Compile-time Optimization and Customization in Java and C + MPI

Hirotaka Ogawa

A Dissertation Submitted to the Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Japan

In Partial Fulfillment of the Requirements for the Degree of Doctor of Science

March, 2009
Abstract

To achieve both of runtime efficiency and maintainability of large-scale and complicated software especially with dynamicity, portability, and concurrency, programmers have strong desire to customize and optimize their software as they need. For example, they may need to perform platform-specific optimizations so as to adapt their software from embedded systems and hand-held devices all the way up to large-scale servers and massively parallel processors (MPPs). Also, they may need to perform application-specific optimizations such as selection of compilation strategies, runtime partial evaluation, as well as application-specific idiom recognition. However, such customizations and/or optimizations generally require complicated program transformation which could only be done by highly-experienced programmers or compiler experts. These situations lead us to the necessity to develop more flexible approaches to make users-demanded customizations and optimizations easier.

From such a perspective, this thesis focuses on two different kinds of runtime environments, namely Java and C + MPI, which could be problematic to achieve both of runtime efficiency and maintainability of large-scale and complicated software. Java is a portable and dynamic programming language which has a runtime environment which employs portable intermediate representations named Java bytecode and utilizes Just-In-Time (JIT) compilers, which compile (parts of) programs into native code at run-time. However, all the Java JIT compilers today only largely focus on standard platforms such as Workstations and Desktop PCs, and stress optimizing for speeding up single-threaded execution of general programs. As a result, Java programmers have neither effective measures to customize their programs for embedded systems, nor to optimize their programs by employing any advanced compilation strategies. C + MPI, on the other hand, is gaining acceptance as a standard for message-passing in high-performance computing, due to its powerful and flexible support of various communication styles. C + MPI is used not only for writing new parallel applications, but also as an effective tool for parallelizing existing applications, as well as serving as an underlying message-passing library for implementations of parallel languages, such as High Performance Fortran (HPF). However, the complexity of its API poses significant software overhead, and C + MPI
programmers have no opportunity to remove such overhead. As a result, applicability of MPI has been restricted to rather regular, coarse-grained, and computation-intensive applications.

From the viewpoint of software engineering, one solution to this problem is known to employ a “reflective” open compiler or a compiler framework and perform program transformation, but its practical effectiveness has still been unclear. In the case of Java, most JIT compilers today are typically written in C or C++ and architected to be closed and monolithic, and do not facilitate interfaces, frameworks, nor patterns as a means of customization. And as for C + MPI, as far as we know, there is no preceding work which adapts current compiler frameworks such as Standford SUIF to meet the practical needs of low-latency, high-performance message-passing libraries.

In order to resolve these situations, we have developed two different practical systems; namely OpenJIT for Java and OMPI (Optimizing MPI) for C + MPI. OpenJIT is an open-ended, reflective JIT compiler framework for Java written almost entirely in Java itself. Although in general self-descriptive systems have been studied in various contexts such as reflection and interpreter/compiler bootstrapping, OpenJIT is a first system we know to date that offers a stable, full-fledged Java JIT compiler that plugs into existing monolithic JVMs, and offer competitive performance to JITs typically written in C or C++. At the same time, OpenJIT is designed to be a compiler framework in the sense of Stanford SUIF, in that it facilitates high-level and low-level program analysis and transformation framework for the users to customize. And, OMPI system is a compile-time optimizer for C + MPI programs based on Stanford SUIF compiler framework, which eliminates much of the communication overhead using partial evaluation techniques which exploit static information of MPI calls in the target programs. Because partial evaluation alone is insufficient, we also utilize template functions for further optimization.

This thesis addresses the design and implementation of OpenJIT and OMPI system, as well as our practical experiences and technical contributions.
Acknowledgments

First and foremost, I am deeply indebted to my supervisor, Professor Satoshi Matsuoka. His long-term encouragement, support, and advice have been immensely valuable. And, I greatly thank my thesis committee, Professor Masataka Sassa, Professor Shigeru Chiba, Professor Takuo Watanabe, and Professor Kazuyuki Shudo.

I thank Kouya Shimura, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura, who joined OpenJIT project team. And, I am grateful to Matt Welsh, who coded parts of OpenJIT during his summer job at Fujitsu, and Ole Agesen, who discussed various technical issues on Java Just-in-Time compilers.

Matsuoka Laboratory (Tokyo Institute of Technology) and Grid Research Center / Information Technology Research Institute (National Institute of Advanced Industrial Science and Technology, Japan) provided ideal work environments for me to continue the work of this thesis. I am particularly grateful to Satoshi Sekiguchi, Satoshi Itoh, Yoshio Tanaka, Tomohiro Kudoh, Hidemoto Nakada, and Atsuko Takefusa.

Last, but by no means least, I owe a great debt to my wife, Yoko Ogawa, for her support, encouragement, and sense of humor; her companionship has been invaluable in all facets of my life.
# Contents

1 Introduction .................................................. 1
   1.1 OpenJIT .................................................. 3
      1.1.1 Motivation .......................................... 3
      1.1.2 Our Solution ........................................ 5
   1.2 OMPI .................................................... 6
      1.2.1 Motivation .......................................... 6
      1.2.2 Our Solution ........................................ 7
   1.3 Technical Contributions .................................. 8
   1.4 The Structure of This Thesis ............................. 9

2 Related Work ................................................ 13
   2.1 OpenJIT .................................................. 13
      2.1.1 Virtual Machines and JIT Compilers ............... 13
      2.1.2 Self-descriptive Object-Oriented Language Systems 17
      2.1.3 Compiler Frameworks Targeted to Java ............ 21
      2.1.4 Reflective Systems Targeted to Java .............. 23
      2.1.5 Other Related Work ................................. 26
   2.2 OMPI .................................................... 28
      2.2.1 Message-Passing Interface (MPI) ................. 28
      2.2.2 Hardware-specific Implementations ............... 29
      2.2.3 General Implementations ........................... 30
      2.2.4 Implementations for Grid ......................... 31
      2.2.5 Automatic Tuning Libraries ....................... 31
   2.3 Our Impact on Succeeding Work ........................ 32
      2.3.1 OpenJIT ............................................. 32
      2.3.2 OMPI ................................................. 35

3 Design and Implementation of OpenJIT ...................... 39
   3.1 Overview of OpenJIT ..................................... 39
      3.1.1 Conceptual Overview ............................... 40
      3.1.2 Architectural Overview ............................ 41
   3.2 OpenJIT Frontend System ................................ 43
      3.2.1 Overview ........................................... 43
CONTENTS

3.2.2 Classfile Annotation ................................................. 44
3.2.3 OpenJIT Bytecode Decompiler Module ................................. 45
3.2.4 OpenJIT Class Annotation Analyzer Module .......................... 46
3.2.5 OpenJIT High-level Optimizer Toolkit ............................... 47
3.2.6 Abstract Syntax Tree Package ....................................... 48
3.3 Application-Specific Optimization using OpenJIT Frontend .......... 50
3.3.1 Benchmarking Environment ........................................... 52
3.3.2 Benchmarking Result ................................................ 52
3.4 OpenJIT Backend System ............................................... 55
3.4.1 Overview .......................................................... 55
3.4.2 Technical Challenges in a Reflective Java JIT Compiler ........ 56
3.5 Chapter Summary ...................................................... 66

4 Performance Analysis of OpenJIT ........................................ 69
4.1 Benchmarking Environment ............................................. 70
4.2 Benchmarking Contents ................................................ 72
4.2.1 The Size of OpenJIT ................................................ 72
4.2.2 Baseline Performance ................................................. 72
4.2.3 Bootup Performance and Heap Behavior ............................... 76
4.3 Adaptive OpenJIT Compilation Strategies ............................... 85
4.4 Chapter Summary ...................................................... 90

5 OMPI: Optimizing C + MPI Programs .................................... 91
5.1 Overview .......................................................... 91
5.2 Analysis of Software Overhead in MPI .................................. 93
5.2.1 General Overhead .................................................... 93
5.2.2 MPI-Specific Overhead .............................................. 94
5.2.3 Opportunities for Eliminating Overhead ............................... 95
5.3 Optimizing MPI Programs Using Partial Evaluation .................. 97
5.3.1 Architectural Overview of OMPI .................................... 97
5.3.2 Static Analysis and Partial Evaluation Using SUIF ................ 98
5.3.3 Optimization Passes ................................................ 98
5.3.4 An Example of MPI Optimization .................................. 102
5.4 Performance Measurements ............................................ 104
5.4.1 Comparison against Dynamic Caching Optimization ............. 104
5.4.2 OMPI Implementation on AP1000 .................................. 105
5.4.3 Baseline Performance ................................................. 108
5.4.4 Numerical Application Core Benchmarks ............................ 109
5.5 Chapter Summary ...................................................... 111

6 Conclusion .......................................................... 113
6.1 Technical Contributions ................................................ 114
6.2 Future Directions ...................................................... 118
List of Tables

2.1 Self-descriptive object-oriented Java VM implementations 20
2.2 Compiler frameworks targeted to Java 23
2.3 Reflective systems targeted to Java 27
3.1 Results of OpenJIT frontend optimization 55
4.1 SPEC JVM98 benchmark information 70
4.2 Code size of OpenJIT and C runtimes 72
4.3 Heap behavior statistics for Hello 76
4.4 Heap behavior statistics for _201_compress 76
4.5 Heap behavior statistics for _202_jess 77
4.6 Heap behavior statistics for _209_db 77
4.7 Heap behavior statistics for _213_javac 77
4.8 Heap behavior statistics for _227_mtrt 78
4.9 Heap behavior statistics for _228_jack 78
4.10 Alteration of JIT compilation frequency for all methods 87
4.11 Alteration of JIT compilation frequency for OpenJIT methods only 88
5.1 List of building template functions for MPI_Send 107
5.2 List of building template functions for MPI_Recv 107
A.1 Essential hook functions 139
A.2 Other runtime functions 169
List of Figures

3.1 Comparison of traditional JIT compilers and OpenJIT ............. 40
3.2 Overview of the OpenJIT frontend system .......................... 44
3.3 Class hierarchy of the Abstract Syntax Tree Package .............. 49
3.4 Expression class for the “Multiply” expression (binary expres-
sion) ............................................................... 50
3.5 Statement class for the “If” statement .............................. 51
3.6 Overview of LoopTransformer ..................................... 53
3.7 Matrix multiply method (original) .................................. 54
3.8 Matrix multiply method (transformed) ............................. 54
3.9 Overview of the OpenJIT backend system ......................... 57
3.10 Invoking OpenJIT .................................................. 59
3.11 Constant pool introspection methods ............................. 64
3.12 Native method allocation and reflection .......................... 64
3.13 Class resolution methods (used for inlining) .................... 65

