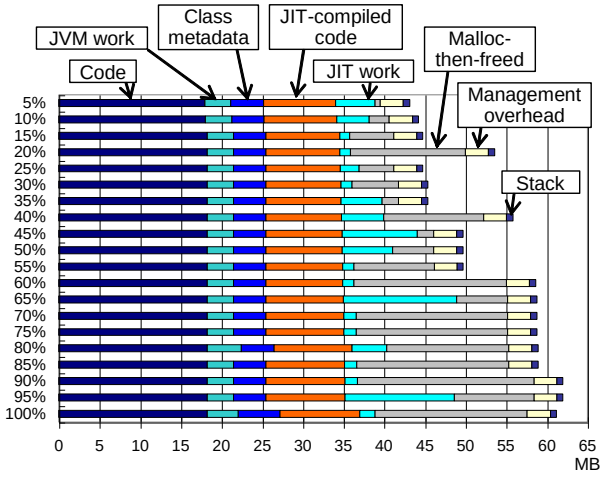


Figure 5.10. Results of DaCapo bloat on POWER.

The sizes of the JIT work areas and the malloc-then-freed areas varied widely within a single execution. Note that our measurement approach captures snapshots of the memory as it changes continuously during the execution of the program. Therefore, the sizes shown in Figure 5.9 do not necessarily show the maximum sizes in each period.

The JIT compiler uses a large work area when it compiles a large method, which may be due to inlining many methods or due to aggressive optimization. The largest JIT work areas in this measurement were about 20 MB in most of the intervals after the 25% allocation point. This is the reason the malloc-then-freed area increased after the 30% allocation point.

The size of the malloc-then-freed area occasionally increased, though it was around 9 MB for most of the intervals after the 30% allocation point. This is still under investigation, but we believe most of the malloc-then-freed area was the same memory used for the JIT work area. Since the JIT work area was large in some compilations, the size of the malloc-then-freed area increases after those compilations. However, as we noted in Section 5.4.2, not all of the freed memory is held in the malloc-then-freed area. The size of this area is the result of the interactions between the memory allocation and deallocation in the JIT compiler, and the algorithm used to maintain the free list in libc.

Figure 5.10 shows the corresponding memory usage on POWER. The code and JIT-compiled code memory areas were larger than for x86, as we observed with other benchmarks. The total difference in these areas was about 20 MB. The malloc-then-freed area was also larger than x86 by about 10 MB. These larger areas doubled the total resident set size of bloat compared with x86.

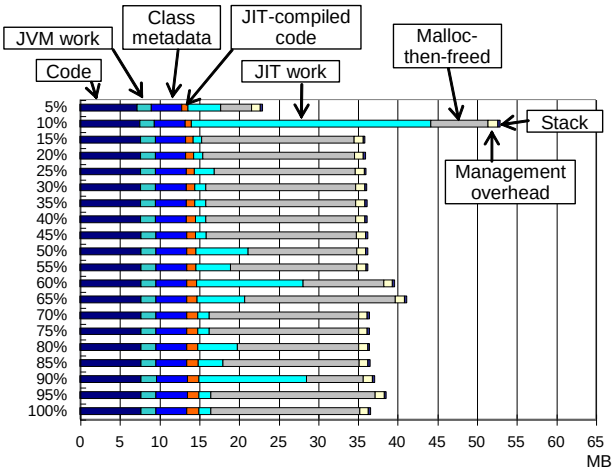


Figure 5.11. Results for DaCapo bloat on x86 when the size of malloc-then-freed area is large. (Graph scale is the same as Figure 5.9.)

The largest JIT work area was 35 MB on POWER. The number of JIT compilations that used more than 10 MB of the work area on POWER was 1.5 times more than on x86. More aggressive JIT optimization on POWER resulted in the larger malloc-then-freed area.

5.6 Reducing the Resident Set Size of the Malloc-then-freed Area

In this section, we discuss the problems with the malloc-then-freed area and evaluate a technique to reduce the resident set size with the macro-benchmark we used in Section 5.5.

5.6.1 Run-to-run Fluctuation of the Resident Set Size of the Malloc-then-freed Area

As shown in Sections 5.4 and 5.5, the malloc-then-freed area consumes a large amount of resident memory. The problem with this freed area is not just the extra memory consumption, but the difficulty of evaluating the true memory consumption of a program. We found the size of the freed area fluctuates widely during a single execution, and also varies significantly between runs of the same program. Even if the resident set size reported by the ps command changes after a program is modified, that does not prove that the code modification affected the memory use.

Figure 5.11 shows a breakdown of the memory usage in another execution of DaCapo bloat. The total resident set size was about 37 MB or more throughout the execution after the 10% allocation point, while it was around 28 MB at many points in the results of Figure 5.9.

5.6.2 Releasing Physical Pages to Reduce the Resident Set Size

As described in Section 5.2.6, the malloc-then-freed area is the memory held in the free list managed by libc, and this amount of memory depends on the algorithm used in the library. Since there is no API to tell the library about the intention of the memory usage in a program, applications have no control over whether a freed chunk should be kept in the free list for reuse in the near future, or whether it should be returned to OS. This lack of any API between libc and applications prevents effective memory management between the application and libc.

We can reduce the resident set size of the malloc-then-freed area by directly telling the OS to remove physical memory pages from the process memory. In Linux, we can use the madvise() system call for this purpose. Although the memory areas in the free list will still occupy address space, the size of resident memory can be reduced. This technique has been applied to general memory management systems [Feng and Berger 2005; TCMalloc[1]] and a Java heap [Hertz et al. 2005]. We applied this technique to the JIT work area because it produced most of the malloc-then-freed area. We reduced system call overhead by limiting the target memory area to the JIT work area, exploiting the knowledge of the implementation of the Java VM.

5.6.2.1 The Madvise System Call in Linux

This section briefly describes the madvise() system call in Linux and many UNIX-like operating systems. It advises the kernel how to handle paging input and output. An application can tell the kernel how it expects to use specific mapped or shared memory areas. The kernel can then choose appropriate read-ahead or caching techniques, though the kernel is also free to ignore this advice from the application. The available options and their behaviors differ among operating systems.

For example, MADV_DONTNEED is an option for madvise() indicating that the specified pages will not be accessed in the near future. In Linux, the kernel immediately releases the physical memory pages, but continues to reserve the virtual addresses. Subsequent accesses to the released pages will succeed, but the pages will be zeroed out. This behavior is specific to Linux, while some other operating systems (such as FreeBSD) require both the MADV_FREE and MADV_DONTNEED options for this effect. Since no application code can access the content of the freed areas, we can safely call madvise(MADV_DONTNEED) to remove such memory pages from process memory.

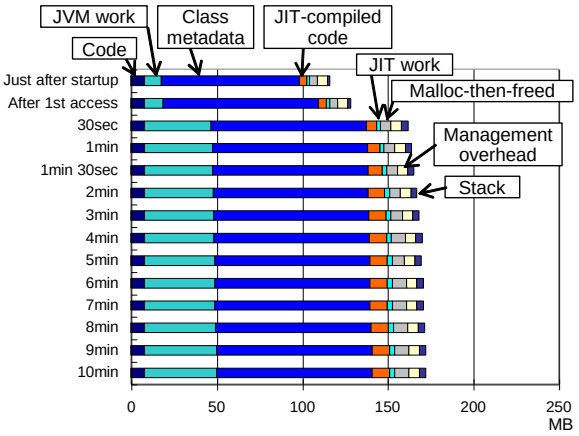


Figure 5.12. Non-Java memory breakdown for WAS 7.0 running Apache DayTrader on x86 when madvise() is called as the JIT work areas are freed. (Graph scale is the same as Figure 5.7.)

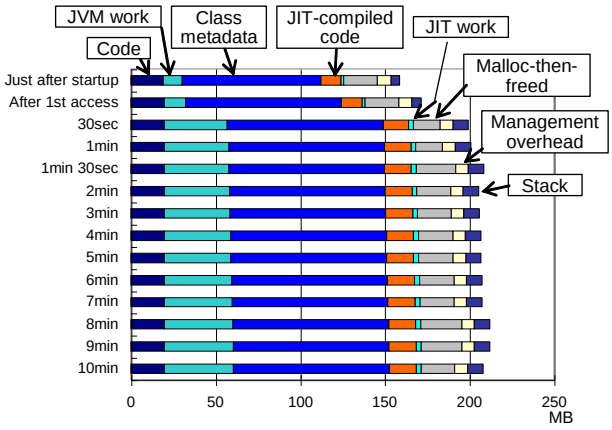


Figure 5.13. Non-Java memory breakdown for WAS 7.0 running Apache DayTrader on POWER when madvise() is called as the JIT work areas are freed. (Graph scale is the same as Figure 5.8.)

5.6.2.2 Calling the Madvise System Call from the JVM

We modified the IBM JVM to call the madvise() system call whenever a chunk of memory is freed. We actually need to call madvise() before the chunk is freed because another thread might reuse that memory when free() returns. Since our JVM has its own internal memory manager, we modified the memory manager to call madvise() just before calling free() when it is requested to free a memory chunk by other JVM components.

Note that we can call madvise() only when the size of a freed chunk includes an entire page within the address range of the freed chunk. We cannot call madvise() for a page that is only partially included in the range, because removing a memory page with madvise() erases all of the data in that page.

To avoid performance degradation, we should call madvise() only for memory chunks that will not be reused in the near future, such as a JIT work area. Therefore we modified the JVM to call madvise() only when a JIT work area is freed.

5.6.3 Savings by Calling the Madvise System Call

Figures 5.12 and 5.13 show the resident set size of Apache DayTrader in WAS 7.0 on x86 and POWER, respectively. We modified our JVM to use the madvise() system call upon freeing the JIT

---

[1] TCMalloc: Thread-Caching Malloc. Available at http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html

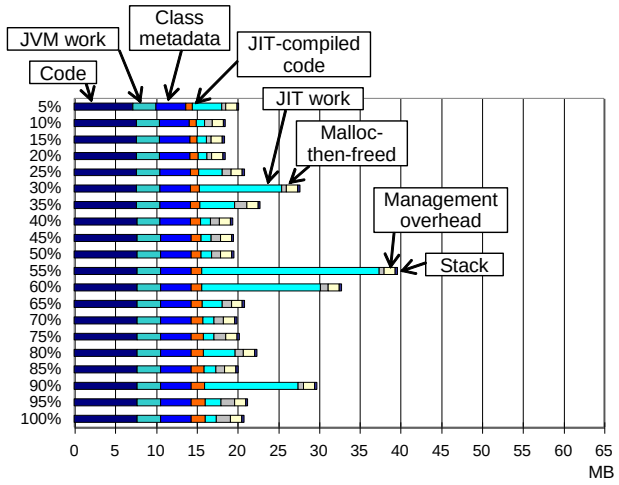


Figure 5.14. Non-Java memory breakdown for DaCapo bloat on x86 when madvise() is called as the JIT work areas are freed. (Graph scale is the same as Figure 5.9.)

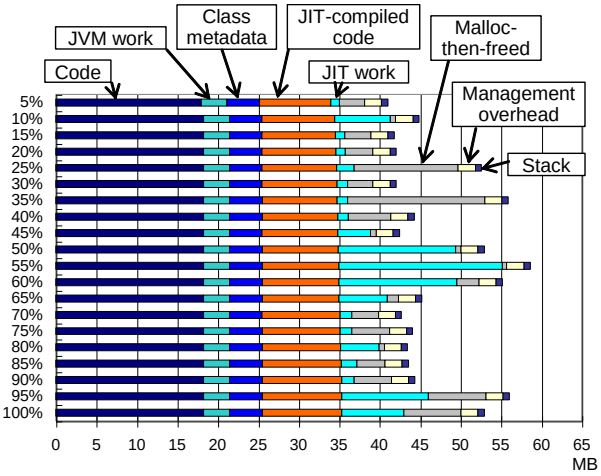


Figure 5.15. Non-Java memory breakdown for DaCapo bloat on POWER when madvise() is called as the JIT work areas are freed. (Graph scale is the same as Figure 5.10.)

work areas. We measured the same scenario as in Section 5.5.1. Using the madvise() system call, we reduced the resident set size of the malloc-then-freed area to around 10-15% on both x86 and POWER.