4.1 SPEC JVM98 on Solaris (SPARC) ................................. 74
4.2 SPEC JVM98 on Linux (x86) ....................................... 74
4.3 SciMark 2.0 on Solaris (SPARC) .................................. 75
4.4 SciMark 2.0 on Linux (x86) ......................................... 75
4.5 Timeline behavior of heap usage and live object allocated by
   interpreter ................................................................. 79
4.6 Timeline behavior of heap usage and live object allocated by
   sunwjit ............................................................... 79
4.7 Timeline behavior of heap usage and live object allocated by
   OpenJIT ............................................................ 80
4.8 Timeline behavior of heap usage and live object allocated by
   OpenJIT-int ........................................................ 80
4.9 Timeline behavior of heap usage with SPEC JVM98 compress
   [sunwjit vs. openjit] ............................................. 81
4.10 Timeline behavior of heap usage with SPEC JVM98 jess (sun-
    wjit vs. openjit) ................................................. 81
4.11 Timeline behavior of heap usage with SPEC JVM98 db (sun-
    wjit vs. openjit) ................................................. 82
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.12</td>
<td>Timeline behavior of heap usage with SPEC JVM98 javac (sunwjit vs. openjit)</td>
<td>82</td>
</tr>
<tr>
<td>4.13</td>
<td>Timeline behavior of heap usage with SPEC JVM98 mtrt (sunwjit vs. openjit)</td>
<td>83</td>
</tr>
<tr>
<td>4.14</td>
<td>Timeline behavior of heap usage with SPEC JVM98 jack (sunwjit vs. openjit)</td>
<td>83</td>
</tr>
<tr>
<td>4.15</td>
<td>Timeline behavior of heap usage for adaptive compilation for all methods (best cases)</td>
<td>89</td>
</tr>
<tr>
<td>4.16</td>
<td>Timeline behavior of heap usage for adaptive compilation for OpenJIT methods only (best cases)</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of MPI calls MPI_Send</td>
<td>94</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of MPI calls MPI_Recv</td>
<td>94</td>
</tr>
<tr>
<td>5.3</td>
<td>Overview of optimization passes</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>Multiple versions of building template functions</td>
<td>101</td>
</tr>
<tr>
<td>5.5</td>
<td>An example of MPI optimization via partial evaluation</td>
<td>103</td>
</tr>
<tr>
<td>5.6</td>
<td>Cache manager handle</td>
<td>106</td>
</tr>
<tr>
<td>5.7</td>
<td>Ping-Pong overhead result</td>
<td>108</td>
</tr>
<tr>
<td>5.8</td>
<td>Numerical application core benchmarks</td>
<td>110</td>
</tr>
<tr>
<td>A.1</td>
<td>Flow of OpenJIT compiler execution</td>
<td>139</td>
</tr>
<tr>
<td>A.2</td>
<td>Rewriting sequence of multiple instructions</td>
<td>162</td>
</tr>
<tr>
<td>A.3</td>
<td>Flow of execution</td>
<td>163</td>
</tr>
<tr>
<td>A.4</td>
<td>Jumping into an exception handler</td>
<td>166</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

To achieve both of runtime efficiency and maintainability of large-scale and complicated software, especially with dynamicity, portability, and concurrency, programmers have strong desire to customize and optimize their software as they need. For example, they may need to perform platform-specific optimizations so as to adapt their software from embedded systems and hand-held devices all the way up to large-scale servers and massively parallel processors (MPPs). Also, they may need to perform application-specific optimizations such as selection of compilation strategies, runtime partial evaluation, as well as application-specific idiom recognition. However, such customizations and/or optimizations generally require complicated program transformation which could only be done by highly-experienced programmers or compiler experts. These situations lead us to the necessity to develop more flexible approaches to make users-demanded customizations and optimizations easier.

From such a perspective, this thesis focuses on two different kinds of runtime environments, namely Java and C + MPI, which could be problematic to achieve both of runtime efficiency and maintainability of large-scale and complicated software.

Java is a portable and dynamic programming language which has a runtime environment which employs portable intermediate representations named Java bytecode and utilizes Just-In-Time (JIT) compilers, which compile (parts of) programs into native code at run-time. However, all the Java JIT compilers today only largely focus on standard platforms such as Workstations and Desktop PCs, and stress optimizing for speeding up single-threaded execution of general programs. As a result, Java programmers have neither effective measures to customize their programs for embedded systems, nor to optimize their programs by employing any advanced compilation strategies.

C + MPI (Message Passing Interface), on the other hand, is gaining acceptance as a standard for message-passing in high-performance computing,
due to its powerful and flexible support of various communication styles. C + MPI is used not only for writing new parallel applications, but also as an effective tool for parallelizing existing applications, as well as serving as an underlying message-passing library for implementations of parallel languages, such as High Performance Fortran (HPF) [100]. However, the complexity of Application Programming Interface (API) of MPI poses significant software overhead, and C + MPI programmers have no opportunity to remove such overhead. As a result, applicability of MPI has been restricted to rather regular, coarse-grained, and computation-intensive applications.

From the viewpoint of software engineering, one solution to these situations is to employ a “reflective” open compiler or a compiler framework and perform program transformation. These techniques can be effective measures to customize and optimize Java programs for wider range of platforms and applications, as well as to eliminate runtime overhead caused by the complexity of MPI and improve its adaptability to a wide variety of platforms and applications. Although such an observation has been largely accepted, its practical effectiveness has still been unclear.

In the case of Java, most JIT compilers today are typically written in C or C++ and architected to be closed and monolithic, and do not facilitate interfaces, frameworks, nor patterns as a means of customization. Although there have been a number of work in practical reflective systems that target Java, as described at Section 2.1.4, they have been limited to perform program transformations at the level of Java bytecodes and/or source codes. And, in general self-descriptive systems have been studied in various contexts such as reflection and interpreter / compiler bootstrapping, however, there is no precedent for an open-ended, reflective JIT compiler framework for Java, that offers stability and competitive performance to JIT compilers typically written in C or C++.

In the case of C + MPI, recent efforts is typically concentrating on realizing high-performance libraries, which encapsulate message-passing from users and provide a set of optimized, high-speed mathematical subroutines used to solve linear algebra and other numerically intensive problems. Especially, as described at Section 2.2.5, ATLAS and ABCLib have provided automatic tuning which determines a number of parameters such as blocking algorithms, block sizes, communication strategies, communication buffer sizes and so on, in order to fit them to various platforms from PC clusters up to Supercomputers. Although these efforts perform, so to speak, a kind of compile-time specialization of C + MPI programs, they are too limited to allow users to optimize more general applications such as solving fine-grained, irregular problems. On the other hand, there have been several compiler frameworks targeted to C or C++ such as Standford SUIF [133, 113] and COINS [25] and a number of studies in program transformations using such frameworks, however, they have mainly focused on code generation issues such loop transformation and scheduling. And, as far as we know, there is no
CHAPTER 1. INTRODUCTION

preceeding work which adapts the current compiler frameworks to meet the practical needs of low-latency, high-performance message-passing libraries.

In order to resolve these situations, we have developed two different practical systems; namely OpenJIT [81, 83, 87, 92] for Java and OMPi (Optimizing MPI) [86] for C + MPI. OpenJIT is an open-ended, reflective JIT compiler framework for Java written almost entirely in Java itself. Although in general self-descriptive systems have been studied in various contexts such as reflection and interpreter / compiler bootstrapping, OpenJIT is a first system we know to date that offers a stable, full-fledged Java JIT compiler that plugs into existing monolithic Java VMs, and offer competitive performance to JIT compilers typically written in C or C++. At the same time, OpenJIT is designed to be a compiler framework in the sense of Stanford SUIF, in that it facilitates high-level and low-level program analysis and transformation framework for the users to customize. And, OMPi system is a compile-time optimizer for C + MPI programs based on Stanford SUIF compiler framework, which eliminates much of the communication overhead using partial evaluation techniques which exploit static information of MPI calls in the target programs. Because partial evaluation alone is insufficient, we also utilize template functions for further optimization. Benchmarks will show that OMPi improves execution efficiency by as much as factor of two for communication-intensive application core with minimal code increase.

This thesis addresses the design and implementation of OpenJIT and OMPi system, as well as our practical experiences and technical contributions.

1.1 OpenJIT

1.1.1 Motivation

Programming Languages with high-degree of portability, such as Java, typically employ portable intermediate program representations such as bytecodes, and utilize Just-In-Time (JIT) compilers, which compile (parts of) programs into native code at run-time. However, all the Java JIT compilers today as well as those for other languages such as Lisp [82], Smalltalk-80 [82], and SELF [18, 57], only largely focuses on standard platforms such as Workstations and PCs, merely stress optimizing for speeding up single-threaded execution of general programs, usually at the expense of memory for space-time tradeoff. This is not appropriate, for example, for embedded systems where the tradeoff should be shifted more to memory rather than speed. Moreover, we claim that JIT compilers could be utilized and exploited more opportunely in the following situations:

- Platform-specific optimizations
CHAPTER 1. INTRODUCTION

Execution platforms could be from embedded systems and hand-held devices all the way up to large servers and massively parallel processors (MPPs). There, requirements for optimizations differ considerably, not only for space-time tradeoffs, but also for particular class of applications that the platform is targeted to execute. JIT compilers could be made to adapt to different platforms if it could be customized in a flexible way.

- **Platform-specific compilations**
  
  On related terms, some platforms require assistance of compilers to generate platform-specific codes for execution. For example, Distributed-Shared Memory (DSM) systems and persistent object systems require specific compilations to emit code to detect remote or persistent reference operations. Thus, if one were to implement such systems on Java, one not only needs to modify the Java VM, but also the JIT compiler. We note that, as far as we know, representative work on Java DSM (cJVM[7] by IBM) and persistent objects (PJama[11] at University of Glasgow) lack JIT compiler support for this very reason.

- **Application-specific optimizations**
  
  One could be more opportunistic by performing optimizations that are specific to a particular application or a data set. This includes techniques such as selection of compilation strategies, runtime partial evaluation, as well as application-specific idiom recognition. By utilizing application-specific as well as runtime information, the compiled code could be made to execute substantially faster, or with less space, etc. compared to traditional, generalized optimizations. Although such techniques have been proposed in the past, it could become a generally-applied scheme and also an exciting research area if efficient and easily customizable JIT compilers were available.

- **Language-extending compilations**
  
  Some work stresses on extending Java for adding new language features and abstractions. Such extensions could be implemented as source-level or byte-code level transformations, but some low-level implementations are very difficult or inefficient to support with such higher-level transformations in Java. Especially in Java, much of the underlying execution platforms are abstracted and/or not available.

  The above-mentioned DSM is a good example: Some DSMs permit users to add control directives or storage classifiers at a program level to control the memory coherency protocols, and thus such a change must be done at JVM and native code level. One could facilitate this by encoding such extensions in bytecodes or classfile attributes,
and customizing the JIT compilers accordingly to understand such extensions.

- **Environment-specific or Usage-specific compilations and optimizations**

  Other environmental or usage factors could be considered during compilation, such as adding profiling code for performance instrumentation, debugging and so on. In fact, we do exactly that in the benchmarking we show later in Chapter 4, which for the first time characterizes the behavior of a self-descriptive JIT compiler.

  Moreover, with Java, we would like these customizations to occur within an easy framework of portable, security-checked code downloaded across the network. That is to say, just as applets and libraries are downloadable on-the-fly, we would like the JIT compiler customization to be downloaded on-the-fly as well, depending on the specific platform, application, and environment. For example, if a user wants to instrument his code, he will want to download the (trusted) instrumentation component from the network on-the-fly to customize the generated code accordingly.