Figures 5.14 and 5.15 show the resident set size of the DaCapo bloat benchmark on x86 and POWER, respectively. Using the madvise() system call, we reduced the resident set size of the malloc-then-freed area to 16% on x86. In comparison, the reduction on POWER was smaller than on x86, only 41% of the size without madvise(). This variation is again explained by differences in the base page size. As described in Section 5.6.2.2, an entire page needs to be included in the address range of the freed area when the page is released with madvise(). Thus, the physical pages can be released only when the JIT compiler frees a memory chunk larger than 64 KB on POWER, while it is possible for a chunk larger than 4 KB on x86.

5.6.4 Performance Impact of Calling the Madvise System Call

Figure 5.16 shows the relative performance when the JVM calls madvise() when freeing the JIT work areas compared to the JVM without madvise(). We used DayTrader throughput and DaCapo benchmark execution times. Throughput was measured with Apache JMeter[1], and the number of iterations for the DaCapo benchmark was set to one.

[1] Apache JMeter is available at http://jmeter.apache.org/

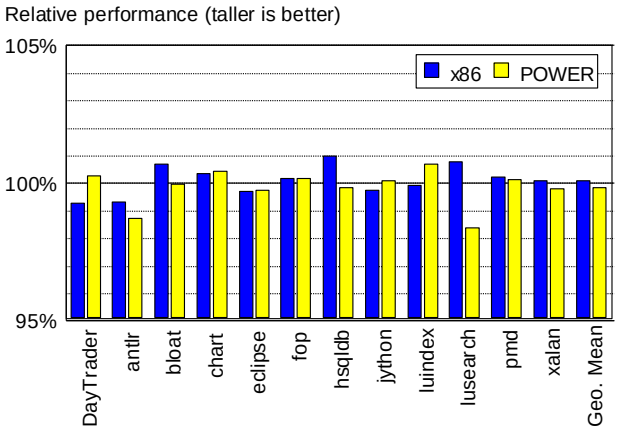


Figure 5.16. Relative performance of a Java VM that calls madvise() when freeing the JIT work areas compared to the Java VM without madvise().

Table 5.5. Execution environment for measuring disk I/O for swap-in and swap-out.

Hardware environment	
Machine	IBM BladeCenter LS21
CPU	Dual-core Opteron (2.4 GHz), 2 sockets
RAM size	8 GB
Hypervisor	Xen 3.1.0
CPU and memory allocated to the tested virtual machine	
CPU	1 CPU
Memory	1 GB
Software environment	
OS	RedHat Enterprise Linux 5.3
Kernel version	2.6.18
Java VM	IBM Java J9 VM for Java 6 (SR7), 32bit
Java heap size	333 MB (1/3 of allocated memory)

The differences of the performance between the JVM that calls madvise() when freeing the JIT work area and the JVM without madvise() were up to 1.0% and 1.6% on x86 and POWER, respectively. The geometric means of the differences on x86 and POWER were 0.1% and 0.2%, respectively. This measurement shows our approach has very small impact on performance.

5.6.5 Discussion

Since the malloc-then-freed area will eventually be reclaimed by the OS, it seems that we do not have to worry about this area even if a large amount of memory is allocated. However, since the OS does not know whether or not the content of the page will be used, it must swap the unnecessary data out to disk, and then swap it in when the malloc MMS touches the swapped-out pages while handling an allocation request. This can cause unnecessary thrashing in high-memory-use situations.

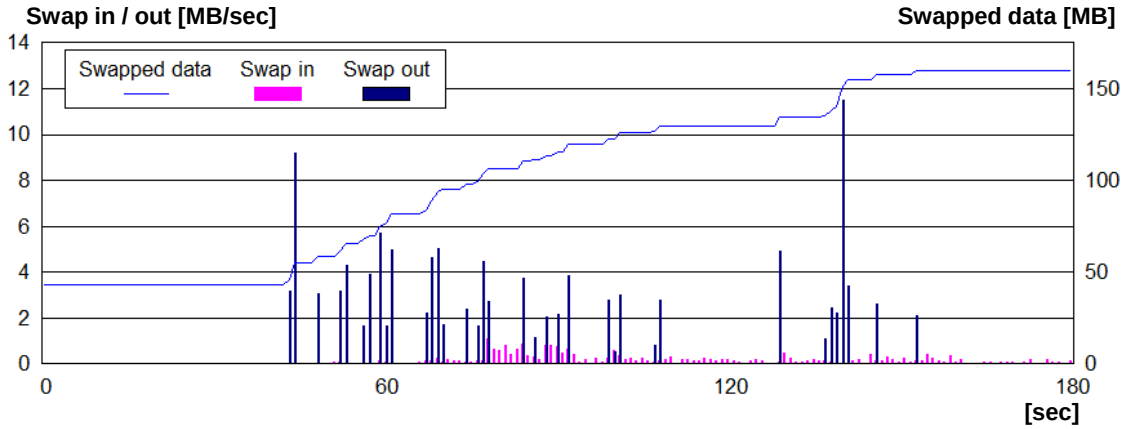


Figure 5.17. Disk I/O rate and the amount of swapped data during execution of two WAS processes running Apache DayTrader when madvise() was not called.

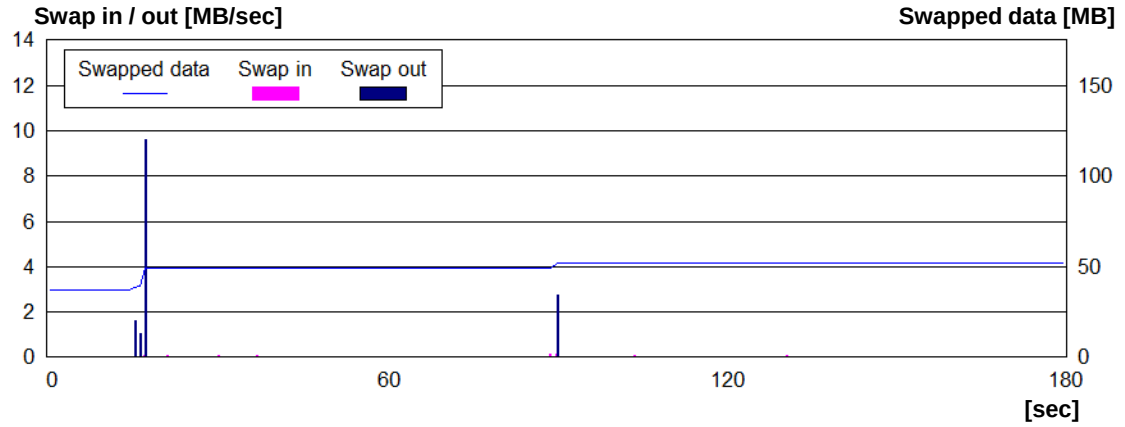


Figure 5.18. Disk I/O rate and amount of swapped data during execution of two WAS processes running Apache DayTrader when madvise() was called for the JIT work area.

We measured the number of bytes swapped in and out for a three minute of execution of two WAS processes, both running DayTrader. Table 5.5 describes the settings of this test environment. We used the Xen hypervisor to produce a high-memory-use situation by allocating a small amount of memory to the tested guest virtual machine.

Figure 5.17 shows the swapping activity when madvise() was not called. In this case, large amounts of swapping out occurred periodically during execution, and the total amount of swap space increased by 118 MB during this period. Swapping in also occurred continuously during this period. Figure 5.18 shows the results when madvise() was used. In this case, swapping was greatly reduced,

and the increase in the total size of the swap space was only 14.5 MB. This indicates that deleting unused data from the process memory and releasing the corresponding physical pages prevents unnecessary swapping and to retain good performance.

5.7 Related Work

There have been numerous papers and reports analyzing the Java heap, so we will only review a few of the most important ones. Sun's Java Development Kit Version 1.2 introduced the Java Virtual Machine Profiler Interface (JVMPI), and included the HPROF agent that interacts with the JVMPI to profile the use of the Java heap and the CPU [Liang and Viswanathan 1999]. For example, this agent can generate a heap allocation profile that shows the numbers and sizes in bytes of the allocated and live objects for each allocation site. The agent relates the allocation sites to the source code by tracking the dynamic stack traces that led to the allocations. The HPROF agent can also generate a complete heap dump to find unnecessary object retentions or memory leaks. In JDK 5.0, the JVMPI was replaced by the Java Virtual Machine Tool Interface (JVMTI) [1], and the HPROF [Sun Microsystems 2008a] agent was re-implemented using the JVMTI.

The IBM Support Assistant (ISA)[1], a free software serviceability workbench, analyzes the dump produced by a JVM, helping developers identify common problems such as memory shortages, deadlocks, and crashes. It provides basic support for diagnosing memory problems, such as showing statistics for the live objects in a Java heap and the class metadata.

Even if complete heap dumps are available and there are available tools for viewing the dumps, diagnosing memory leaks is a significant challenge for developers. Mitchell and Sevitsky [Mitchell and Sevitsky 2003] proposed an automated and lightweight tool, LeakBot, for diagnosing memory leaks. It ranks data structures by their likelihood of containing leaks, identifies suspicious regions, characterizes the expected evolution of memory use, and tracks the actual evolution at run time. It can improve accuracy of the analysis by finding growing data structures if multiple dumps of the same process are provided. LeakBot was incorporated into another tool named Memory Dump Diagnostic for Java (MDD4J) [Poddar and Minshall 2006], which is also available as a plug-in for ISA.

Mitchell and Sevitsky [Mitchell and Sevitsky 2007] did an analysis of Java heaps, focusing on the overhead of collections. They introduced a *health signature* to distinguish the roles of the bytes

[1] http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html

based on the roles of the objects in the collections, and provide concise and application-neutral summaries of the heap usage. Kawachiya et al. and Horie et al. did another analysis of Java heaps, focusing on Java strings [Kawachiya et al. 2008; Horie et al. 2014] . Analyzing Java heap snapshots, they found that there are many identical strings, and proposed three different savings techniques, including one to "unify" the duplicates at garbage collection time.

Java's non-Java memory is not well described or documented before our original paper was published. Chawla provides a brief overview of how IBM's 32-bit Java VM uses the address space in AIX, though IBM’s Java VM for 1.4.2 can behave differently from IBM’s Java 5 and Java 6 VMs [Chawla 2003]. Hanik describes the memory layout of a JVM process, and considers the causes of and solutions for out of memory errors [Hanik 2007].

Some people became aware of the overhead of non-Java memory, especially the class metadata area. Schatzl et al. analyzed the GC time to traverse references in the class metadata and reduced it by separating class metadata area into metaspaces prepared for each class loader and using reference links among the metaspaces to avoid full traversal of the class metadata [Schatzl et al. 2011]. Sewe et al. created a benchmark suite for Scala. They verified that the ratios of interface and abstract classes against all loaded classes are comparable to those ratios in the DaCapo benchmark suite [Sewe et al. 2011]. Lin et al. verified that the amount of objects for Jikes RVM itself is not small [Lin et al. 2012]. They analyzed the amount of non-Java memory by tracking execution context to distinguish Jikes RVM's objects from applications' objects, both of which are allocated in the Java heap.

The large non-Java memory becomes a visible overhead in the case of a metacircular Java VM, which is a Java VM written in Java, because the data structures of the Java VM itself are created in the Java heap. Wimmer et al. reported that the minimum size of the Java heap to run an application is larger in their metacircular Java VM Maxine than the HotSpot VM because of the Maxine's framework objects [Wimmer et al. 2013]. The HotSpot VM stores class metadata into the permanent area, which is a part of non-Java memory .

There are some researches for ameliorating the issue caused by access patterns that does not fit with the LRU policy. EELRU [Smaragdakis et al. 1999] detects a memory access pattern that repeatedly scan larger amount of memory than physical memory size and evicts pages of n-th most recently used pages. This technique, however, is not applicable to avoid the memory blocks lagging

[1] IBM Support Assistant is available at http://www.ibm.com/software/support/isa/

in the free list because they are not repeatedly scanned. Another approach is to explicitly specify memory blocks that will not be used later and can be deleted regardless of their recency. evict-me [Wang et al. 2002] implemented such mechanism as a bit in the data cache in a processor and let compiler to set the bit. Cooper et al. implemented such mechanism in garbage collector for SML [Cooper et al. 1992]. Their garbage collector avoids paging out unused pages by telling Mach external pager that those pages are discardable.

5.8 Summary

We quantitatively analyzed the usage of non-Java memory for a wide range of Java applications. Using a modified version of a production Java VM for Linux, we verified that a Java application consumes a considerable amount of non-Java memory. We found that non-Java memory could become as large as the Java heap in many Java programs.

A Java VM uses non-Java memory for various purposes. The non-Java memory holds shared libraries, builds the class metadata, provides the work area for generating the JIT-compiled code, and has the dynamic memory used to interact with the operating system. Although a plethora of memory problems affect the Java heap, similar problems can also appear in the non-Java memory. For example, an out-of-memory exception will be raised when the virtual machine loads or dynamically generates too many classes based on the requests from an application. Modern Java VMs tend to use more non-Java memory. For example, they may dynamically generate classes to optimize reflective invocations, while also allocating direct byte buffers to improve I/O performance. In addition, a trend to build scripting language runtimes on top of JVMs also tends to use more non-Java memory by generating Java classes dynamically. Examples include JRuby, Jython, and Groovy.

Through time series analysis, we observed that the JIT work area had significant fluctuations in the use of non-Java memory, because the JIT compiler intermittently requires large amounts of temporary memory for aggressive optimizations. We also observed that the libc memory management system (MMS) has a profound impact on the resident memory of non-Java memory, because it may retain the memory chunks freed by an upper-level MMS. This suggests that the layers of MMSes should be more carefully integrated. For example, an upper-level MMS may need the ability to force the libc MMS to return free memory to the OS-level MMS. In this research, we evaluated a technique to compensate for the lack of integration between libc MMS and upper-level MMS by directly telling the OS-level MMS to remove memory pages.

We also verified that this technique reduced swapping activity during the execution of two WAS processes in a high-memory-use situation. Since virtualized computation environments on a hypervisor, such as the servers in a cloud data center, are becoming popular, such high-memory-use situations will be more common. Our technique for in-depth analysis of non-Java memory is also useful for improving effectiveness of the memory over commitment by identifying unnecessary memory use.

6. Replay Debugging of the Java JIT Compiler

The performance of Java has been tremendously improved by the advance of Just-in-Time (JIT) compilation technologies. However, debugging such a JIT compiler is much harder than a static compiler. Recompiling the problematic method to produce a diagnostic output does not necessarily work as expected, because the compilation of a method depends on runtime information at the time of compilation.

In this chapter[1], we propose a new approach, called *replay JIT compilation*, which can reproduce the same compilation remotely by using two compilers, *the state-saving compiler* and *the replaying compiler*. The state-saving compiler is used in a normal run, and, while compiling a method, records into a *log* all of the input for the compiler. The replaying compiler is then used in a debugging run with the system dump, to recompile a method with the options for diagnostic output.

6.1 Overview

The performance of Java has been tremendously improved. Undoubtedly, advances in dynamic compilation technologies have significantly contributed to these improvements. Java JIT compilers perform increasingly more advanced, and thus more complicated, optimizations [Ishizaki et al. 2000; Paleczny et al. 2001; Suganuma et al. 2000], and can even generate more efficient code than static compilers by taking advantage of runtime profiles.

However, a JIT compiler is much harder to debug than a static compiler. Assume that an application crashed in a production environment, and analysis suggests that the code generated for a certain method may be causing the crash. What will then be the next step? If the application was developed with a static compiler, we can simply recompile the method with an option to produce *diagnostic output*. The diagnostic output contains all the details of what the compiler does, including what optimizations are applied and how each optimization transforms the code. This greatly helps a compiler writer analyze a bug in the compiler, and the compiler writer can often recognize the bug without re-executing the compiler using a debugger. Without a diagnostic output, it is very difficult to associate each generated machine instruction with the source code.

[1] This chapter is based on my work presented at OOPSLA 2006 [Ogata et al. 2006].

We could do the same thing when the application is written in Java. More precisely, we could again JIT compile the problematic method by rerunning the application with an option specified to produce diagnostic output. However, this does not necessarily work, because the method may not be compiled in exactly the same way. The reason is that the compilation of a method depends not only on the method's bytecode but also on the runtime information at the time of compilation, such as the resolution status of classes referenced in the method, the class hierarchy, and the runtime profile. This runtime information is not necessarily the same from run to run because the Java application is multi-threaded, and non-determinism in execution is unavoidable. We actually observed that the combination of applied optimizations had changed in at least one out of ten executions for each of the Java programs we evaluated because of changes in the execution order of threads and the runtime profiles.

A straightforward solution to get diagnostic output would be to run an application with the diagnostic option specified even in a production environment. However, this significantly increases the compilation time and thus the execution time of the application. In addition, forcing the compiler to always generate the diagnostic output would require prohibitively large amounts of disk space. For example, the diagnostic output for a single execution of a SPECjvm98 benchmark can be in the hundreds of megabytes. Enterprise applications would produce much larger diagnostic output, and spending large amount of customers' disk just for debugging JIT compiler will not be acceptable for the customers.

In this research, we propose a new approach for debugging a JIT compiler, replay JIT compilation, which allows methods to be recompiled in exactly the same way as in a previous run. Our approach uses two compilers, the state-saving compiler and the replaying compiler. The state-saving compiler is used in a normal run, and, while compiling a method, records into a log all of the runtime information referenced during the compilation. The log is in the main memory, and automatically included in the system dump when the application crashes. The replaying compiler is then used in a debugging run with the system dump, to recompile a method with the options for diagnostic output. This technique is a kind of trace-and-replay technique, but we successfully minimized its overhead by saving in memory (not on hard disk) only the additional trace information that will otherwise not appear in the system dump. As a result, the system dump will always include all the information required for replaying the JIT compiler to recreate the failure.

It is worth noting that using a system dump to reproduce a problem that crashed a mission-critical application is much better than trying to reproduce the problem by recreating the environment in which the application crashed at a remote site. Such an application tends to be very complicated to install, configure, and deploy, and may demand substantial hardware resources. Thus, it would be laborious to set up the same environment at a remote site to reproduce the observed problem. In addition, it may be impossible to obtain the data to run the application if the data includes highly confidential or sensitive information such as credit card numbers. Without the same data, the application will run differently and the problem may not be reproduced.

We implemented our prototype based on the J9 Java VM [Grcevski et al. 2004] and the TR JIT compiler [Stepanian et al. 2005; Sundaresan et al. 2006] for AIX. This prototype also implements confidence-based filtering to save only the logs that are likely to be needed for debugging. Our experiment showed that the time overhead for saving the input is only 1%, and the space overhead for saving the input is only 10% of that of the compiled code.

This research makes the following contributions:

- **Replay JIT compilation:** We reduced the overhead for saving the input for the JIT compiler by using a system dump so that it can always be enabled in a production environment.
- **Confidence-based filtering:** We can reduce the size of the input to save by considering the likelihood that a method may cause an error in the JIT compiler.

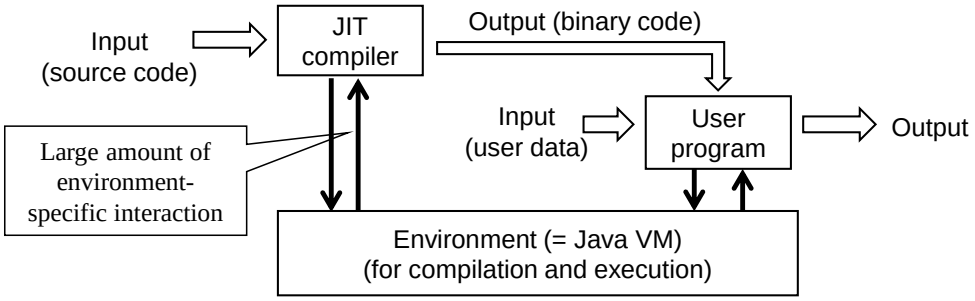
6.2 Reproducing the Behavior of a JIT Compiler

This section explains the difficulties in debugging a JIT compiler and gives the basic idea of our solution.

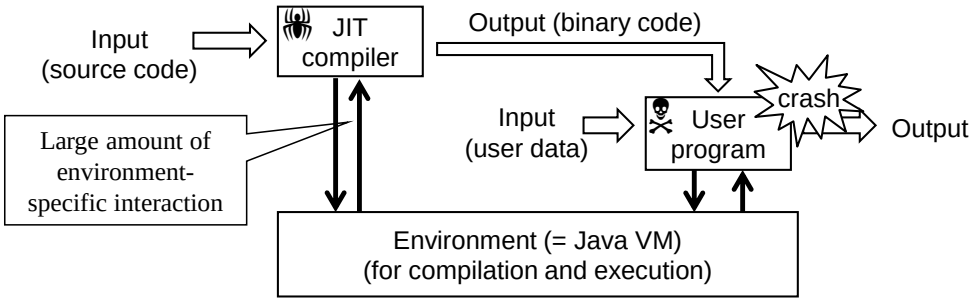
6.2.1 Execution Models for a JIT Compiler

Let us explain the difficulty in debugging a JIT compiler by comparing its execution model with the model of a static compiler, as shown in Figure 3.1 (b) and Figure 3.2 (b).

Figure 6.1 (a) shows the execution model for the JIT compiler. The difference from the model for a static compiler is the environment for compilation. A JIT compiler shares its environment with the user program, and thus, the executions of it and the user program affect each other through the environment.



(a) Execution model of a JIT compiler



(b) Execution model when a compiled user program crashes because a problematic JIT compiler generated the wrong binary code

Figure 6.1. Execution model of a JIT compiler.

Unlike static compilers, JIT compilers aggressively interact with the environment because they can generate better binary code that is customized to the environment at that time. They often generate more optimized code than static compilers do as a result of customizing the code to the current environment. For this reason, JIT compilers try to use as much information as possible for generating highly customized code.

6.2.2 Difficulty in Debugging a JIT Compiler

A drawback of this execution model is that it makes a JIT compiler much harder to debug than a static compiler. For debugging a JIT compiler, compiler developers need to access the environment where the user program was executed and need to reproduce the execution of the user program, as shown in Figure 6.1 (b). If a problem occurred in the customers' environment, the compiler developers need to access that environment for debugging it, but this is very difficult in reality.

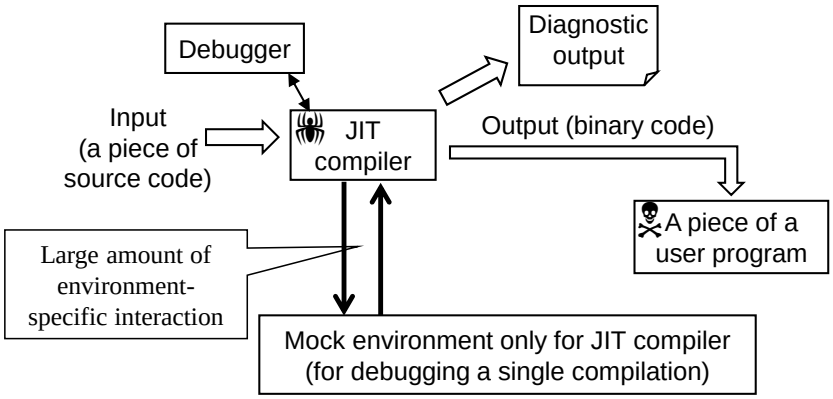


Figure 6.2. Execution model for debugging a JIT compiler using a mock environment.