  Unfortunately, most JIT compilers today, especially those for Java, are architected to be closed and monolithic, and do not facilitate interfaces, frameworks, nor patterns as a means of customization. Moreover, JIT compilers are usually written in C or C++, and live in a completely separate scope from normal Java programs, without enjoying any of the language / systems benefits that Java provides, such as ease of programming and debugging, code safety, portability and mobility, and so on. In other words, current Java JIT compilers are “black boxes”, being in a sense against the principle of modular, open-ended, and portable design ideals that Java itself represents.

1.1.2 Our Solution

In order to resolve such a situation, the collaborative group between Tokyo Institute of Technology and Fujitsu Limited have been working on a project OpenJIT\[81, 88, 87, 92\], which stands for “an open-ended, reflective Just-In-Time Compiler framework for Java”. OpenJIT itself is a “reflective” Just-In-Time open compiler framework for Java written almost entirely in Java itself, and plugs into the standard Sun Microsystems Java Development Kit (JDK) 1.1, 1.2, and 1.3 Java VMs\[117\]. All compiler objects coexist in the same heap space as the application objects, and are subject to execution by the same Java Virtual Machine, including having to be compiled by itself, and subject to static and dynamic customizations. At the same time, it is a fully-fledged, Java Compatibility Kit (JCK)\[115\] compliant JIT compiler, able to run production Java code. In fact, as far as we know, it
CHAPTER 1. INTRODUCTION

is the ONLY Java JIT compiler whose source code is available in public, and is JCK compliant other than that of Sun Microsystems'. And, as the benchmarks will show, although being constrained by the limitations of the “classic” Java VMs, and still being in development stage lacking sophisticated high-level optimizations, it is nonetheless equal to or superior to the Sun Microsystems’ (classic) JIT compiler on SPEC JVM98\[111\] and SciMark 2.0\[112\] benchmarks, and attains about half the speed of the fastest JIT compilers that are much more complex, closed, and requires a specialized Java VM. At the same time, OpenJIT is designed to be a compiler framework in the sense of Stanford SUIF\[139, 113\], in that it facilitates high-level and low-level program analysis and transformation framework for the users to customize.

OpenJIT is still in active development, and we have just started distributing it for free for non-commercial purposes from the OpenJIT Project Web page (http://www.openjit.org/). It has shown to be quite portable, thanks in part to being written in Java — the SPARC version of OpenJIT runs on Solaris, and the x86 version runs on different breeds of UNIX including Linux, FreeBSD, and Solaris. We are hoping that it will stem and cultivate interesting and new research in the field of compiler development, reflection, portable code, language design, dynamic optimization, and other areas.

1.2 OMPI

1.2.1 Motivation

With the proliferation of Massively Parallel Processors (MPPs) and commodity clusters, standardized message passing interfaces such as PVM\[99\] and MPI\[85, 108\] are becoming increasingly popular. They are used not only for writing new parallel applications, but also as an effective tool for parallelizing existing applications, as well as serving as a runtime message passing library for implementations of parallel languages, such as High Performance Fortran (HPF)\[100\].

Since the first version of MPI Standard was announced at 1994, it is gaining increasing popularity, thanks to its powerful and flexible support of communication, such as different communication contexts via communicators, various synchronous / asynchronous communication modes, derived datatypes, group communications, etc. There have been a number of recent implementations of MPI as well. But, due to the inherent design of its API, the incurred software overhead is large, even compared to previous message passing libraries such as P4 or PVM. This is especially problematic when the hardware latency is low, because much of the benefits of fast networks are lost because of software overhead. This phenomenon not only applies to MPPs, but also to PC clusters, where the availability of low-cost, low-
CHAPTER 1. INTRODUCTION

latency networks such as the Myrinet is making low-latency communication possible.

As a result, application area of MPI has been somewhat restricted to regular, coarse-grained, and computation-intensive applications. In other words, attaining efficiency in fine-grained, irregular problems using MPI has been difficult. This is unfortunate, since standard message-passing libraries should encompass a wide variety of platforms and applications, including non-numerical applications, which are typically irregular and communication intensive. From C + MPI programmers’ point of view, they have no opportunities to remove such a large software overhead, and to fit their programs, especially with fine-granularity and irregularity, to the target platforms.

On the other hand, there has also been a string of work that has focused on reducing software overhead in message passing as much as possible. Notably, with Berkeley Active Messages, the incurred software overhead is in the order of several microseconds. The drawback is the relative lack of power and flexibility, and portability to some extent. Programming with native Active Messages library is much more difficult compared to programming with MPI, because primitives are “lower-level”. Furthermore, current Active Message does not support OS-level multithreading nor network heterogeneity well.

The question then is, would it be possible to have the best of both approaches, i.e., would it be possible to have a low-latency, high-performance message passing library, while retaining the flexibility and power of MPI?

1.2.2 Our Solution

In order to resolve such a situation, we have built OMPI (Optimizing MPI) system, where much of the software overhead is eliminated with partial evaluation techniques, attaining performance which approaches that of Active Messages. C programs that contain MPI function calls are statically analyzed in order to determine which arguments are static, and specialized with respect to those arguments. Because the current partial evaluation techniques are not sufficiently powerful to eliminate all the software overhead, we propose a technique where partial evaluation is combined with selection of pre-optimized template functions.

As a result, OMPI guarantees generality and portability of MPI programs, while allowing architecture-specific optimizations to be incorporated at compile-time. OMPI itself is also semi-portable, in that only template functions need to be reimplemented for a particular architecture. This is in contrast to traditional research on MPI implementation, where optimizations were highly architecture-specific.

To validate the effectiveness of our OMPI system, we performed some baseline benchmarks, and more extensive benchmarks on a set of appli-
cation cores with different communication characteristics, on the 64-node Fujitsu AP1000 MPP. The basic point-to-point latency improved from 338 microseconds to 76 microseconds, for communication intensive CG solver core, speedup of over a factor of two has been achieved. Even compared to traditional runtime optimization employing dynamic caching techniques, our OMPI system was consistently faster. The results show that our system is effective for various patterns of communication, significantly reducing the software overhead.

1.3 Technical Contributions

As for OpenJIT, an open-ended, reflective Just-In-Time Compiler framework for Java, this thesis presents the following technical contributions:

1. We propose an architecture for a reflective JIT compiler framework on a monolithic “classic” Java VM, and identify the technical challenges as well as the techniques employed. The challenges exist for several reasons, that the JIT compiler is reflective, and also the characteristics of Java, such as its pointer-safe execution model, built-in multi-threading, etc.

2. We show an API that adds to the existing JIT compiler APIs in “classic” Java VM to allow reflective JIT compilers to be constructed. Although still early in its design, and requiring definitions of higher-level abstractions as well as additional APIs for supporting JIT compilers on more modern VMs, we nonetheless present a minimal set of APIs that were necessary to be added to the Java VM in order to facilitate a Java JIT compiler in Java.

3. We demonstrate how reflective JIT compilers could be useful class-specific or application-specific customization and optimization by providing an important reflective “hook” into a Java runtime system, with the notion of compilets. Although the current examples are small, we nevertheless present a possibility of larger-scale deployment of OpenJIT for uses in the above-mentioned situations.

4. We perform extensive analysis of the performance characteristics of OpenJIT, both in terms of execution speed and memory consumption. In fact, as far as we know, there have not been any reports on any self-descriptive JIT compilation performance analysis, nor memory consumption reports for any JIT compilers. In particular, we show that (1) JIT compilation speed does not become a performance issue, especially during the bootstrap process when much of the OpenJIT compiler is run under interpretation, (2) memory consumption of reflective JIT compilers, however, could be problematic due to recursive
CHAPTER 1. INTRODUCTION

compilation, especially in embedded situations, (3) that there are effective strategies to solve the problems, which we investigate extensively, and (4) that the solutions do not add significant overhead to overall execution, due to (1). In fact, the self-compilation time of OpenJIT is quite amortizable for real applications.

Also, as for OMPI, a compile-time optimizer for C + MPI programs, this thesis addresses the following technical contributions:

1. We first analyze the source of the software overhead, identifying problems pertinent to message passing libraries in general, and those that are specific to MPI. We then investigate the opportunities for optimization by removing the overhead when static information is exploited.

2. In order to exploit the optimization opportunities analyzed, we propose OMPI, a system which optimizes MPI programs using partial evaluation techniques. OMPI works as a preprocessor to programs written in C + MPI, is semi-portable, and do not require customized C compilers, operating systems, or hardware.

3. In order to validate the effectiveness of our OMPI system, we performed some baseline benchmarks, and also more extensive benchmarks on application cores. The results show that our system is effective for various patterns of communication, and significantly reduces the software overhead, even compared with traditional optimization techniques. In other words, our proposal eliminates much of the overhead analyzed so far, achieving the speed approaching Active Messages, while retaining the generality, flexibility, and portability of the MPI.

1.4 The Structure of This Thesis

From the next chapter, we will present the design and implementation of OpenJIT and OMPI systems. The structure of the rest of this thesis is as follows:

- Chapter 2: Related Work

There have been a number of work related to constructing an open-ended, reflective JIT compiler framework for Java and a compile-time optimizer for C + MPI programs. To make clear our work’s significance, we will first introduce and discuss preceding and succeeding work targeted to self-descriptive and/or reflective object-oriented compilers, object-oriented compiler frameworks, MPI implementations, and so on.
• Chapter 3: Design and Implementation of OpenJIT

In this chapter, we will describe the entire design and implementation of OpenJIT. OpenJIT has some characteristic requirements and technical challenges that were previously not seen in traditional reflective systems as well as JIT compilers. In order to better describe the technical challenges, we will first overview the OpenJIT framework.

And, we will show the overall architecture of the OpenJIT frontend system. The OpenJIT frontend system provides a Java class framework for higher-level, abstract analysis, transformation, and specialization of Java programs which had already been compiled by the Java bytecode compiler (javac). Also, we demonstrate how reflective JIT compilers could be useful class-specific or application-specific customization and optimization by providing an important reflective “hook” into a Java runtime system, with the notion of compilets. In particular, we perform “loop transformation” of the program by using the OpenJIT frontend system and show its result.

Moreover, we describe the architecture of the backend system, and some new technical issues (How is the circularity resolved? How does the JIT bootstrap? How does the JIT interface into the Java VM? How do multiple and self-compiling JIT threads interact with each other? etc.) involved in constructing a reflective JIT compiler, and their solutions.

• Chapter 4: Performance Analysis of OpenJIT

To give the viability of our solutions, we now analyze the behavior of OpenJIT with detailed benchmarks. Our concern is both execution speed and memory usage. The former is obvious, as the execution overhead of the JIT compiler itself as well as quality of generated code will have to match that of conventional JIT compilers. Memory usage is also important, especially in areas such as embedded computing, one of major Java targets.

• Chapter 5: OMPI: Optimizing C + MPI programs

In this chapter, we will present the design and implementation of our OMPI system.

We first analyze the source of the software overhead, identifying problems pertinent to message passing libraries in general, and those that are specific to MPI. We then investigate the opportunities for optimization by removing the overhead when static information is exploited.

In order to exploit the optimization opportunities analyzed, we propose OMPI, a system which optimizes MPI programs using partial evaluation techniques. OMPI works as a preprocessor to programs
written in C + MPI, is semi-portable, and do not require customized C compilers, operating systems, or hardware.

Also, in order to validate the effectiveness of our OMPI system, we performed some baseline benchmarks, and also more extensive benchmarks on application cores. The results show that our system is effective for various patterns of communication, and significantly reduces the software overhead, even compared with traditional optimization techniques.