Another issue is non-determinism of the JIT compiler's behavior. Even if compiler developers can access the customers' input and environment, they may not be able to reproduce the problem by simply rerunning the program because enterprise programs often use multiple threads and operate non-deterministically. A Java VM also creates system threads for compilation and garbage collection, and it can change the time when the JIT compiler is invoked from run to run. Java VMs may also use runtime profilers, and they can change the list of methods to be compiled and the set of optimization algorithms to be applied for each method. These factors non-deterministically change the states of the execution environment. Thus, different runs of an application may result in different methods being compiled. Even if a specific method is compiled in two runs of an application, the compiled code may not be the same because the inputs for the JIT compiler may not be identical.

6.2.3 Reproducing JIT Compilation Using a Mock Environment

We can use a mock environment to reproduce the JIT compilation of a specific method, as shown in Figure 6.2. A mock environment emulates the real environment by providing the same data to the JIT compiler in the same order as the real environment. The JIT compiler must operate in the same way as if it were invoked in the production environment because it obtains the same sequence of inputs.

A mock environment is also useful for efficient debugging. Compiler developers can invoke a JIT compiler as if it were a stand-alone tool by specifying a method to compile. They can start debugging the compilation of the method soon after running the mock environment, without waiting for execution of the user program until a problematic compilation starts.

6.3 Creating a Mock Environment for a JIT Compiler

We can use trace-and-replay techniques to implement the mock environment. This section describes how we can efficiently apply trace-and-replay techniques to a JIT compiler by utilizing the characteristics of the input to a JIT compiler.

6.3.1 Trace-and-replay Techniques for a JIT Compiler

Trace-and-replay is a common technique for debugging multi-threaded programs, and it can be categorized to two approaches: the ordering-based and content-based [Ronsse et al. 2000] approaches.

The ordering-based approach is to record and replay the order of synchronization events [LeBlanc and Mellor-Crummey 1987], such as locking and message passing. For this approach, many techniques [Bacon 1991; Choi and Srinivasan 1998; LeBlanc and Mellor-Crummey 1987; Miller and Choi 1988] have been developed and discussed to trace and replay the program execution with small overhead. For a Java JIT compiler, the compiler itself operates deterministically, but the compilation results may change because the input for the compiler may change non-deterministically during the execution of the Java program. The input for the JIT compiler is runtime information from the Java VM, and that data is changed by many of the Java operations, such as object allocation, access to a field variable, or method invocation. The changes made by other threads immediately change the input for the compiler. Thus, it is impractical to use the ordering-based approach for a Java VM because recording the order of all of those operations is needed to reproduce the input for the JIT compiler.

The content-based approach is to save and restore the values of the input for JIT compilers. It is straightforward approach, and it is easy to implement the mock environment for a JIT compiler, because a JIT compiler is usually an independent component in a Java VM and it interacts with the Java VM using a well-defined API. Thus, we can record all the values provided from the Java VM to the JIT compiler by capturing data go through the API.

The problem of content-based approach is high runtime overhead for recording data. For example, Recap [Pan and Linton 1988] is estimated to generate 1 Mbyte of trace per second, even on a slow VAX-11/780 machine, and a faster machine could generate an unacceptably huge amount, such as 1 Gbyte per second. It causes a large overhead because the JIT compiler needs to generate log file regardless of if it is being debugged. The large overhead results in degrading the performance of the

user programs because the execution time of a JIT compiler is accumulated to the execution time of the program compiled with the JIT compiler.

This means, however, that we can use this approach if we devise a technique to reduce overhead to record the input as small as possible.

6.3.2 Runtime Information as Inputs

While compiling a method, the JIT compiler exploits runtime information which can be categorized into three types, the system configuration data, the virtual machine states, and the runtime profiles.

An example of system configuration data is whether the underlying system is uniprocessor or multiprocessor. The JIT compiler generates faster code for synchronization for a uniprocessor system. Another example is the processor architecture of the underlying system. The JIT compiler exploits the information to generate instructions only available in a specific processor architecture. Note that, with the static compiler, the user can specify system configuration data as command line options. This implies that the user must prepare different executables for different configurations, and pick up the right executable based on the actual execution platform.

A common example of the virtual machine states is the resolution status for external references [Gosling et al. 1996]. The bytecode of a method contains external references to classes, fields, and methods. When the virtual machine loads a class, all of the external references are *symbolic*. During the execution, references will undergo resolution and become *direct* references. The JIT compiler generates faster code for direct references, and code to force the resolution for symbolic references. Another example is the hierarchy of classes loaded into a virtual machine. The JIT compiler analyzes the hierarchy to devirtualize method invocations [Ishizaki et al. 2000]. Devirtualization is one of the most important optimizations for object-oriented programs.

The dynamic compilation system is considered to be best positioned for the profile-guided optimizations since it can be made transparent to collect runtime profiles. The runtime profile of a method could be based on block profiling, edge profiling, or path profiling. The JIT compiler generates optimized code in favor of frequently executed basic blocks, edges, or paths [Arnold and Ryder 2001; Whaley 2000; Yasue et al. 2003]. The runtime profile may even include value profiles, or distributions of values of arguments and variables. The JIT compiler then creates specialized versions of a method based on values frequently observed.

6.3.3 Variable and Fixed Inputs

We can categorize the inputs for a JIT compiler into two types, *variable inputs* and *fixed inputs*. If the inputs may change after a compilation of a method, we call them variable inputs. Otherwise, we call them fixed inputs. For example, the resolution statuses of the external references are variable inputs. Also, the class hierarchy is variable input since new classes may be loaded after the compilation. On the other hand, the system configuration data is fixed input. Also, the bytecode of the method can be a fixed input if the virtual machine prohibits the rewriting of the bytecode.

Table 6.1 of Section 6.3.2 shows a more detailed summary of inputs for the JIT compiler, including whether they are variable or fixed. Among the four types of input (the target method, the system configuration, the virtual machine states, and the runtime profiles), the first two types are fixed inputs and the others are variable inputs.

Note that, while the values of the variable inputs must be saved at the time of compilation, those of the fixed inputs can be saved at the arbitrary time.

6.4 Overview of Replay Compilation

We use the content-based trace-and-replay technique to reproduce the compilation by a JIT compiler. Concretely, we create two versions of a JIT compiler, the *state-saving compiler* and the *replaying compiler*. In our approach, the virtual machine invokes the state-saving compiler to dynamically compile methods. This compiler saves all of the inputs for each compilation into a *log*. Later, the user invokes the replaying compiler by specifying a method to be replayed. The replaying compiler reproduces the compilation of the method by restoring all of the inputs for the compilation from the corresponding log.

Since the virtual machine invokes the state-saving compiler while running an application, the overhead of the compiler must be sufficiently small both in terms of time and space. As we will show, we employ a cascade of techniques to reduce the overhead.

Note that the replay compilation is meant to support the debugging of a JIT compiler. Assume that a problem occurred while a Java application is running. The replay compilation is not a tool for analyzing the problem in general. Instead, it should be used when the analysis of the problem *suggests* that an execution of a compiled method caused the problem, and that the JIT compiler failed to generate the code correctly.

6.4.1 System Dump Based Approach

We assume that the virtual machine is configured to generate a *system dump* at crashes and user interrupts. As a `core` file in UNIX systems [IEEE and The Open Group], a system dump contains the memory image of an OS process. Exploiting this assumption, the state-saving compiler creates the logs for compilations in the main memory, not explicitly writing them into a file. The logs are then automatically saved into a system dump when the operating system creates one. Avoiding expensive I/O during compilation significantly helps reduce the time overhead of the state-saving compiler.

Note that this system-dump-based approach has a significant advantage. It does not require the virtual machine to be invoked to rerun an application. It suffices to invoke the replaying compiler with the system dump. Furthermore, the platform where the replaying compiler is invoked does not have to be identical to the platform where the system dump was generated.

Thus, one of the scenarios which are only made possible by our approach is as follows. The customer invokes the virtual machine in a production environment which runs a mission critical application, invoking the state-saving compiler. The virtual machine crashes, and a system dump is generated. The customer sends the system dump to the support personnel at a different site. They invoke the replaying compiler in their environment to fix a problem in the compiler.

Developers of a JIT compiler can also benefit from our approach. Assume that a test case is highly multi-threaded and thus runs in a very non-deterministic manner. Obviously, it is hard to reproduce an error in such a test case by rerunning it. With replay compilation, developers do not have to run the test case repeatedly. They only have to invoke the replaying compiler.

6.4.2 Log Structure

In general, the JIT compiler gets the inputs for a compilation, whether variable or fixed, by accessing data structures and calling functions. For an input by data structure access, the state-saving compiler records a pair of the address and the value into a log. Later, the replaying compiler retrieves the value from the log, by using the address as the key. For an input in a function call, the state-saving compiler records into a log the return value together with a list of function identifier and zero or more parameter values. The replaying compiler retrieves the return value from the log by using the list as a key.

The system-dump-based approach makes an optimization possible for a fixed input by data structure access. As mentioned in Section 6.2.6, while the values for variable inputs must be saved at

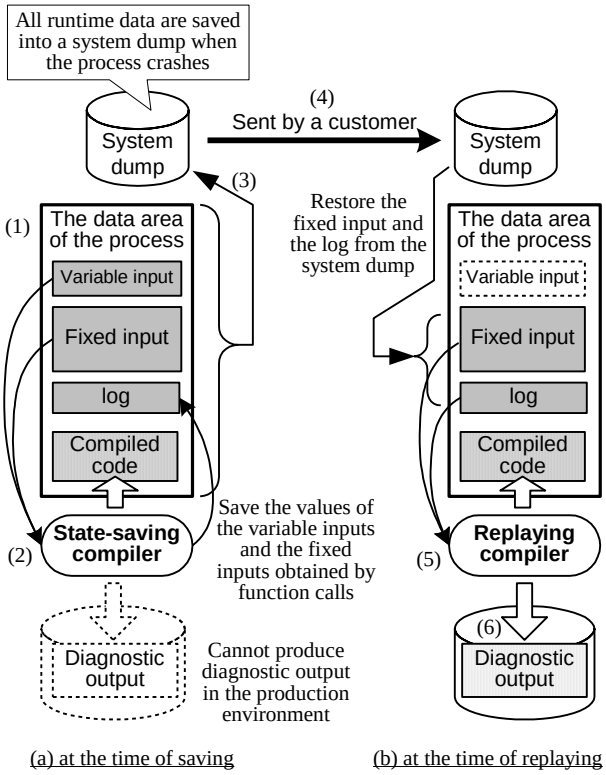


Figure 6.3. Replay JIT compilation

the time of the compilation, the values for fixed inputs can be saved at the arbitrary time. In particular, the values for fixed inputs can be saved at the time of creating the system dump. Thus, the state-saving compiler does not have to save anything for fixed inputs during compilation, simply relying on the system dump that will later be generated. Since fixed inputs in data structures access occupy the largest share in inputs for a Java JIT compiler, this also contributes to reducing the overhead of the state-saving compiler.

Figure 6.3 summarizes the discussion so far by using a typical scenario. (1) The virtual machine is running a Java application at a customer site (left figure). (2) The virtual machine invokes the state-saving compiler, which saves the inputs for compilations into logs. (3) The virtual machine then crashes, causing the operating system to automatically create the system dump. (4) The customer sends the system dump to the service personnel at a remote site. (5) The service personnel invoke the replaying compiler with the system dump to reproduce a problematic compilation. (6) The replaying compiler, running at their site, restores the inputs for the compilation from the corresponding log, generating detailed diagnostic output for the compilation.