- **Chapter 6: Conclusions**

  We will conclude this thesis.
Chapter 2

Related Work

2.1 OpenJIT

There have been a number of previous work related to constructing *OpenJIT*: an open-ended, reflective JIT compiler framework for Java. To make clear our framework’s significance, we will first introduce a brief history of the design and implementation of virtual machines with Just-In-Time (JIT) compilation and optimization, and discuss preceding self-descriptive object-oriented language systems, compiler frameworks targeted to Java, reflective systems targeted to Java, and other related work including open compilers and dynamic code generation and specialization.

2.1.1 Virtual Machines and JIT Compilers

Programming Languages designed to be executed on virtual machines have been widely spread in the last decade. Especially, Java and Common Language Runtime (CLR) have made virtual machine technology very popular and common even for the commercial and mainstream use. Such virtual machine-based implementations provide several software engineering advantages over static compilation and ahead-of-time (AOT) compilation: portable program representations, built-in memory management, thread management, security, and class-loading time dynamic program composition. However, such dynamic features posed significant overhead in many cases, and hence required new research and development for achieving efficient and high-performance execution.

In this section, we will describe a brief history of the research and development of virtual machines and JIT compilers.

Virtual Machines

In the area of virtual machines, the following five developments are well-known as milestones in the evolution of JIT compilation and/or adap-
Lisp\cite{82}, Adaptive Fortran (AF)\cite{55}, Smalltalk-80\cite{32}, SELF\cite{18, 57, 126}, and Java\cite{50}.

- **Lisp**

Lisp\cite{82} and its interpreters undoubtedly represent the first widely-used virtual machines in the research area. Although Lisp program itself has the form of high-level representation, its syntax allows Lisp implementers to realize simple parsing and direct interpretation. Lisp implementations introduced many of the characteristics that today’s virtual machines provide. Especially, garbage collection and dynamic loading are outstanding features. Actually, Lisp has promoted the development of automatic memory management and garbage collection for longer than a quarter of a century. And, Lisp supported an early precursor to dynamic loading, that is, Lisp’s “eval” allows evaluating an expression in the current environment and dynamically adding programs to the runtime system.

- **Adaptive Fortran (AF)**

Hansen’s thesis\cite{55} investigates an adaptive compiler system named Adaptive Fortran (AF) that performs, during program execution, code optimizations based on the dynamic behavior of the program, as opposed to static approaches that employ a fixed code generation strategy (i.e., one in which a predetermined set of code optimizations are applied at compile-time to an entire program). We believe that this work provided the first in-depth study on issues in online adaptive compilation and optimization. And, his thesis describes many of the challenges that adaptive compiler systems may face, such as how and when to determine modules and heuristics to perform compilation, how to deal with multiple optimization levels, and how to control runtime systems with using online profiling.

- **Smalltalk-80**

Deutsch and Schiffman\cite{32} describes the Smalltalk-80 virtual machine, the first “modern” virtual machine. Their Smalltalk-80 virtual machine implemented many of the core concepts of VM implementations today: full-fledged JIT compiler, inline caches to optimize polymorphic dispatch, and native code caches. In addition, their paper demonstrated that it was possible to implement viable virtual machines on

---

\footnote{Due to space constraints, this section focuses on the work that has the most direct influence to today’s VM implementations. More in-depth review will be found in Arnoldi’s survey paper\cite{4} which reviews the evolution and current state of adaptive optimization technology in virtual machines. And, Aycock’s survey paper\cite{12} reviews related dynamic compilation techniques other than virtual machines.}
existing architectures with only a modest increase in memory require-
ments and without any of the special language-specific hardware sup-
port, such as special-purpose microcodes, tagged memory, garbage
collection co-processor.

- **SELF**
  
The SELF project\[126\] developed many of the more advanced tech-
niques that were not introduced by Smalltalk-80 virtual machines
but were widely accepted by virtual machines today. Their techni-
cal contributions include: polymorphic inline caches (namely PICs),
on-stack replacement, selective compilation with multiple compilers,
type prediction and splitting, dynamic deoptimization for debugging,
and profile-directed inlining.

- **Java**
  
Sun Microsystems introduced the Java programming language
(Java)\[50\] in 1995, and the Java virtual machine undoubtedly became
the first virtual machine that spread into mainstream markets. To re-
alize efficient and high-performance Java runtime systems, many Java
vendors employed the JIT compilation and/or runtime optimization
techniques that had formerly been introduced by Smalltalk-80 and
SELF virtual machines. It was just like a movement that strongly
pushed forward with research and development of virtual machine-
based language systems. Microsoft also followed this movement with
the Common Language Runtime\[54, 83\], and this fact convincingly
proves that virtual machines has already been an established way of
constructing efficient language systems.

**JIT Compilers**

To realize efficient runtime performance, virtual machines began to incorpo-
rate runtime compilers that could perform a translation from interpretable
representation (typically source code or bytecode) to native code immedi-
ately before it executes. This simplest policy is widely known as a *JIT
compiler*.

Smalltalk-80\[32\] was a representative of JIT compilers as we mentioned
before. Smalltalk-80 implementation executed interpretable representation
(called v-code) by translating it to native code (called n-code) when needed.
Since optimized n-code was measured as five times larger than the corre-
sponding v-code, Deutsch and Schiffman introduced a code cache for n-code
which could be discarded and regenerated through recompilation as needed.
This code cache was known to be quite effective, especially in the platforms
whose memory resource was limited.
CHAPTER 2. RELATED WORK

After Smalltalk-80, several projects adopted this simplest “JIT-only” policy, such as the first generation of SELF and the early implementation of Java virtual machines.

“JIT-only” policy incurred compilation overhead before any bytecodes were executed. It would be problematic when compilation resources such as compilation time, working space for compilation, and compiled code space, were limited. To cope effectively with such situations, some virtual machine began to exploit the well-known fact that any programs spend the majority of time in a small fraction the code. They devoted most of compilation resources only to frequently-executed code, namely “hot spot”. Subsequently, so called, the second generation of JIT-enabled virtual machines employed two implementations, a light implementation (typically a fast interpreter or a fast non-optimizing compiler) and a more expensive optimizer applied only to codes in hot spot.

The SELF-93 (the third generation SELF) implementation was considered to be a representative of the second generation of JIT compiler enabled virtual machines. SELF-93 initially compiled each method with the fast non-optimizing compiler and invoked the optimizing compiler only for the frequently executed methods (or hot methods). They stressed the importance of the observed pause times for an interactive user, rather than the total spent time by the runtime compiler, because SELF was designed to be an interactive and pause-free (or low pause times) language system. SELF-93 identified hot methods using method invocation counts. In general, there is a trade-off between the granularity of sampling and the runtime sampling overhead, so they discuss many of the open problems in choosing a counter heuristic.

Detlefs and Agesen investigated mixed-mode execution more in-depth by exploring the trade-offs between an interpreter, a template-based fast JIT compiler, and a highly-optimizing compiler adapted for use as a JIT compiler. They showed that a combination of the fast JIT compiler and moderate use of the optimizing JIT compiler just for the most important methods provided the best results on SPEC JVM98 benchmark suite. They assumed an ”oracle” (in the form of profile-feedback) that identified the most important methods to compile with the optimizing compiler.

SELF-93 also introduced the integration of recompilation and inlining decisions. When an invocation count reached a threshold, the SELF system would traverse the call stack starting from the top method, using a heuristic to find a “base” method suitable for recompilation. Then, SELF-93 compiler would inline all of the traversed call stack into the base method.

The recompilation policy of HotSpot Server VM is very similar to that of SELF-93, including the inlining heuristic. IBM’s mixed-mode interpreter system also makes recompilation decisions with using method invocation counts, but uses different inlining policies from those of SELF-93 and HotSpot Server VM. The Intel Microprocessor Research Labs VM
(MRL VM)\cite{22} implemented continuous compilation that uses a spare processor to periodically probe the profile data (including method invocation counts) to determine hot methods needed to be recompiled.

And, Jikes RVM\cite{5,8} employed a more generalized technique called call stack sampling to select between multiple levels of optimization by relying on a cost-benefit model. That is, the system recompiles a method at a particular optimization level only when it estimates that the benefit of additional optimization outweighs the cost of recompilation. To estimate these quantities, Jikes RVM depends on models to predict the speedup in compiled code due to optimization, the cost of recompilation, the times the compiled code will be invoked, and so on.

**Historical Position of Our Research**

OpenJIT is based on the early implementation of Java virtual machines (Sun Microsystems Java Development Kit 1.1, 1.2, and 1.3), which is a representative the “JIT-only” VMs, as same as Smalltalk-80 and the first generation of SELF.

However, OpenJIT can also be considered to be a bridging technology from the first generation JIT compilers to the second generation JIT compilers. OpenJIT is a self-descriptive JIT compiler which provides “reflective” Just-In-Time compiler framework. Using OpenJIT, we can extend and customize a JIT compiler itself to support runtime profiling of method invocations, adaptive compilation, recompilation, and so on. In Section \ref{sec:43}, we introduce an experimental result to adapt OpenJIT several adaptive compilation strategies for memory suppression of the bootstrapping phase (in other words, we can determine hot methods needed to be compiled at JIT compilation time, and compile only them for faster execution.), as well as to perform extensive analysis to prove their effectiveness.

Today, JavaScript (ECMAScript as specified in ECMA-262, third edition) implementations with “JIT-only” virtual machines such as V8\cite{61} or Tracing JIT\cite{46}, are gaining popularity as a faster browser-based application platform. In order to realize faster compilation on a limited compilation resource, those systems only perform a direct transformation from an AST or SSA form representation to native code. We believe that OpenJIT will provide them a practical and/or theoretical base of realizing extensibility and customizability.

### 2.1.2 Self-descriptive Object-Oriented Language Systems

Most modern compilers and language systems are bootstrapped in a self-descriptive fashion, but they do not coexist at run-time. In fact, although Lisp\cite{82} and Smalltalk\cite{32} systems embodied their own compilers written in terms of itself and executable at run-time, they are typically source-to-
CHAPTER 2. RELATED WORK

bytecode compilers, and not bytecode-to-native code compilers, which JIT compilers are. In fact, as far as we know, there have not been any reports of a JIT compiler for a particular language being reflective. Most JIT compilers we have investigated, including those for Lisp, Smalltalk, Java as well as experimental languages such as SELF, have been written in C/C++ or in assembly language.

By all means, there are some technical commonalities in bootstrapping, but OpenJIT being a reflective JIT system it embodies several technical issues that otherwise would not occur for the above-mentioned systems, as we will cover in Section 3.4.2. Moreover OpenJIT, its frontend system in particular, is architected to be a compiler framework rather than just a JIT compiler, so that users could add their own customizations at the application level using compilets, as we will also describe in Section 3.2.1.

More recent efforts in self-descriptive, practical object-oriented system is Squeak. Squeak employs the Blue Book self-definition of Smalltalk, then bootstraps it using C, then further optimizes the generated VM. Bootstrapping in Squeak involves the VM only, and not the JIT compiler. The recently announced JIT Squeak compiler is written in C and basically only merges the code fragments corresponding to individual bytecode. Thus, this is not a true compiler in a sense, but rather a simple bytecode to binary translator. This was done to achieve very quick porting of Squeak to various platforms, and stems from some of the earlier work done in [94].

As for self-descriptive object-oriented Java VM implementations, we are aware of several efforts, namely JavaInJava, The Rivet virtual machine, MyJVM, Fast JVM Interpreter by Michael Bebenita et al., and Jikes RVM, which was formerly known as Jalapeño.

We will first introduce and discuss these preceding and succeeding self-descriptive object-oriented systems.

- **JavaInJava**
  JavaInJava is a Java virtual machine written in Java. The system was built at Sun Microsystems Laboratories in order to examine the feasibility of constructing high-quality virtual machines using Java and to experiment with new virtual machine implementation techniques. JavaInJava was written to be a clean, extensible platform and to serve as a reference platform for Java VMs. Since it interprets bytecodes and implements a Java virtual machine stack, its performance is poor. OpenJIT, on the other hand, compiles bytecodes and runs them directly on the hardware with the underlying Java VM stack. JavaInJava uses the underlying virtual machine for memory allocation and garbage collection, as OpenJIT does.

- **The Rivet virtual machine**
The Rivet virtual machine\textsuperscript{64} is an experimental Java virtual machine purely written in Java. Rivet is designed to develop and make available new advanced debugging and analysis tools for the Java environment. Same as JavaInJava and OpenJIT, it relies on the underlying Java VM for memory allocation and garbage collection. Since debugging and analysis are its main objectives, Rivet’s performance is not so high. OpenJIT, on the other hand, compiles bytecodes and runs them directly on the hardware with the underlying Java VM stack.

- **MyJVM**

MyJVM\textsuperscript{30} is also an experimental Java virtual machine purely written in Java. It is designed to be open and easily extensible at compile-time and at run-time, in order to realize language extensions, preprocessors implementation, modifications of the bytecode or of the virtual machine behavior, as OpenJIT is architected to be. Since MyJVM interprets bytecodes and implements a Java virtual machine stack, its performance can be poor. OpenJIT, on the other hand, compiles bytecodes and runs them directly on the hardware with the underlying Java VM stack.

- **Fast JVM Interpreter by Michael Bebenita et al.**

Michael Bebenita et al.\textsuperscript{13} have also implemented an experimental Java virtual machine which is purely written in Java and runs atop a host Java VM. They intended to implement the additional guarantees into the Java runtime system, and the execution overhead of their nested Java VM was quite acceptable in practice. From the viewpoint of security guarantees, that is, building trustworthy and safe Java runtime system, OpenJIT still depends on the existing Java VM implementations written in unsafe C language, but OpenJIT can opportunistically customize JIT compilers in order to generate native codes trustworthy.

- **Jikes RVM**

Jikes RVM\textsuperscript{5, 8}, which was formerly known as Jalapeño, is a major IBM effort in implementing a self-descriptive Java system. In fact, Jikes RVM is an aggressive effort in building not only the JIT compiler, but the entire Java VM in Java. On surface, one might assume that building a self-descriptive Java system subsumes building only the JIT compiler in Java. But the technical goals and achievements, as well as the advantages and the disadvantages of OpenJIT and Jikes RVM, are different, and could be even considered complementary. Further discussion between these systems will be described later in this section.
Table 2.1: Self-descriptive object-oriented Java VM implementations

<table>
<thead>
<tr>
<th>Name</th>
<th>Feature</th>
<th>Base-level VM</th>
<th>JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaInJava</td>
<td>VM</td>
<td>underlying</td>
<td>underlying</td>
</tr>
<tr>
<td>Rivet</td>
<td>VM</td>
<td>underlying</td>
<td>underlying</td>
</tr>
<tr>
<td>MyJVM</td>
<td>VM</td>
<td>underlying</td>
<td>underlying</td>
</tr>
<tr>
<td>Fast JVM Interpreter</td>
<td>VM</td>
<td>underlying</td>
<td>underlying</td>
</tr>
<tr>
<td>Jikes RVM</td>
<td>VM + JIT</td>
<td>Self VM (bootstrap written in C)</td>
<td>Self JIT</td>
</tr>
<tr>
<td>OpenJIT</td>
<td>JIT</td>
<td>underlying</td>
<td>Self.JIT (bootstrap written in C)</td>
</tr>
</tbody>
</table>

Discussion

Table 2.1 shows the comparison between all the above-mentioned five systems and OpenJIT. Except OpenJIT and Jikes RVM, all the other systems only concentrated on realizing Java VM in Java and demonstrating how such implementations are not so inefficient because of the use of the underlying Java VM and JIT compiler. Bootstrapping in these systems involves only the Java VM, and not the JIT compiler, unlike OpenJIT and Jikes RVM.

Since Jikes RVM is the only effort paralleling our research, from now on, we discuss the difference between OpenJIT and Jikes RVM further.

The fundamental difference stems from the fact that Jikes RVM rests on its own customized Java VM with completely shared address space, much the same way the C-based JIT compilers are with C-based Java VMs. Thus, there is little notion of separation of the JIT compiler and the VM for achieving portability, and the required definition of clean APIs, which is mandated for OpenJIT. For example, the JIT compilers in Jikes RVM can access the internal objects of the Java VM freely, whereas this is not possible with OpenJIT. So, although OpenJIT did not face the challenges of Java VM bootstrapping, this gave rise to investigation of an effective and efficient way of interfacing with a monolithic, existing JIT compilers, resulting in very different technical issues as will be described in Section 3.4.

The manner in which Jikes RVM bootstraps is very similar to Squeak and other past systems. The way the type safety of Java is circumvented, however, is similar to the technique employed in OpenJIT: there is a class called Magic, which defines a set of native methods that implements operations where direct access to VM internals are required. In OpenJIT, the Compile class defines a set of APIs using a similar technique. Unfortunately, again there is no mention of attempting to develop the API into a clean one for generalized purposes of self-descriptive JIT compilers for Jikes RVM.

There are other technical differences as well: OpenJIT is architected to be a compiler framework, supporting features such as decompilation, various
frontend libraries, whereas it is not with Jikes RVM. No performance benchmarks have been made public for Jikes RVM, whereas we present detailed studies of execution performance validating the effectiveness of reflective JIT compilers, in particular memory profiling technique which directly exploits the openness of OpenJIT. Interestingly enough, Jikes RVM is claimed to be only targeting server platforms, and not desktop nor embedded platforms. It would be quite interesting to investigate the memory performance of Jikes RVM in the manner we have done, in particular to test whether it makes sense to target smaller platforms or not.

Still, the Jikes RVM work is quite impressive, as it has a sophisticated three-level compiler system, and their integrated usage is definitely worth investigating. Moreover, there is a possibility of optimizing the the application together with the runtime system in the Java VM. This is akin to optimization of reflective systems using the First Futamura projection in object oriented languages, as has been demonstrated by Masuhara et al. work in [76] and also in [77], but could produce much more practical and interesting results. Such an optimization is more difficult with OpenJIT, although some parts of the Java VM could be supplanted with Java equivalents, resulting in a hybrid system.

2.1.3 Compiler Frameworks Targeted to Java

As we will describe in Chapter 3 more precisely, OpenJIT provides the frontend system and the backend system. The frontend system is a high-level program optimizer class framework in the spirit of systems such as SUIF (Stanford University Intermediate Format) Compiler System [139,118]. It decompiles and recovers the Abstract Syntax Tree (AST) from the bytecodes, and performs program transformation on AST in a standard way with modules such as flowgraph construction module, program analysis module, and program transformation module. Then, it compiles AST to the bytecodes. And, the backend system is a small JIT compiler in itself, which performs lower-level optimization over the outputted bytecodes from the frontend system and generates native code. It is also customizable from Java programs, with inheriting existing backend compiler modules.

We know of four related work in implementing compiler frameworks targeted to Java, namely Marmot[39], BLOAT [58], SableVM and Soot[45,128], and Joeq VM[135].

We will first introduce and discuss these preceding and succeeding efforts.

- Marmot

Marmot[39] is an optimizing compiler infrastructure for Java which is written almost entirely in Java. It includes an optimizing native code compiler, runtime system, and libraries for a large subset of Java. Marmot compiler implements lots of standard scalar optimizations,
as well as object-oriented optimizations. It employs a multi-level IR and a strongly-typed SSA form and supports several garbage collectors written in C++. We cannot evaluate the design and implementation of Marmot further, because it is not available.

- **BLOAT**
  
  BLOAT (Bytecode-Level Optimization and Analysis Tool)\cite{58} is Java bytecode optimizer written entirely in Java. By optimizing Java bytecode, code improvements can occur regardless of the compiler that compiled the bytecode or the virtual machine on which the bytecode is run. BLOAT performs many traditional program optimizations such as constant/copy propagation, constant folding and algebraic simplification, dead code elimination, and peephole optimizations. Additionally, it performs partial redundancy elimination of arithmetic and field access paths. BLOAT employs three intermediate representations: bytecode representation, typed 3-address representation, and typed SSA form.

- **SableVM and Soot**
  
  SableVM\cite{45} is a retargetable Java virtual machine written in C. It is designed to be a small, fast, and efficient Java VM, and architected to provide a research platform for conducting compiler research. They implement many interesting techniques into SableVM, such as bidirectional object layouts, a threaded interpreter and efficient locking. Soot\cite{128} is their framework for analyzing and optimizing Java programs and deals with three intermediate representations: a streamlined bytecode representation named Baf, a typed 3-address representation named Jimple, and a version of Jimple with aggregated high-level information named Grimp.

- **Joeq VM**
  
  Joeq VM\cite{135} is a virtual machine and compiler infrastructure implemented entirely in Java and designed to be a research platform for virtual machine technologies such as JIT or AOT (ahead-of-time) compilers, scheduling algorithms, and advanced runtime techniques including advanced garbage collection technologies. It is language independent, so any programs in supported languages, such as Java class files, SUIF intermediate representation files, and ELF binary files, can be seamlessly and dynamically compiled, linked, and executed. Joeq also provides a framework for analyzing and optimizing Java programs and deals with two intermediate representations, namely the register-based Quad IR which is a set of instructions that are organized into a control flow graph, and the stack-based Bytecode IR which is based on
**Table 2.2:** Compiler frameworks targeted to Java

<table>
<thead>
<tr>
<th>Name</th>
<th>Supported IR</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marmot[39]</td>
<td>multi-level IR, strongly-typed SSA form</td>
<td>bytecode-to-native (AOT compiler)</td>
</tr>
<tr>
<td>BLOAT[58]</td>
<td>bytecode IR, typed 3-address IR, typed SSA form</td>
<td>bytecode-to-bytecode (static bytecode optimizer)</td>
</tr>
<tr>
<td>SableVM and Soot[15, 128]</td>
<td>bytecode IR, typed 3-address IR</td>
<td>bytecode-to-bytecode (static bytecode optimizer)</td>
</tr>
<tr>
<td>Joeq VM[139]</td>
<td>bytecode IR, typed 3-address IR (Quad IR)</td>
<td>bytecode-to-native (AOT and JIT compiler)</td>
</tr>
<tr>
<td>OpenJIT</td>
<td>AST, bytecode IR (IL), typed 3-address IR (RTL)</td>
<td>bytecode-to-native (JIT compiler)</td>
</tr>
</tbody>
</table>

Byte Code Engineering Library (BCEL)\[1\] and corresponds directly to the bytecode in the Java class file.