Table 6.1. Types of input used by the Java JIT compiler and the values to be saved in a log

Type of input	Input for the JIT compiler	Value to be saved into a log	How the JIT compiler uses the input
Target method (fixed input)	Bytecode, literals, and the metadata for external references	Address of the data structure associated with the target method	The JIT compiler reads these inputs as source code.
System configuration (fixed input)	Configuration of the hardware and the software	Model, cache size, number, and specifications of the processors in the machine, and the type and version of the operating system	The JIT compiler can generate code that can run faster in a specific environment than generic code.
	Command line options and the environment variables	Address of the data structure that holds the parsed command line options and the environment variables	Those options may change the compilation process for all or particular methods.
Virtual machine states (variable input)	Set of classes that are referred to and that have been initialized	A flag for each class indicating if the class has been initialized	When the class has already been initialized, the JIT compiler can generate faster code, since the generated code need not handle the initialization.
	Address of the compiled code	Addresses of the compiled code invoked from the method being compiled	The JIT compiler can generate code that directly calls the compiled code of the callee method if it is already compiled.
	Saved results of the JIT optimizations	Addresses of the classes that hold the results of the inter-procedural analysis	The JIT compiler can reuse the saved results of inter-procedural analysis to reduce compilation time.
	Class hierarchy of the loaded classes	A set of the parameters and the return value of each function call for devirtualization	The JIT compiler can use the class hierarchy analysis to devirtualize the method invocation of virtual and interface methods.
	Resolution statuses	A bitmap indicating which of the external references have been resolved	For each resolved reference, the JIT compiler can generate faster code, since the generated code need not handle the resolution. The JIT compiler may be able to inline the callee method when the reference to it has been resolved.
Runtime profiles (variable input)	Runtime profiler output	The values of the runtime profiles	The JIT compiler will apply more aggressive optimizations to the frequently executed path, or can generate code that is specialized for the frequently appearing values.
	Optimization level	A value of the optimization level that is determined based on the runtime profile	The JIT compiler selects the set of optimizations to apply based on the optimization level that was determined based on the profiler output.

Table 6.1 summarizes the discussion so far for the inputs for a JIT compiler. It shows the types of inputs for a JIT compiler together with values to be saved into logs. It also describes how a Java JIT compiler uses the inputs for optimizations.

6.4.3 Building State-saving and Replaying Compilers

We describe how the state-saving and the replaying compilers are developed. We first build a compiler in such a way that it uses macros to get the inputs for a compilation. Since our base JIT compiler gets the inputs by accessing data structures and by calling functions, we converted those codes getting the inputs to use macros. We then provide two sets of definitions for the macros, one for the state-saving compiler and the other for the replaying compiler. The definitions for the state-saving compiler will store the inputs into a log as well as return the inputs, while the definitions for the replaying compiler will retrieve the inputs from a log. In this way, we can derive the two compilers from a single source.

Figure 6.4 illustrates the details. Figure 6.4 (a) shows the code of the conventional compiler. Receiving the name of the target method, the function `compile` obtains the number of available processors, a data structure `md` associated with the target method, the addresses of the bytecode, and a bitmap that holds the resolution status. We assume that the number of the available processors and the bytecode are the fixed inputs, while the others are the variable inputs. Figure 6.4 (b) presents the macro version for replay compilation. We wrap with a macro every piece of the code which obtains an input. We use different macros, depending on whether inputs are variable or fixed and whether inputs are obtained by accessing data structure or by calling functions.

Figure 6.4 (c) lists the definitions of the macros for the state-saving compiler. Because of the optimization mentioned in Section 6.2.6, the `getFixedInputByDataAcc` macro is simply defined to do the same operation as in Figure 6.4 (a), and not to save anything into a log. Figure 6.4 (d) shows the definitions for the replaying compiler. The `getFixedInputByDataAcc` macro is defined to get the input from the system dump, not from the log. As we will explain in Section 6.3.4, we may need to adjust the address appropriately.

The replay compilation assumes that both of the state-saving and the replaying compilers apply the same set of optimizations for the same inputs. Thus, the versions of the source code for the state-saving and the replaying compilers must be synchronized. Deriving the two compilers from a single source greatly simplifies this task of synchronization.

```
compile(char *signature) {
  int      numCPU;
  Method  *md;
  char    *bytecode;
  BitMap  *map;


  numCPU = getNumProcessor(); //Fixed input

  md = getMethod(signature);  //Variable input

  bytecode = md->bytecode;    //Fixed input

  map = md->resolveMap;       //Variable input

  ...
```

(a) an example of the code in a base compiler

```
compile(char *signature) {
  int      numCPU;
  Method  *md;
  char    *bytecode;
  BitMap  *map;
  Log     *log = setupLog(signature);

  getFixedInputByFunc0Arg(int,
                  numCPU, getNumProcessor);
  getVariableInputByFunc1Arg(Method *,
                 md, getMethod, signature);
  getFixedInputByDataAcc(Method,
                   bytecode, md, bytecode);
  getVariableInputByDataAcc(BitMap*, Method,
                      map, md, resolveMap);
  ...
```

(b) the code modified for the replay compilation

```
#define getFixedInputByFunc0Arg(           \
                    destTpye, dest, func) { \
  dest = func ## ();                       \
  putLogForFunc(dest, getFuncID(func));    \
}

#define getVariableInputByFunc1Arg(        \
              destType, dest, func, arg1) { \
  dest = func ## (arg1);                   \
  putLogForFunc(dest,                      \
                    getFuncID(func), arg1); \
}

#define getFixedInputByDataAcc(            \
              baseType, dest, ptr, field) { \
  dest = (ptr)->field;                     \
}

#define getVariableInptByDataAcc(destType,  \
             baseType, dest, base, field) { \
  int  offset = offsetof(baseType, field);  \
  dest = (base)->field;                     \
  putLogFodDataAcc(dest,                    \
        getTypeID(baseType), base, offset); \
}
```

(c) the definitions of the macros (state-saving compiler)

```
#define getFixedInputByFunc0Arg(           \
                    destTpye, dest, func) { \
  dest = (destType)getLogForFunc(          \
                          getFuncID(func)); \
}

#define getVariableInputByFunc1Arg(        \
              destType, dest, func, arg1) { \
  dest = (destType)getLogForFunc(          \
                    getFuncID(func), arg1); \
                                           \
}

#define getFixedInputByDataAcc(            \
              baseType, dest, ptr, field) { \
  dest = adjustAddr(baseType, ptr)->field;  \
}

#define getVariableInptByDataAcc(destType,  \
             baseType, dest, base, field) { \
  int  offset = offsetof(baseType, field);  \
  dest = (destType)getLogFodDataAcc(        \
        getTypeID(baseType), base, offset); \
                                           \
}
```

(d) the definitions of the macros (replaying compiler)

Figure 6.4. An example of the code to get the input for the compiler

6.4.4 Replaying the Compilation

While the virtual machine invokes the state-saving compiler, the user invokes the replaying compiler by specifying one or more compilations to be reproduced. This is usually done by specifying method signatures or addresses of compiled code. Note that, like a static compiler, the replaying compiler

independently reproduces the specified compilations. There is no restriction on the replay order, and we can even replay all of the compilations in the reverse of the order in which the sate-saving compiler did.

As in Figure 6.3, it is common that the replaying compiler and the virtual machine invoking the state-saving compiler run on different platforms. Using the replaying compiler, the support personnel do not have to create exactly the same platform as the state-saving compiler used. This significantly reduces the cost for the support division.

When it is invoked, the replay compiler first loads the system dump into its address space. It then attempts to reproduce a compilation by retrieving the inputs for the compilation. A tricky part of the replaying compiler is that if an input retrieved from the log is an address, it is the address in the process for the state-saving compiler (state-saving process). The replaying compiler cannot use the address to retrieve the value of a fixed input, since the system dump is not necessarily loaded at the same address as the sate-saving process. This is the reason why we need the adjustment in the `getFixedInputByDataAcc` macro.

We may be able to avoid this adjustment as follows. Assume that we can know the address range in the system dump where fixed inputs reside. If the replaying compiler can reserve the address range at start-up, it can load the data from the system dump into the address range, restoring the fixed inputs at the same addressees. However, it depends on the underlying operating system and the details of how the process for the replaying compiler is initialized.

6.4.5 Further Reducing the Size of Logs

The overhead of the state-saving compiler must be small enough both in terms of time and space, since the virtual machine invokes it while running an application. Exploiting the system-dump-based approach, the state-saving compiler allocates logs in the main memory, and skips saving fixed inputs by data structure access. The former significant help contributes to reducing the time overhead, while the latter contributes to reducing both the time and space overhead.

Here we show three techniques to further reduce the space overhead. The first technique is to compress the logs in memory. This is simple yet very effective in reducing the log size.

The second technique is to exploit *default* values. The state-saving compiler skips saving into a log an input when the value equals the default value, while the replaying compiler interprets the value of an input as default when it can find in the log no value corresponding to the input. Default values

should be defined as frequently observed values, and different default values can be defined for different functions and data structure types.

The third technique is to skip saving all of the inputs for the compilation of a method if we have high *confidence* in the method, or if we can assume that the method is very likely to be compiled correctly. We call this technique confidence-based filtering. We define the confidence of a method as follows.

- Increase the confidence of a method, if the method is in heavily used libraries such as the Java core classes
- Decrease the confidence of a method, if the control flow of the method is complex.

The rationale behind the first criterion is that methods in heavily used libraries are frequently compiled and executed, and thus the paths to compile them are already well tested. The second criterion is simply based on our rule of thumb. We often observed that a problem in the compiler appeared when the compiler processed a complex control flow. We can approximate the complexity of the control flow of a method as the number of basic blocks. The time spent on compiling the method is also a good approximation.

6.4.6 Discussion

By definition, the replaying compiler uses exactly the same set of the options for compiling a method as the state-saving compiler used, except an additional option for diagnostic output. However, in some cases the support personnel may want to replay a compilation with a slightly different set of options in order to narrow down the cause of a problem. At a first glance, this would be impossible since the replaying compiler may now need to obtain the inputs which the state-saving compiler did not use and thus did not save. However, we can exploit the mechanism of default values described in Section 6.3.5. That is, we simply let the replaying compiler pick up the default value for an input when it fails to find in the log a value corresponding to the input.

The Java VM may unload classes and delete data structures associated with them from the main memory. As a result, the system dump will not include the data structures for unloaded classes, such as bytecode for methods. This means that the replaying compiler is unable to replay the compilation of a method in an unloaded class.

We could argue that this is not a serious issue, based on the following observation. A class is unloaded only when there is no reference to the class in the virtual machine. That is, there is no instance of the class, and no method of the class currently being executed. Assume that a system crash occurred and that it was actually caused by incorrectly compiled code for a method. We believe that such a method is very likely to have a stack frame, actively being executed.

We could also modify the state-saving compiler to store the data structures of a class into an external file when the class is unloaded. We could do so by generating a system dump at the time of unloading if we could unload classes as a batch process. The state-saving compiler also timestamps the logs for compilations so that the replaying compiler can properly associate the logs and the data structures for unloaded classes.

6.5 Implementation

This section describes our prototypes of a state-saving compiler and a replaying compiler. We implemented the prototype based on the J9 Java VM [Grcevski et al. 2004] and the TR JIT compiler [Stepanian et al. 2005; Sundaresan et al. 2006] for AIX.

6.5.1 State-Saving Compiler

Our state-saving compiler allocates a memory area as the log for each compilation of a method. This compiler compresses each log using zlib library[1]. The log works as if it were a cache, so that the compiler can avoid saving duplicated input that happens to be constant during the compilation. That is, even if the state-saving compiler tries to get a variable input multiple times, it actually gets the value only on the first access, and subsequent accesses get the value saved in the log.

The state-saving compiler associates each log with the address of the JIT-compiled code. This makes it possible to identify the log for a particular compilation, even if the method is compiled multiple times for different optimization levels and there are multiple compiled code blocks for the method. Our prototype does not support replay compilation for unloaded classes. To defer dealing with this complex issue, we simply disabled class unloading in our experiments.

[1] The zlib library is available at http://www.gzip.org/zlib/

6.5.2 Replaying Compiler

Our prototype restores a system dump into the same address as the state-saving process to avoid the overhead by the adjustment of pointer variables, as described in Section 6.4.4.

Our state-saving compiler creates a special data structure, called an anchor structure, so that the replaying compiler can find the important data from a system dump. The data structure has markers in its header and trailer, and contains its size, the version number of the state-saving compiler, and a pointer to the list of logs. The state-saving compiler manages all logs as a linked list, and stores the pointer to the head into the anchor. The anchor structure also saves the fixed input that can be determined when the Java VM is initialized, such as the system configuration.