**Discussion**

Table 2.2 shows the comparison between all the above-mentioned four systems and OpenJIT. Except OpenJIT and Joeq VM, all the other frameworks just work as static optimizing compilers or AOT compilers, and they don’t support any program transformations at the JIT compile-time. Whaley’s Joeq VM is, so to speak, a paralleling effort of our research.

While Joeq VM provides a full-fledged compiler framework and supports both of JIT and AOT compilations, OpenJIT provides a limited compiler framework compared to Joeq VM and supports only JIT compilation. To compromise this disadvantage, however, OpenJIT can cooperate with source-to-bytecode compile-time and/or bytecode load-time reflective systems, as we will describe in Section 2.1.4.

On the other hand, while OpenJIT allows handling AST, stack-based IL, and register-based RTL, Joeq VM only allows program transformation on the Bytecode IR (similar to our stack-based IL) and the Quad IR (similar to our RTL).

**2.1.4 Reflective Systems Targeted to Java**

There have been a number of work in practical reflective systems that target Java, such as EPP\[10\], OJ (OpenJava)\[123\], AspectJ\[70\], jContractor\[15\], Kava\[132\], Javassist\[124\], MetaXa\[115\], just to name a few. Welch and Stroud present a comprehensive survey of Java reflective systems, discussing differences and tradeoffs of where in the Java’s execution process reflection should
CHAPTER 2. RELATED WORK

occur\textsuperscript{[131]}. 

- **EPP**
  
  EPP\textsuperscript{[60]} is an extensible Java source-to-source preprocessor which can introduce new language features. The user can specify EPP plug-ins at the top of the Java source code by writing \texttt{#epp} name in order to incorporate various extensions of Java. Multiple plug-ins can be retrieved simultaneously as long as they do not collide with each other. Emitted source codes can be compiled by ordinary Java compilers and debugged by ordinary Java debuggers. EPP works not only as an extensible Java preprocessor but also as a language-experimenting tool for language researchers; a framework for extensible Java implementation; and a framework for a Java source code parser / translator. The EPP’s source code is written in Java extended by EPP itself, and it was bootstrapped by EPP written in Common Lisp. The bytecode is available in any platform where Java is supported.

- **OJ (OpenJava)**
  
  OJ (a.k.a. OpenJava)\textsuperscript{[123]} is an extensible language based on Java, and it has the OJ MOP (Metaobject Protocol) which can be used for extending the language features. By employing the MOP API, programmers can handle source code as object oriented language constructs such as classes, methods, fields, and so on. Although its translation is performed at compile-time (source to source translation), the APIs are similar to Java Reflection API at run-time and easy to use for high-level translations. For instance, getting information about methods, adding methods, modifying methods and so on are easier. OJ itself is fully written in Java, and its translated codes are also written in standard Java language and can be executed on any Java VMs.

- **AspectJ**
  
  Aspect-oriented programming (AOP) is a technique for improving separation of concerns in software, which enables restricted structural program transformation on a target program by specifying declarative meta programs usually called “aspect”. AspectJ\textsuperscript{[70]} is the most popular and practical representative of AOP implementations in Java. It enables clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, logging, debugging, and so on. AspectJ itself is fully written in Java and generates standard Java bytecode which can be executed on any Java VMs.
• jContractor
jContractor is a 100% pure Java implementation of Design By Contract for Java. Contracts are written as methods that follow a simple naming convention. jContractor provides runtime contract checking by instrumenting the bytecode of classes that define contracts. jContractor can either add contract checking code to class files to be executed later, or it can instrument classes at run-time as they are loaded. All contracts are written in standard Java, so there is no need to learn a special contract specification language. jContractor is purely library based, requires no preprocessing or modifications to the Java VM.

• Kava
Kava is a portable implementation of a behavioural reflection for Java. It provides a metaobject protocol for specifying changes to run-time behaviour and implements these changes through the use of structural rewriting toolkits such as JOIE, Byte Code Engineering Library (BCEL), or Javassist. It is written entirely in Java, and doesn’t require a specialized Java Virtual Machine. Kava also provides support for properties such as strong non-bypassability and reflection on inherited methods that other reflective Java implementations do not address.

• Javassist
Javassist (Java programming Assistant) is a 100% pure Java implementation of a class load-time reflective system for Java. It is a class library for editing bytecodes in Java; it enables Java programs to define a new class at run-time and to modify a class file when the Java VM loads it. Unlike other similar systems, Javassist provides source-level abstraction; programmers can modify a class file without detailed knowledge of the Java bytecode. This ease of use is a unique feature of Javassist against other tools.

One of the applications of Javassist is runtime reflection; Javassist enables Java programs to use a metaobject that controls method calls on base-level objects. No specialized compiler or virtual machine are needed. Another application is remote method invocation. Javassist enables applets to call a method on a remote object running on the web server. Unlike the Java RMI (Remote Method Invocation), the programmer does not need a stub compiler such as rmic; the stub code is dynamically produced by Javassist.

• MetaXa
MetaXa overall is a comprehensive Java reflective system, constructing a fully reflective system whereby many language features
CHAPTER 2. RELATED WORK

could be reified, including method invocations, variable access, and locking. MetaXa has built its own VM and a JIT compiler; as far as we have communicated with the MetaXa group, their JIT compiler is not full-fledged, and is specific to their own reflective Java VM. Moreover, their JIT compiler is not robust enough to compile itself. In fact, we are considering collaborative porting of OpenJIT to their system.

Besides the above reflective systems, we are aware that a bunch of other reflective systems are proposed and developed, such as Reflective Java\cite{103} (source preprocessor), Bean Extender (bytecode preprocessor), Rjava\cite{29} (customized VM), Guaraná\cite{89} (an extension of Kaffe OpenVM\cite{65} which allows dynamic extension of classes), and Reflex\cite{36} (composite framework for multiple reflective extensions).

Discussion

Table 2.3 shows the comparison between all the above-mentioned systems and OpenJIT. OpenJIT is the sole instance that can reflectively customize and extend the behavior of the JIT compiler. OpenJIT can also perform bytecode-to-bytecode structural program transformation using OpenJIT frontend system.

The disadvantage in OpenJIT is that structural reflection to the bytecodes never occurs when they are interpreted and executed by the Java VM. Therefore, the effectiveness of the OpenJIT frontend system is considered to be limited. However, OpenJIT can exploit runtime information which can never be utilized from source or bytecode preprocessors and bytecode load-time rewriting systems. In addition, OpenJIT can cooperate with structural rewriting toolkits such as OJ or Javassist and perform reflective modifications to the target programs at the source-to-bytecode compile-time and/or bytecode load-time, as we mentioned in Section 2.1.3.

2.1.5 Other Related Work

Although a number of work in the context of open compilers have stressed the possibility of optimization using reflection such as OpenC++\cite{19}, our work is the first to propose a system and a framework in the context of a dynamic Just-In-Time compiler, where runtime information could be exploited.

From such a perspective, another related area is dynamic code generation and specialization such as \cite{53, 52, 21}. Their intent is to mostly provide a form of runtime partial evaluation and code specialization based on runtime data and environment. They are typically not structured as a generalized compiler, but have specific libraries to manipulate source structure, and generate code in a “quick” fashion. In this sense they have high commonalities with the OpenJIT frontend system, sans decompilation and being able to
Table 2.3: Reflective systems targeted to Java

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation Level</th>
<th>Description</th>
<th>JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPP[60]</td>
<td>source</td>
<td>offline source preprocessor</td>
<td>OK</td>
</tr>
<tr>
<td>Reflective Java[103]</td>
<td>source</td>
<td>offline source preprocessor</td>
<td>OK</td>
</tr>
<tr>
<td>OJ[123]</td>
<td>source-to-bytecode compile-time</td>
<td>structural reflection using compile-time MOP</td>
<td>OK</td>
</tr>
<tr>
<td>AspectJ[71]</td>
<td>source-to-bytecode compile-time</td>
<td>structural reflection using AOP weaver</td>
<td>OK</td>
</tr>
<tr>
<td>Bean Extender</td>
<td>bytecode</td>
<td>offline bytecode preprocessor</td>
<td>OK</td>
</tr>
<tr>
<td>jContractor[60]</td>
<td>bytecode</td>
<td>offline bytecode preprocessor with “Design By Contract” for Java</td>
<td>OK</td>
</tr>
<tr>
<td>Kava[132]</td>
<td>bytecode</td>
<td>behavioural and structural reflection</td>
<td>OK</td>
</tr>
<tr>
<td>Javassist[123]</td>
<td>source-to-bytecode compile-time, class load-time</td>
<td>structural reflection using compile-time and class load-time MOP</td>
<td>OK</td>
</tr>
<tr>
<td>MetaXa[19]</td>
<td>runtime</td>
<td>customized VM for reflection</td>
<td>NG</td>
</tr>
<tr>
<td>Rjava[29]</td>
<td>runtime</td>
<td>wrapper-based reflection which requires new bytecodes and a customized VM</td>
<td>NG</td>
</tr>
<tr>
<td>Guaraná[39]</td>
<td>runtime</td>
<td>an extension to Kaffe VM (customized VM)</td>
<td>NG</td>
</tr>
<tr>
<td>OpenJIT</td>
<td>bytecode, JIT compile-time</td>
<td>JIT compiler which supports bytecode-to-bytecode structural reflection and customization of JIT compiler itself</td>
<td>OK</td>
</tr>
</tbody>
</table>
handle generalized compilation. It is interesting to investigate whether specialization done with a full-fledged JIT compiler such as OpenJIT would be either be more or less beneficial compared to such specific systems. This not only includes execution times, but also ease of programming for customized compilation.

And, Consel et al. have investigated a hybrid compile-time and runtime specialization techniques with their Tempo and Harrisa system \[129, 112\], which are source-level Java specialization system written in C; techniques in their systems could be applicable for OpenJIT with some translator to add annotation information for predicated specializations.

### 2.2 OMPI

There have also been many previous work related to constructing *OMPI: Optimizing C + MPI Programs*. In this section, we will introduce a brief history of MPI (Message Passing Interface) itself, and discuss previous MPI implementations for special hardware, general purpose hardware, and Grid. And, we will introduce high-performance libraries for large-scale servers and massively parallel processors (MPPs), which typically encapsulate the underlying MPI messaging from users and provide a set of optimized, high-speed mathematical subroutines.

#### 2.2.1 Message-Passing Interface (MPI)

*Message Passing* is a programming paradigm used widely on certain classes of parallel and distributed computing environments, especially distributed memory machines. Although there are a number of variations, the basic concept of processes communicating through messages is well known.

In the area of high-performance computing, in particular, various message-passing software had been developed in early 80s. Many vendors developed for their own special purpose machines such as nCUBE Express and Vertex, and Intel NX2, on the other hand, mainly universities and national laboratories developed for PC clusters such as the P4 library (from Argonne National Laboratory), PICL and PVM\[99\] (from Oak Ridge National Laboratory), LAM (from Ohio Supercomputer Center), to name a few. There was also a package developed specially for quantum chemistry called TCGMSG and commercial message-passing libraries such as nCUBE Express.