The replaying compiler scans the markers in the restored system dump, which is a block of unstructured binary data, for finding the anchor structure. When the compiler finds a marker, it verifies the size and the version of the state-saving compiler. It then finds the pointer to the head of the log list, and scans the list for the log corresponding to the compilation to be replayed.

The anchor structure has another pointer to the address of the log for currently being compiled (called the current log). Since an incomplete log may crash the replaying compiler, the current log should not be accessible in the list of "complete" logs. While the JIT is compiling a method, the pointer holds the address of the current log, and clears it when the compilation has finished successfully. Using this pointer variable, the replaying compiler can tell if the system crashed during a JIT compilation.

6.6 Experimental Results

Using the prototypes of the state-saving and the replaying compilers described in Section 6.4, we measured size and time overhead for saving the input into logs. Various machine configurations and programs were used for the evaluation, as shown in Table 6.2 and Table 6.3. For all measurement, we used the exploitation of default values described in Section 6.4.5. The confidence-based filtering is not uses, unless otherwise is specified. This prototype forces the Java VM to always create a system dump when it terminates after executing the specified Java program (because none of the tested program crashes the Java VM).

Table 6.2. Configurations of the tested machines

	Machine-1	Machine-2	Machine-3
CPU	POWER3, single processor	POWER4, 4-way SMP	POWER3, 2-way SMP
RAM	768 Mbytes	8 Gbytes	768 Mbytes
OS	AIX 5.2L	AIX 5.2L	AIX 4.3.3

Table 6.3. Evaluated programs

Program	Description
mtrt, jess, compress, db, mpegaudio, jack, javac	Each of the programs included in the benchmarks suite SPECjvm98.
SPECjbb	The SPECjbb2000 benchmark.
xml parser	The operation of parsing a sample XML file using the XML parser for Java. The sample file is included in the package. The execution performance was measured by the elapsed time for parsing the sample file.
jigsaw	The operation to start the Jigsaw HTTP server release 2.2.5a, and load the default top page using a Web browser. The execution speed was not measured because this is an I/O bound program.

6.6.1 Replayed Compilation

Our prototype successfully reproduced the compilation for all of the methods that were compiled for these programs. We executed the state-saving and the replaying compilers in the same machine in each of the three tested machines. We verified that the replaying compiler reproduced the same compilation by generating the JIT compiled code at the same address as that of the state-saving compiler and by comparing the generated code with the code saved in the system dump.

In addition, we also verified that the replaying compiler successfully reproduced the compilation from the system dump generated by a different machine. The replaying compiler succeeded in replaying all six possible combinations of the machines listed in Table 6.2 to execute the state-saving compiler with the replaying compiler.

6.6.2 The Size of a Log

Table 6.4 shows the size of the logs and the size of the diagnostic output for each program. We used neither compression nor the confidence-based filtering for this measurement. This result shows that the size of the diagnostic output is too large to save in memory, and it is also too large to save on disk because the overhead to save so much data on disk during the execution of the application program

Table 6.4. Comparison in size: diagnostic output vs. log

Program	Diagnostic output [MB]	Log [MB] (not compressed or filtered)
mtrt	378	0.054
jess	243	0.077
compress	80	0.011
db	83	0.019
mpegaudio	200	0.045
jack	360	0.075
javac	588	0.258
SPECjbb	1033	0.222
xml parser	101	0.030
jigsaw	136	0.056
Geo. mean	227	0.056

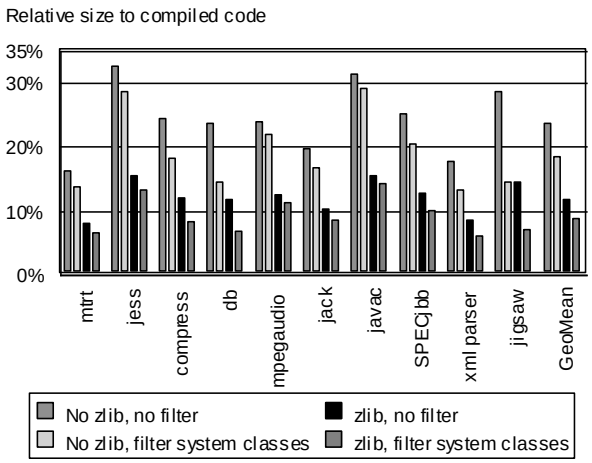


Figure 6.5. Reduction in the size of log with zlib and filter

will slow down the program. The replay compilation technique reduced the size of the trace information so much that saving input can always be enabled even in a production environment.

Figure 6.5 shows how the total size of the log changes due to compression and filtering. The sizes are relative to the total size of the compiled code. The left two bars for each program show the results when no compression is used, and the right two bars show the results with compression using the zlib library. The bars labeled `no filter' (first and third) show the results when the compiler does not use the confidence-based filtering. The bars labeled `filter system classes' show the results when the compiler uses filtering of the system classes (the classes in the java.lang, java.util, java.math, and java.io packages).

The reduction in the log size by filtering the system classes was 22.7% and 25.4% without and with compression, respectively. (All percentages are geometric means.) Compression reduced the total size of the logs by approximately half, and the reduction made by the combination of compression and filtering was 62.9%. As a result, the geometric mean of the size of the logs was reduced to less than 10% of the compiled code, which was our initial target as being acceptable for many users.

Table 6.5 shows the number of logs when filtering the methods of the system classes is used or not used. It also shows the reduction in the numbers and the sizes of the logs by filtering. The reduction in size is the summary of Figure 6.5 when zlib is not used. The reduction in the number of

Table 6.5. Reduction in the number of logs

Program	No filter	Filter system classes	Reduction in size by filtering (no zlib)
mtrt	153	113 (-26%)	-16%
jess	153	104 (-32%)	-12%
compress	39	22 (-44%)	-25%
db	59	22 (-63%)	-38%
mpegaudio	161	141 (-12%)	-9%
jack	201	141 (-30%)	-16%
javac	604	514 (-15%)	-7%
SPECjbb	502	345 (-31%)	-20%
xml parser	78	44 (-44%)	-25%
jigsaw	160	64 (-60%)	-49%
Geo. mean	152	94 (-38%)	-23%

Table 6.6. Comparison in size: system dump vs. Java heap

Program	System dump [MB]	Java heap [MB]
mtrt	334	256
jess	331	256
compress	330	256
db	330	256
mpegaudio	331	256
jack	331	256
javac	336	256
SPECjbb	413	256
xml parser	124	64
jigsaw	212	128

methods was 38%, but the reduction in the size of the logs was only 23%. We think this is because many of the filtered methods are smaller than average methods.

The total size of the log was less than 0.1% of the total memory usage of the Java VM process for these programs when both filtering and compression are used. This is because the size of the Java heap is much larger. Thus, for the measured programs, filtering and compression may not be needed for some users because the total size is still less than about 0.3% of the total memory usage. However, the number of compiled methods will increase in large commercial applications, such as Web application servers, and thus the ratio of the size of the compiled code will be higher than for the measured programs. For such programs, it is important to keep the total size of the log as small as possible by applying filtering and compression techniques. We also think it is still beneficial to keep the size of the log much smaller than the size of the JIT compiled code, because the compiled code is used to directly benefit the user by improving the execution performance of the programs. However, the log is used only for improving debuggability, while it reduces the available memory size for the user programs regardless of whether or not a problem occurs.

Table 6.6 shows the sizes of the system dumps and the heap sizes of the Java VM that created the system dumps. A system dump contains all of the data areas in the process memory, such as the Java heap, the Java stack, the native heap, the native stacks of all threads, and the JIT compiled code

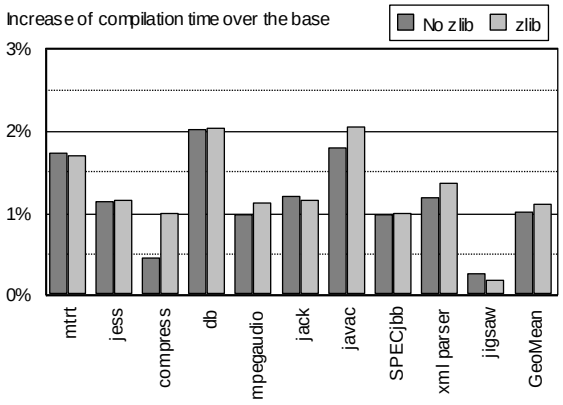


Figure 6.6. Increase in compilation time without and with zlib compression

blocks, though the largest part is the Java heap. Note that the JIT compiler usually does not directly use the values in Java objects for compiling a method, but uses the values collected by the runtime profiler. Thus, a large portion of this large system dump is not needed by the replaying compiler. Although a large system dump requires a large free space on disk, this is considered to be acceptable in many production environments because the system dump is normally used for debugging problems that occurred in those environments.

6.6.3 Compilation Time

Figure 6.6 shows how much the compilation time is increased over the base compiler when zlib is used. Each bar labeled `no zlib' shows the compilation time when no compression is used, whereas each bar labeled `zlib' shows the compilation time when zlib compression is used. Since the set of methods compiled in an execution of the program has changed on every execution due to the non-determinism, for fair comparison, we accumulated the elapsed time of the compilations that performed the same optimizations in all executions for the base, `no zlib', and `zlib'.

The increase in compilation time was up to 2.0% in both cases. The geometric mean of the increase was about 1.0% and 1.1% for `no zlib' and `zlib', respectively. This very small increase in compilation time indicates that the time to save the input to the logs was negligible.

The increase in compilation time in the replaying compiler over the base was 9.4%, as a geometric mean, when the diagnostic output was not generated. When the diagnostic output was generated, most of the compilation time is used for writing hundreds of megabytes of text to disk, regardless of whether or not the replay compilation technique is used.

6.6.4 Execution Speed

The operations to save the input into logs is the only additional overhead in the state-saving compiler against the base compiler because, for the same inputs, the state-saving compiler applies the same compilation and generates the same compiled code as the base compiler. There is no additional overhead in the compiled code.

Since the increase in compilation time was small, the slowdown of execution speed was also small. The geometric mean of the slowdown was only 1%.

6.7 Related Work

Many techniques for content-based approach have been proposed, but they caused large overhead for recording. Recap [Pan and Linton 1988] records the input for a program. However, it is estimated to generate 1 Mbyte of trace per second, even on a slow VAX-11/780 machine, and a faster machine could generate an unacceptably huge amount, such as 1 Gbyte per second. The jRapture system [Steven et al. 2000] records the parameters and the return values of a Java API that interacts with the underlying system. However, their prototype was three to ten times slower than normal execution. The idea of the content-based approach is simple and easy to adopt, but it tends to cause prohibitively large overhead in size and speed for use in practical systems. Using a system dump, we successfully minimized the overhead of recording the information required for replaying the JIT compiler.

The first error data capture (FEDC) concept [Koerner et al. 2004] also aims to improve the debuggability of a mainframe system used in a production environment. It uses special hardware, firmware, and software that continuously record information about each component in the system by using a separate computer. When an error occurs, support personnel can analyze the recorded information to debug the error. This is an effective approach for problem determination without replaying the system, but it needs special support by hardware and firmware.

Non-determinism in execution can also cause problems in performance analysis. For example, we cannot discriminate between the causes of performance improvements because of the non-determinism. OOR [Huang et al. 2004] and PEP [Bond and McKinley 2005] solved this problem by using advice files produced by the JIT compiler in the previous best run. Those files record the compilation level and the results of profilers, and they are used by the complier in a performance measurement run.

Their methodology was implemented in Jikes RVM 2.4.0 as an experimental feature[1], and many studies [Bond et al. 2007; Georges et al. 2008; Garner et al. 2011] use this feature to reduce performance non-determinism caused by the adaptive compilation. Georges et al. investigated possibility to further reduce performance non-determinism by using multiple compilation plans and statistical data analysis [Georges et al. 2008].