In early 90s, Message Passing Interface (MPI)\[85, 108\] was designed and standardized to define both of the syntax and semantics of a core set of message-passing library. In designing MPI, community have sought to make use of the most attractive features of a number of existing message-passing libraries as mentioned above, rather than selecting one of them and adopting it as the standard. This policy made MPI standard powerful and flexible,
thereby supporting a rich set of features including different communication contexts via communicators, various synchronous / asynchronous communication modes, derived data types, group communications, and so on. The software overhead that this policy incurred became the main reason why we developed the OMPI system as we describe in Chapter 5.

The standardization of MPI provides users portability and ease-of-use, as well as, provides vendors and developers with a clearly defined core set of routines that they can efficiently implement or possibly provide hardware support for. Today, MPI systems are available everywhere; C + MPI programs can be executed on all PC clusters, large-scale SMPs, Cray X1, or even on the Earth Simulator.

The first MPI standard, called MPI-1, was completed in May 1994. The second version, MPI-2, was completed in 1998. And the third version MPI 3.0 is under standardization process.

2.2.2 Hardware-specific Implementations

As far as we know, all the efforts to lower communication latency in MPI have been to tune the libraries so that their software overhead becomes minimal, given the fact that all the arguments are dynamic. None have employed static compiler techniques to improve performance.

In [40], Franke and Hochschild report that the lowest latencies achieved with their MPI implementations on SP1 and SP2 are 30 microseconds and 40 microseconds respectively, with throughputs of 9 Mb/sec. and 35Mb/sec. However, the figures are based on polling, and do not apply to multi-process environments in practice. In such a case, interrupt-based implementation must be used, where the overhead increases to 200 microseconds.

MPI/DE [73] is a implementation of MPI on NEC’s Cenju-3 MPP, where the underlying operating system is Mach 3.0. Because Mach has kernel-supported threads which could be notified via kernel upcalls, interrupt handling could be made faster. Konishi reports that lowest latencies are 60 microseconds, 90 microseconds, and 140 microseconds for polling-based, upcall-based, and standard interrupt-based implementations, respectively. However, because Cenju-3 that network DMA controllers operate in physical space while MPI/DE works in logical space and no user-level facilities are provided for performing the necessary translation, the messages must always be copied between user buffers and kernel buffers, incurring significant overhead. Furthermore, polling-based implementation is not usable in a multi-process setting, and upcall optimization is not portable in that it relies heavily on Mach functionalities.

Sitsky describes the implementation of MPI on AP1000 [107], where the underlying CellOS is slightly modified, and the broadcast network is utilized to lower group communication. Still, the latency are reported to be 171.8 microseconds and 64 microseconds for the in-place method similar to DMA.
CHAPTER 2. RELATED WORK

Interrupts and the protocol method respectively, and the throughput are 2.69 Mbytes/sec and 14.83 Mbytes/sec for the in-place method and the protocol method respectively, indicating that hardware performance is not well utilized.

More recently, high-performance systems don’t always have hardware-specific MPI implementations. Even in BlueGene/L with special network hardware, starting with MPICH, a general-purpose MPI implementation, as a basis, they provided an implementation that uses the tree and the torus networks efficiently and that has two modes of operation for leveraging the two processors in a node. SiCortex MPI implementation also starts with MPICH2 and specializes it for the SiCortex fabric. None of them have employed static compiler techniques to improve performance, and can avoid the overhead incurred from the software stack of MPICH.

2.2.3 General Implementations

As opposed to hardware-specific implementations, there have been several general-purpose MPI implementations, targeted to clusters of the stock computers connected with the commodity network, such as Ethernet, InfiniBand, or Myrinet.

MPICH is a widely portable implementation of MPI standard (both MPI-1 and MPI-2), which is an source open software by Argonne National Laboratory and Mississippi State University. It provides an MPI implementation that supports variety of commodity clusters connected with Ethernet or other high-speed networks such as 10 Gigabit Ethernet, InfiniBand, Myrinet, or Quadrics. And, MPICH is designed to be a research platform for MPI through an extensible modular framework for other derived implementations. For example, implementing Abstract Device Interface (ADI) to the lower-level communication interface, MPICH can be adapted to the proprietary HPC systems such as BlueGene, Cray, and SiCortex. As we will describe in Section 5.4, we also implemented ADI for Fujitsu AP1000 MPP, as a baseline for comparison.

LAM/MPI is another open source implementation of MPI-1 and MPI-2 maintained by Indiana University, which is originally derived from LAM message-passing library of Ohio Supercomputer Center. Same as MPICH, LAM/MPI also supports variety of commodity clusters connected with Ethernet or other high-speed networks. The notable feature of this systems is enhanced support for advanced functions such as checkpoint/restart using Berkeley Lab Checkpoint/Restart (BLCR), faster job startup, integration with Altair PBS job scheduling service, SMP-aware collective communication, and so on.

Open MPI is the successor to LAM/MPI, LA-MPI, and FT-MPI, which is developed and maintained by a consortium of academic, research, and industry partners. The main goal of Open MPI is to
CHAPTER 2. RELATED WORK

resolve the incompatibility issues between multiple MPI implementations and to enable the usage of multiple components with a single MPI process. To make it possible to do so, Open MPI provides a component architecture called MPI Component Architecture (MCA). This component architecture allows users to select and employ several network device drivers (Point-to-Point Transport Layer modules) simultaneously in a single MPI process at run-time. It is also convenient to support hierarchical network environments with heterogeneity, as well as to use third party software supporting only binary distribution.

Open MPI is undoubtedly the most promising MPI implementation today, which will be bundled with mainstream OSes, however its component architecture introduces still more software overhead needed to be eliminated, as we eliminated software overhead from the AP1000 MPI implementation with using OMPI.

2.2.4 Implementations for Grid

More recent efforts in implementing MPI systems, is concentrating on Grid environment, in which clusters are connected by wide-area networks. Especially, PACX-MPI\cite{44}, MPICH-G2\cite{67}, and MagPie\cite{72} are known as MPI implementations designed for the Grid environment. All of them exploit the hierarchy of communication media ranging from memory systems to wide-area networks, and to adapt to the complex structure in latency and topology of wide-area networks.

GridMPI\cite{79, 80} is also known as one of the most famous MPI implementation for Grid. It establishes a synthesized cluster computer by binding multiple cluster computers distributed geographically. GridMPI targets to make global communications efficient, by optimizing the behavior of protocols over the links with non-uniform latency and bandwidth, and also to hide the details of the lower-level network geometry from application developers. And, they also try to provide variations of collective communication algorithms and an abstract layer to hide network geometry, and an interface to the TCP/IP communication layer to make it adaptive to those algorithms.

By all means, Grid environment will be another source of software overhead, which our OMPI system can eliminate.

2.2.5 Automatic Tuning Libraries

There have been a number of work in high-performance libraries for large-scale servers and massively parallel processors (MPPs), which typically encapsulate message-passing from users and provide a set of optimized, high-speed mathematical subroutines used to solve linear algebra and other numerically intensive problems.

Especially, in order to preserve *performance portability* from PC clusters
up to Super computers, ATLAS (Automatically Tuned Linear Algebra Software) [138, 137] and ABCLib (Automatically Blocking-and-Communication adjustment Library) [39] have provided automatic tuning facility. They automatically determine a number of parameters for tuning performance of the target libraries, such as blocking algorithms, block sizes, communication strategies, communication buffer sizes and so on.

Although these work perform, so to speak, compile-time, load-time, or runtime specialization of C + MPI programs, they are too limited and inflexible to allow users to optimize more general applications such as solving fine-grained, irregular problems, because they don’t facilitate any compiler frameworks or infrastructures.

2.3 Our Impact on Succeeding Work

There have been a number of literature referred to our work. To make clear how our work has impacted (and will impact) on the current and future research and development of this area, in this section, we will introduce succeeding work that has direct and/or indirect influence from our OpenJIT and OMPI frameworks.

2.3.1 OpenJIT

The biggest technical impact of OpenJIT itself is to provide an open-ended, reflective JIT compiler framework for Java, which is customizable and extensible from users’ Java programs. Thereby, OpenJIT allows developers to customize the JIT compiler as they need, for example, to extend bytecodes for adding new language features and/or abstractions, to retarget the code generator to various unsupported platforms, to customize and optimize the OpenJIT compiler itself, and so on.

Moreover, as we will describe in Section 3.4, OpenJIT is the first system we know to date that realizes a JIT compiler bootstrapping in almost entirely Java. Although self-descriptive language systems have been studied in various contexts, there have been no practical systems, except OpenJIT, that can boot JIT compilers in a completely self-descriptive fashion; existing systems cannot boot from the “cold” state VMs. In fact, even Jikes RVM requires a pre-determined compiler that compiles a boot VM image prior to its boot, thereby it has no opportunity to customize the compiler itself after booting VMs. OpenJIT, on the other hand, allows programmers to modify the behavior of the JIT compiler more completely and reflectively than existing systems, because it has no such a restriction.

These characteristics of OpenJIT are particularly advantageous to facilitate (1) customized code generation, (2) program specialization, and (3) reflection and AOP. We will introduce succeeding work with respect to these
technologies and make clear how our work has influenced and impacted on today’s research and development on Java and its runtime systems.

**Customized Code Generation**

OpenJIT enables customized code generation on Java JIT compilers. This fact largely impacted on succeeding work to realize JIT compile-time optimizations and customizations specific to runtime platforms, and language-extending compilations.

Welsh’s Jaguar system[134] provides efficient access to hardware resources such as VIA, through a bytecode specialization technique which transforms Java bytecode sequences to make use of inlined Jaguar bytecode which implements low-level functionality. And, they employ OpenJIT to execute the Jaguar bytecode extensively and efficiently.

Dutchyn et al.[34] extend the Java VM to realize multiple dispatch semantics almost same as CLOS and cecil. Coupling the modified Classic VM with the modified OpenJIT, they examine their technique in practice, and show that their technique provides smaller dispatch latency than programmer-written double-dispatch code with equivalent functionality.

Hovemeyer et al.[59] propose idiom recognition as a lightweight technique for expressing atomic instructions, in order to support atomic memory operations for higher levels of concurrency than lock-based synchronization in Java. Their technique is simple to implement and fully compatible with the semantics of Java. In order to examine their technique in practice, they employ OpenJIT to add compare-and-swap (CAS) instructions to Java bytecodes. OpenJIT allows them to extend a JIT compiler to transform CAS bytecodes to SPARC casa instruction directly.

Sohda et al.[109] propose a software architecture model, JDSM which realizes Distributed-Shared Memory (DSM) in a portable manner. Since DSM systems generally require emitting code to detect remote reference operations, they address that they plan to implement an OpenJIT compiler module to detect remote references and insert read/write blocks. Unfortunately their work has discontinued, however, their key idea, using OpenJIT to facilitate to compile programs with respect to such specific platforms, is still alive. And, similar to JDSM, Atkinson et al.[10] also suggest making use of OpenJIT to detect and handle persistent reference operations properly in their PJama system which realizes orthogonal persistent objects.

Stepanian et al.[115] propose a strategy for inlining native functions into Java applications using a JIT compiler. Their strategy can substantially reduce the overhead of performing Java Native Interface (JNI) calls, while preserving the key safety and portability properties of the JNI. Their result suggests that OpenJIT native APIs can be made much faster by employing their technique, and OpenJIT can also be employed to inline native functions into Java bytecodes.
Program Specialization

Program specialization is a software engineering technique that adapts programs to given execution contexts, which are provided by the programmer, derived from invariants present in the programs, or runtime information. This technique can also be considered as a variation of customized code generation above-mentioned, and OpenJIT can propell existing research on program specialization into the JIT compile-time specialization.