6.8 Summary

We have proposed a new approach, called replay JIT compilation, to reproduce the same JIT compilation offline and remotely by using two compilers, the state-saving compiler and the replaying compiler. The state-saving compiler is used in a normal run, and, while compiling a method, records into a log all of the runtime information referenced during the compilation. The log is in the main memory, and automatically included in the system dump when the application crashes. The replaying compiler is then used in a debugging run with the system dump, to recompile a method with the options for diagnostic output.

We developed our prototype based on the J9 Java VM and the TR JIT compiler for AIX and showed that the prototype successfully reproduces the same compilations done by the state-saving compiler. We also developed confidence-based filtering to save only the logs that are likely to be needed for debugging. The time overhead of running the state-saving compiler was only 1% and the size overhead for saving states was only 10% of the compiled code.

[1] Detailed description is available at http://www.jikesrvm.org/UserGuide/ExperimentalGuidelines/index.html

7. Conclusion

This research discussed technologies to improve the overall performance of production systems that use mixed-mode VMs, such as Java VMs, to run application programs. The overall performance of a production system can be increased by improving the performance of a single mixed-mode VM and by using a larger number of mixed-mode VMs to run many application programs in a machine. In addition, ease of debugging is also important for production systems because downtime hurts the customers' business.

We proposed three technologies to: a) improve the performance of a single VM, b) increase the number of VMs runnable in a system by reducing the memory usage of a JIT compiler, and c) implement a replay debugging feature in the JIT compiler with a small overhead. We conclude this dissertation by summarizing our proposals and contributions.

Improving the Performance of a Mixed-mode Bytecode Interpreter

There are three causes of overhead in a naïve bytecode interpreter: frequent execution of indirect branches, low instruction-level parallelism (ILP), and redundant memory accesses for fetching bytecode instructions. The threaded code interpreter is a common technique for reducing the overhead of indirect branches and low ILP.

This research showed that we can improve the performance of a bytecode interpreter by reducing the redundant memory accesses for fetching bytecode, and we proposed a technique to reduce the overhead by avoiding misaligned loads and caching the loaded words in registers. A bytecode instruction set usually takes a variable length format, and thus, the beginning of a bytecode instruction is not necessarily aligned to a word boundary. A modern processor, however, always accesses memory in units of a word and handles misaligned loads by performing two word-aligned loads internally. This operation increases the amount of additional memory accesses for fetching bytecode.

Our contributions to the mixed-mode bytecode interpreter are:

1. We proposed a technique to reduce the amount of memory accesses for fetching bytecode instructions by always aligning the memory accesses to a word boundary and avoiding duplicate memory loads in a processor for handling misaligned loads. The technique uses

multiple versions of the handler of the bytecode, and each of them is customized to a byte position in a word, so the handler can extract bytes without runtime checking the current byte position of the bytecode.

2. We proposed a technique to implement speculative decoding for a superscalar processor. This technique reserves a register for each byte position in a word to hold the speculatively decoded result. The customized bytecode handlers reuse speculatively decoded results without copying them because the byte position of a speculatively decoded byte in a word is the same regardless of the position of the currently executing bytecode.

Reducing the Memory Overhead of the JIT Compiler

Increasing the number of Java VMs runnable in a system is a common approach to improving the overall performance of a machine. Here, we need to reduce the memory usage of each Java VM; otherwise, running many Java VMs may cause thrashing and hurt performance.

This research gave a detailed breakdown of memory usage in a Java VM and pointed out that the JIT compiler can cause memory inefficiency because its memory allocation pattern does not fit the LRU policy. The JIT compiler's work areas stay in memory even after they have been deallocated and neither memory management APIs nor kernel functions can free up this memory without explicit reclamation.

Our contributions to reducing the memory overhead of JIT compilers are:

1. We showed that Java memory usage other than that for the Java heap is too large to ignore, through a detailed analysis of memory usage in a Java VM running an enterprise Web application. We summarized the memory usage by accumulating the memory allocated by each of the Java VM components.

2. We showed that there is a memory allocation and deallocation pattern that can increase the amount of memory in the free list.

3. We proposed a technique to reduce physical memory usage by advising the OS to release physical pages for the memory areas in the `malloc`'s free list. Although the pages in the free list are less important, the OS will keep them in memory and swap out other pages

first. The LRU policy does not recognize less important pages because those pages are used until they are deallocated. This technique avoids thrashing in high-memory-pressure situations and improves performance.

Improving Debugging Functionality with Small Overhead

Minimizing service downtime helps to make enterprise systems more effective. Reducing the time it takes to debug problems is a key issue to reducing downtime. On the other hand, it is also important to keep the performance of production systems at a high level. The trouble is that these two issues tend to be a trade-off because increasing the debugging capability usually degrades performance.

This research proposed a technique to regenerate debug traces of a JIT compiler with a small overhead. We focused on debugging a JIT compiler because a bug in the compiler often causes hard-to-debug problems in the generated machine code and leads to long downtimes. We focused on regenerating debug traces of the JIT compiler because such traces include invaluable information for debugging. Our technique records the JIT compiler's operation during a production run and exactly replays it offline by using the recorded information. It reduces overhead to as small as it can be in production environments. We used a process memory dump of a Java VM, which is created when a Java process crashes or when a developer requests, and hence, we can avoid recording data whose values do not change during execution.

Our contributions to improving the debugging function are:

1. We proposed a technique to record the input to the JIT compiler during production runs with a small overhead. The size of the recorded data is only about 10% of the size of the code generated by the compiler. We recorded only the input to the JIT compiler whose values can be changed during execution. For inputs whose values remain constant, we used a process memory dump to record them.
2. We presented a list of commonly used input to a JIT compiler and categorized what information can change its value during an execution and which do not. The latter can be saved using a process memory dump.

By combining these three techniques, this dissertation has contributed to improving the overall performance of enterprise application programs written in the programming language using mixed-mode VMs.

Bibliography

[Agesen and Detlefs 2000]

Ole Agesen and David Detlefs. *Mixed-mode Bytecode Execution*. Sun Microsystems, Inc. Technical Report TR-2000-87. 2000.

[Alpern et al. 2000]

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*, Vol. 39(1), pp. 211–238. 2000.

[Ammann 1977]

Urs Ammann. On code generation in a PASCAL compiler. *Software: Practice and Experience*, Vol. 7(3), pp. 391–423. June/July 1977.

[Arnold et al. 2000]

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00)*, pp. 47–65. 2000.

[Arnold and Ryder 2001]

Matthew Arnold and Barbara G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pp. 168–179. 2001.

[Bacon 1991]

David F. Bacon. Hardware-Assisted Replay of Multiprocessor Programs. In *Proceedings of 1991 ACM/ONR Workshop on Parallel and Distributed Debugging (PADD '91)*, pp. 194–205. 1991.

[Bell 1973]

James R. Bell. Threaded Code. *Communications of the ACM*, Vol. 16(6), pp. 370–372. 1973.

[Berndl et al. 2005]

Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context Threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the international symposium on Code generation and optimization (CGO '05)*, pp. 15–26. 2005

[Blackburn et al. 2006]

S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*, pp. 169–190. 2006.

[Bodoff et al. 2004]

Stephanie Bodoff, Eric Armstrong, Jennifer Ball, Debbie Carson, Ian Evans, Dale Green, Kim Haase, Eric Jendrock. *The J2EE Tutorial, 2nd Edition*. Prentice Hall 2004, ISBN 0-321-24575-X.

[Bolz et al. 2009]

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*, pp. 18–25. 2009.

[Bond and McKinley 2005]

Michael D. Bond and Kathryn S. McKinley. Continuous Path and Edge Profiling. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '05)*, pp. 130–140. 2005.

[Bond et al. 2007]

Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors. In *Proceedings of Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07)*, pp. 405–422. 2007.

[Chambers and Ungar 1989]

Craig Chambers and David Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation (PLDI '89)*, pp. 146–160. 1989.

[Chan et al. 1998]

Patrick Chan, Rosanna Lee, and Douglas Kramer. *The Java Class Libraries, Second Edition, Volume 1*. Addison-Wesley. 1998, ISBN 0-201-31002-3.

[Chawla 2003]

Sumit Chawla. *Getting more memory in AIX for your Java applications*. 2003.
http://www.ibm.com/developerworks/systems/articles/aix4java1.html

[Choi and Srinivasan 1998]

Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '98)*, pp. 48–59. 1998.

[Cierniak et al. 2003]

Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, James Stichnoth. The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, Vol. 07, Issue 01, pp. 5–18. 2003.

[Cooper et al. 1992]

Eric Cooper, Scott Nettles, and Indira Subramanian . Improving the performance of SML garbage collection using application-specific virtual memory management. In *Proceedings of the 1992 ACM conference on LISP and functional programming (LFP'92)*, pp. 43–52. 1992.

[Deutsch and Schiffman 1984]

L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '84)*, pp. 297–302. 1984.

[Dewar 1975]

Robert B. K. Dewar. Indirect Threaded Code. *Communications of the ACM*, Vol. 18(6), pp. 330–331. 1975.

[ECMA 2006]

ECMA International. Standard ECMA-334 4th Edition. C# Language Specification. June 2006. http://www.ecma-international.org/publications/standards/Ecma-334.htm

[ECMA 2012]

ECMA International. Standard ECMA-335 6th Edition. Common Language Infrastructure (CLI). June 2012. http://www.ecma-international.org/publications/standards/Ecma-335.htm

[Ertl 1995]

M. Anton Ertl. Stack Caching for Interpreters. In *Proceeding of SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI '95)*, pp. 315–327. 1995.

[Ertl and Gregg 2001]

M. Anton Ertl and David Gregg. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. Euro-Par 2001. *LNCS 2150*, pp. 403–412. 2001.

[Ertl and Gregg 2003]

M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI '03)*, pp. 278–288. 2003.

[Ertl and Gregg 2004]

M. Anton Ertl and David Gregg. Combining Stack Caching with Dynamic Superinstructions. In *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators (IVME '04)*, pp. 7–14. 2004.

[Feng and Berger 2005]

Yi Feng and Emery D. Berger. A Locality-Improving Dynamic Memory Allocator. In *Proceedings of the 2005 workshop on Memory system performance (MSP '05)*, pp. 68–77. 2005.

[Garner et al. 2011]

Robin J. Garner, Stephen M. Blackburn, and Daniel Frampton. A Comprehensive Evaluation of Object Scanning Techniques. In *Proceedings of the international symposium on Memory management (ISMM '11)*, pp. 33–42. 2011.

[Georges et al. 2008]

Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java Performance Evaluation through Rigorous Replay Compilation. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA '08)*, pp. 367–384. 2008

[Gloger 2006]

Wolfram Gloger. Wolfram Gloger's malloc homepage. 2006.

http://www.malloc.de/en/

[Gosling and McGilton 1996]

James Gosling and Henry McGilton. *The Java Language Environment*. White Paper. 1996.

http://java.sun.com/docs/white/langenv/

[Gosling et al. 1996]

James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley. 1996. ISBN 0-201-63451-1.

[Grcevski et al. 2004]

Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Proceedings of the Third Virtual Machine Research and Technology Symposium (VM '04)*, pp. 151–162. 2004.

[Gregg and Ertl 2001]

David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an Efficient Java Interpreter. HPCN Europe 2001. *LNCS 2110*, pp. 613–620. 2001.

[Hanik 2007]

Filip Hanik. Inside the Java Virtual Machine. 2007.

http://www.springsource.com/files/Inside_the_JVM.pdf

[Hertz et al. 2005]

Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage Collection Without Paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*, pp. 143–153. 2005.

[Hoogerbrugge et al. 1999]

Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A Code Compression System Based on Pipelined Interpreters. *SOFTWARE — Practice and Experience*, Vol. 29(11), pp. 1005–1023. September 1999.

[Hoogerbrugge and Augusteijn 2000]

Jan Hoogerbrugge and Lex Augusteijn. Pipelined Java Virtual Machine Interpreters. CC/ETAPS 2000. *LNCS 1781*, pp. 35–49. 2000.

[Horie et al. 2014]

Michihiro Horie, Kazunori Ogata, Kiyokuni Kawachiya, and Tamiya Onodera. String deduplication for Java-based middleware in virtualized environments. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*, pp. 177–188. 2014.

[Huang et al. 2004]

Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The Garbage Collection Advantage: Improving Program Locality. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pp. 69–80. 2004.

[IBM 1994]

IBM Corporation. *The PowerPC Architecture: a specification for a new family of RISC processors*. ISBN 1-55860-316-6. August 1994.

[IBM and Motorola 1998]

IBM Corporation and Motorola Inc. *PowerPC 604e RISC Microprocessor User's Manual with Supplement for PowerPC 604 Microprocessor*. G522-0290-00. March 1998.

[IBM 2007]

IBM Corp. *Power ISA, Version 2.05*. October 2007.

[IBM 2012]

IBM Corporation. *IBM Developer Kit and Runtime Environment, Java Technology Edition, Version 6. Diagnostics Guide.* Twelfth Edition. November 2012.

http://www.ibm.com/developerworks/java/jdk/diagnosis/

[IEEE and The Open Group]

The IEEE and The Open Group. *Single UNIX Specification, Version 4*, 2013 Edition.

https://www2.opengroup.org/ogsys/catalog/T101

[Intel 2002]

Intel Corporation. *Intel Itanium Architecture Software Developer's Manual, Volume 3: Instruction Set Reference, Revision 2.1*. Document Number: 246319-004. October 2002.

[Intel 2003]

Intel Corporation. *Intel PXA255 Processor Developer's Manual*. Order Number: 278693-001. March 2003.

[Intel 2009]

Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture*. Order Number: 253665-033US. 2009.

[IPA 2010]

IPA Ruby Standardization WG. Programming Languages — Ruby (Final draft for the Ruby ISO standard ISO/IEC 30170:2012). 2010.

[Ishizaki et al. 1999]

Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. In *Proceedings of the ACM 1999 Java Grande Conference*. pp. 119–128. 1999.

[Ishizaki et al. 2000]

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 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pp. 294–310. 2000.

[ISO/IEC 1997]

The International Organization for Standardization and the International Electrotechnical Commission. Information technology - Programming languages - Forth. First edition (ISO/IEC 15145). Aprl 15, 1997.

[Kawachiya et al. 2008]

Kiyokuni Kawachiya, Kazunori Ogata, and Tamiya Onodera. Analysis and Reduction of Memory Inefficiencies in Java Strings. In *Proceedings of the 23rd ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '08)*, pp. 385–401. 2008.

[Koerner et al. 2004]

S. Koerner, R. Bawidamann, W. Fischer, U. Helmich, D. Koldt, B. K. Tolan, and P. Wojciak. The z990 first error data capture concept. *IBM Journal of Research and Development*, Vol. 48(3/4), pp. 557–568. May 2004.

[Kotzmann et al. 2008]

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 5, No. 1, pp. 7:1–7:32. 2008.

[Landin 1964]

Peter J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, Vol. 6(4), pp. 308–320. 1964.

[Lea 2000]

Doug Lea. A Memory Allocator. 2000.

http://g.oswego.edu/dl/html/malloc.html

[LeBlanc and Mellor-Crummey 1987]

Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, Vol. C-36(4), pp. 471–482. April 1987.

[Liang and Viswanathan 1999]

Sheng Liang and Deepa Viswanathan. Comprehensive Profiling Support in the Java Virtual Machine. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*, pp. 229–242, 1999.

[Lin et al. 2012]

Yi Lin, Stephen M. Blackburn, and Daniel Frampton. Unpicking the Knot: Teasing Apart VM/Application Interdependencies. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12)*, pp. 181–190. 2012.

[Lindholm and Yellin 1996]

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification* (First edition). Addison-Wesley, 1996. ISBN 0-201-63452-X.

Online version was available at http://java.sun.com/docs/books/jvms/index.html

[Mathew et al. 1999]

J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM 1999 conference on Java Grande*, pp. 72–80. June 1999.

[McCandless and Gregg 2011]

Jason McCandless and David Gregg. Optimizing Interpreters by Tuning Opcode Orderings on Virtual Machines for Modern Architectures: Or: How I Learned to Stop Worrying and Love Hill Climbing. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*, pp. 161–170. 2011.

[Miller and Choi 1988]

Barton P. Miller and Jong-Deok Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*, pp. 135–144. 1988.

[Mitchell and Sevitsky 2003]

Nick Mitchell and Gary Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP '03)*, pp. 351–377. 2003.

[Mitchell and Sevitsky 2007]

Nick Mitchell and Gary Sevitsky. The Causes of Bloat, The Limits of Health. In *Proceedings of the 22nd ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '07)*, pp. 245–260. 2007.

[Nori et al. 1975]

K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi. *The Pascal P-Compiler: Implementation Notes*. Institut fur Informatik ETH, Zurich, Technical Report 10. 1975.

[O'Connel and White 2000]

F. P. O'Connel and S. W. White. POWER3: The next generation of PowerPC processors. *IBM Journal of Research and Development*, Vol. 44(6), pp. 873–884. November 2000.

[Ogata et al. 2002]

Kazunori Ogata, Hideaki Komatsu, and Toshio Nakatani. Bytecode fetch optimization for a Java interpreter. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS X)*, pp. 58–67. 2002.

DOI:http://dx.doi.org/10.1145/635506.605404

[Ogata et al. 2006]

Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. Replay compilation: improving debuggability of a just-in-time compiler. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06)*, pp. 241–252. 2006.

DOI:http://dx.doi.org/10.1145/1167515.1167493

[Ogata et al. 2010]

Kazunori Ogata, Dai Mikurube, Kiyokuni Kawachiya, Scott Trent, and Tamiya Onodera. A study of Java's non-Java memory. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*, pp. 191–204. 2010.

DOI:http://dx.doi.org/10.1145/1932682.1869477

[Oracle 2008]

Oracle and/or its affiliates. *Troubleshooting Guide for Java SE 6 with HotSpot VM*. November 2008. http://www.oracle.com/technetwork/java/javase/index-137495.html

[Oracle 2011]

Oracle and/or its affiliates. *Oracle JRockit Introduction Release R28*. Part Number E15058-05. 2011. http://docs.oracle.com/cd/E15289_01/doc.40/e15058/aboutjrockit.htm

[OSGi 2003]

The Open Service Gateway Initiative. *OSGi Service Platform Release 3*. IOS Press. ISBN 1-58603-3115. March 2003.

[Paleczny et al. 2001]

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)*, pp. 1–12. April 2001.

[Pan and Linton 1988]

Douglas Z. Pan and Mark A. Linton. Supporting Reverse Execution of Parallel Programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD '88)*, pp. 124–129. 1988.

[Peng et al. 2004]

Jinzhan Peng, Gansha Wu, and Guei-Yuan Lueh. Code Sharing among States for Stack-Caching Interpreter. In *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators (IVME '04)*, pp. 15–22. 2004.

[Poddar and Minshall 2006]

Indrajit Poddar and Robbie John Minshall. *Memory leak detection and analysis in WebSphere Application Server: Part 1: Overview of memory leaks.*

http://www.ibm.com/developerworks/websphere/library/techarticles/0606_poddar/0606_poddar.html

[Radhakrishnan 2001]

Ramesh Radhakrishnan, Narayanan Vijaykrishnan, Lizy K. John, Anand Sivasubramaniam, Juan Rubio, and Jyotsna Sabarinathan. Java Runtime Systems: Characterization and Architectural Implications. *IEEE Transactions on Computers*, Vol. 50(2), pp. 131–146. February 2001.

[Ritter and Walker 1980]

Terry Ritter and Gregory Walker. Varieties of Threaded Code for Language Implementation. *BYTE*, Vol. 5(9), pp. 206–227. September 1980.

[Rohou et al. 2015]

Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch Prediction and the Performance of Interpreters: Don't Trust Folklore. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*, pp. 103–114. 2015.

[Romer 1996]

Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henny M. Levy. The Structure and Performance of Interpreters. In *Proceedings of the seventh international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pp. 150–159. 1996.

[Ronsse et al. 2000]

Michiel Ronsse, Koenraad D. Bosschere, and Jacques C. de Kergommeaux. Execution replay and debugging. In *Proceedings of the Fourth International Workshop on Automated and Algorithmic Debugging (AADEBUG '00)*. 2000.

[Sasada 2006]

Koichi Sasada, Yukihiro Matsumoto, Atsushi Maeda, and Mitaro Namiki. YARV : Yet Another RubyVM : The Implementation and Evaluation. Information Processing Society of Japan (IPSJ), *Transactions on programming*, Vol. 47(SIG_2(PRO_28)), pp. 57–73. 2006.

[Schatzl et al. 2011]

Thomas Schatzl, Laurent Daynès, and Hanspeter Mössenböck. Optimized Memory Management for Class Metadata in a JVM. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*, pp. 151–160. 2011.

[Sewe et al. 2011]

Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*, pp. 657–676. 2011.

[Silberschatz 2012]

Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts—Update, Eighth Edition*. 2012. ISBN 978-1-118-11273-1.

[Smaragdakis et al. 1999]

Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling Computer Systems (SIGMETRICS '99)*, pp. 122–133. 1999.

[Smith and Nair 2005]

Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, ISBN 1558609105. June 2005.

[Stepanian et al. 2005]

Levon Stepanian, Angela D. Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodly. Inlining Java Native Calls at Runtime. In *Proceedings of ACM/Usenix International Conference on Virtual Execution Environments (VEE '05)*, pp. 121–131. 2005.

[Steven et al. 2000]

John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A Capture/Replay Tool for Observation-Based Testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA '00)*, pp. 158–167. 2000.

[Suganuma et al. 2000]

Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java Just-In-Time Compiler. *IBM Systems Journal*, Java Performance Issue, Vol. 39(1), pp. 175–193. February 2000.

[Suganuma et al. 2001]

Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *Proceeding of Object-Oriented Programming, System Languages, and Application (OOPSLA '01)*, pp. 180–194. 2001.

[Sun Microsystems 2001]

Sun Microsystems. *The Java HotSpot Virtual Machine Technical White Paper*. May 2001.

[Sun Microsystems 2002]

Sun Microsystems. *The Java HotSpot Virtual Machine, v1.4.1, d2*. A Technical White Paper. September 2002.

http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html

[Sun Microsystems 2008a]

Sun Microsystems. HPROF - Heap Profiler. *Troubleshooting Guide for Java SE 6 with HotSpot VM*, pp. 26–31. November 2008.

[Sun Microsystems 2008b]

Sun Microsystems. jhat Utility. *Troubleshooting Guide for Java SE 6 with HotSpot VM*, pp. 39–44. November 2008.

[Sundaresan et al. 2006]

Vijay Sundaresan, Daryl Maier, Pramod Ramarao, and Mark Stoodley. Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*, pp. 87–97. 2006.

[Talluri and Hill 1994]

Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems (ASPLOS-VI)*, pp. 171–182. 1994.

[Ungar and Smith 1987]

David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87 Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA '87)*, pp. 227–242. 1987.

[Wang et al. 2002]

Zhenlin Wang, K. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*, pp. 199–208. 2002.

[Whaley 2000] John Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of ACM 2000 Java Grande Conference*, pp. 78–87. 2000.

[Wimmer et al. 2013]

Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An Approachable Virtual Machine For, and In, Java. *ACM Transactions on Architecture and Code Optimization (TACO)* – Special Issue on High-Performance Embedded Architectures and Compilers, Vol. 9(4), Article No. 30. 2013.

[Yasue et al. 2003]

Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers. In *Proceedings of the Twelfth International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, pp. 148–158. 2003.

[Yermolovich et al. 2009]

Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of Dynamic Languages Using Hierarchical Layering of Virtual Machines. In *Proceedings of the 5th symposium on Dynamic languages (DLS '09)*, pp. 79–88. 2009.

[Zaleski et al. 2007]

Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. YETI: a graduallY Extensible Trace Interpreter. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE '07)*, pp. 83–93. 2007.