Schultz et al.\cite{schultz2005} present an offline and automatic program specializer for Java, called JSpec. JSpec combines inter-procedural static analyses with aggressive global optimizations including partial evaluation, which automatically enables to eliminate overheads, due to the use of object-oriented abstractions in generic programs. In their article\cite{schultz2005}, they suggest that, instead of using offline specialization as JSpec performs, runtime specialization can be directly incorporated into the compilation process by using OpenJIT.

Clausen’s thesis\cite{clausen1997} describes a new system named Jumbo, which embeds runtime code generation features into Java. Jumbo implements a novel form of runtime compilation, where code fragments in binary form from several sources can be combined at run-time to form efficient specialized code. Jumbo also provides a rewriting system based on Java abstract syntax trees intended to optimize the code generators, where we can see the indirect influence of the OpenJIT frontend system by design. His thesis also addresses that, while JIT compilers including OpenJIT face trade-offs between compilation time and speed, they are beneficial to optimize programs more efficiently than Jumbo does. On the other hand, we can also employ their approach in order to realize light-weight code generation atop of OpenJIT as developers need.

Masuhara et al.\cite{masuhara2002} propose Runtime ByteCode Specialization (BCS) technique which analyzes programs and generates specialized programs at run-time in JVML\cite{masuhara2002}, almost equivalent to Java bytecode. BCS can be considered as a more comprehensive technique than JSpec and Jumbo, because it can specialize programs with respect to runtime information, as well as it can optimize the specialized programs (after specialization) through JIT compilers. However, because of the lack of any direct interfaces between specializers and JIT compilers, BCS incurs huge runtime overhead such as translating bytecodes to a JVML program, specializing a JVML program, translating a JVML program to bytecodes, and so on. In order to reduce the runtime overhead, they report an implementation idea that generates specialized code as an intermediate representation of the OpenJIT compiler and compiles it on the JIT compiler directly.
CHAPTER 2. RELATED WORK

Reflection and AOP

As we described above, OpenJIT enables JIT compilers to boot from the “cold” state VMs in a self-descriptive fashion. Thereby, it allows programmers to modify the behavior of the JIT compiler more completely and reflectively than existing systems do. This fact leads researchers to the possibility to develop more comprehensive systems that support reflection and AOP.

As we described at Section 2.1.4, there have been a number of work in practical reflective systems targeted to Java. Several literature\cite{132,49} suggest that they will make use of OpenJIT for realizing efficient and JIT compile-time adaptation.

Popovici et al.\cite{95} propose a new software architecture named spontaneous container which allows any applications to be homogeneously capable of transactional interaction and orthogonal persistence, even if they spontaneously join and/or leave the network. Spontaneous container essentially requires a mechanism which allows dynamic transformation of applications based on the runtime context. While they built their own prototype system named PROSE (PROgrammable extenSions of sErvices)\cite{96}, they also addressed that JIT compile-time adaptation using OpenJIT could be a possible solution.

2.3.2 OMPI

OMPI explicitly demonstrates that compiler-directed approaches including partial evaluation are capable of optimizing rather complex message-passing middlewares such as MPI, so as to eliminate much of their runtime overhead. Although compiler-directed approaches to messaging software have been discussed in the context of compiled communication\cite{28} for a long time, there have been no practical instances that are systematically designed as compiler passes and can easily be integrated with existing compiler frameworks, such as SUIF.

Moreover, we propose a technique where partial evaluation is combined with selection of pre-optimized template functions. Template functions are beneficial not only to eliminate much more software overhead than partial evaluation alone, but also to separate out the machine-specific implementations from the machine-independent implementations easily.

To make clear how our work has influenced and impacted on today’s research and development on message-passing software, we will introduce succeeding work with respect to optimizations for message-passing with using static information of programs, and template functions.

Optimizations for Message-Passing with Using Static Information

OMPI demonstrates how compiler-directed approaches including partial evaluation can be adapted to messaging-passing software, such as MPI, in
order to eliminate much of the runtime overhead. We know of several work that follows our work to apply compiler-directed approaches to Beowulf-style clusters and collective communications.

Karwandet et al. [68] proposes CC–MPI, an MPI prototype on Beowulf-style clusters with Ethernet, that supports “compiled communication” which applies aggressive optimizations to communications whose information is known at compile time. CC–MPI optimizes one-to-all, one-to-many, all-to-all, and many-to-many collective communication routines using the compiled communication technique. Their compiled communication technique seems to be almost as same as what we intended to do, they did not describe enough how they choose and compile one from multiple versions of collective communication routines by using static information of the target MPI programs. However, we can reuse their work to extend our OMPi system to support Beowulf-style clusters and collective communications.

In their succeeding work [38], they proposes STAR-MPI (Self Tuned Adaptive Routines for MPI collective operations), which can adapt collective communication routines to runtime platforms and application workloads. In STAR-MPI, they abandon the idea of static compilation and employ a dynamic technique to select one of the best performing routines at run-time. Although this approach fosters further optimizations to the target MPI programs, it incurs huge runtime overhead than traditional MPI implementations and restricts the applicability of MPI to regular, coarse-grained, and computation-intensive applications.

Cristóbal-Salas et al. [27, 26] surveys and demonstrates non-strict evaluation in applications executed on distributed memory architectures. In order to exploit fine-grained parallelism in distributed memory systems, they introduces, instead of an explicit and strict message-passing model, a new programming model with I-structures and DI-structures which allows non-strict data access and fully asynchronous operations. Based on our work, they also employs partial evaluation of local and remote memory accesses not only to eliminate much of the excess overhead of message passing, but also to reduce the number of messages when some information about the input or part of the input is known. Their results suggests that partial evaluation technique is beneficial to eliminate runtime overhead of message-passing programs as we show in Chapter 5, especially when we can relax the strictness of message-passing model.

**Template Functions**

Although there is no succeeding work that follows our technique where partial evaluation is combined with selection of pre-optimized template functions, we know of a few instances to use template selection technique to optimize MPI implementations.

MPI derived (user-defined) data types allow users to describe non-
contiguous and structured memory layout and communicate non-contiguous data with a single communication function. However, many implementations of MPI derived data types perform poorly, that makes developers avoid using this feature. In [17], Byna et al. presents a technique to automatically select templates that are optimized for memory performance based on the access pattern of derived data types. While our work is limited to contiguous data types, this work enhances it to support more general MPI data types. They shows that performance for various derived data types is significantly improved and comparable to that of optimized routines using manual packing and unpacking.

MPI-IO, which is parallel I/O feature standardized as part of MPI-2, requires data transfers between file and memory buffers. Thereby, the need for copying between differently typed MPI buffers arise. A straightforward solution to this typed copy problem is to pack and unpack the differently structured data via an intermediate buffer, however, this solution incurs huge runtime overhead due to an extra memory copy. Mir et al. [84] presents a technique to construct an input-output type at run-time, which subsumes both of the input and output MPI data types. This type is used to copy directly from input to output buffer by means of a special transpack function. Their input-output type can be considered as a special form of template function. Although they constructs the input-output type at run-time, their paper reports that the runtime overhead can easily be amortized by improvements in copying performance because of a modest overhead of constructing input-output types.
Chapter 3

Design and Implementation of OpenJIT

In this chapter, we will describe the entire design and implementation of OpenJIT. OpenJIT has some characteristic requirements and technical challenges that were previously not seen in traditional reflective systems as well as JIT compilers. In order to better describe the technical challenges, we will first overview the OpenJIT framework.

And, we will show the overall architecture of the OpenJIT frontend system. The OpenJIT frontend system provides a Java class framework for higher-level, abstract analysis, transformation, and specialization of Java programs which had already been compiled by the Java bytecode compiler (javac). Also, we demonstrate how reflective JIT compilers could be useful class-specific or application-specific customization and optimization by providing an important reflective “hook” into a Java system, with the notion of compilets. In particular, we perform “loop transformation” of the program by using the OpenJIT frontend system and show its result.

Moreover, we describe the architecture of the backend system, and some new technical issues (How is the circularity resolved? How does the JIT compiler bootstrap? How does the JIT compiler interface into the Java VM? How do multiple and self-compiling JIT compiler threads interact with each other? and so on) involved in constructing a reflective JIT compiler, and their solutions.

3.1 Overview of OpenJIT

Although there have been reflective compilers and Object-Oriented compiler frameworks as described in Section 2.1, OpenJIT has some characteristic requirements and technical challenges that were previously not seen in traditional reflective systems as well as JIT compilers. In order to better describe the technical challenges, we will first overview the OpenJIT framework.
3.1.1 Conceptual Overview

OpenJIT is a JIT compiler written in Java to be executed on “classic” VM systems such as Sun Microsystems Java Development Kit (JDK) 1.1, 1.2, and 1.3 Java VMs\[^{117}\]. OpenJIT allows a given Java code to be portable and maintainable with compiler customization. With standard Java, the portability of Java is effective insofar as the capabilities and features provided by the Java Virtual Machine (JVM); thus, any new features that has to be transparent from the Java source code, but which Java VM does not provide, could only be implemented via non-portable means. For example, if one wishes to write a portable parallel application under multi-threaded, shared memory model, then some form of distributed shared memory (DSM) would be required for execution under MPP and cluster platforms. However, Java VM itself does not facilitate any DSM functionalities, nor provide any software “hooks” for incorporating the necessary read/write barriers for user-level DSM implementation. As a result, one must either modify the Java VM, or employ some ad-hoc preprocessor solution, neither of which are satisfactory in terms of portability and/or performance. With OpenJIT, the DSM class library implementers can write a set of compiler metaclasses so that necessary read/write barriers, etc., would be appropriately inserted into critical parts of code.

Also, with OpenJIT, one could incorporate platform-, application-, or usage-specific compilation or optimization. For example, one could perform...
various numerical optimizations such as loop restructuring, cache blocking, etc. which have been well-studied in Fortran and C, but have not been well adopted into JIT compilers for excessive runtime compilation cost. OpenJIT allows application of such optimizations to critical parts of code in a pinpointed fashion, specified by either the class-library builder, application writer, or the user of the program. Furthermore, it allows optimizations that are too application and/or domain specific to be incorporated as a general optimization technique for standard compilers, as has been reported by [71].

In this manner, OpenJIT allows a new style of programming for optimizations, portability, and maintainability, compared to traditional JIT compilers, by providing separations of concerns with respect to optimization and code-generation for new features. That is to say, with traditional JIT compilers, we see in the upper half of Figure 3.1, the JIT compilers would largely be transparent from the user, and users would have to maintain code which might not be tangled to achieve portability and performance. OpenJIT, on the other hand, will allow the users to write clean code describing the base algorithm and features, and by selecting the appropriate compiler metaclasses, or even by writing his own separately, one could achieve optimization while maintaining appropriate separation of concerns. Furthermore, compared to previous open compiler efforts, OpenJIT could achieve better portability and performance, as source code is not necessary, and late binding at run-time allows exploitation of run-time values, as is with run-time code generators.

3.1.2 Architectural Overview

The OpenJIT architecture is largely divided into the frontend and the backend processors. The frontend takes the Java bytecodes as input, performs higher-level optimizations involving source-to-source transformations, and passes on the intermediate code to the backend, or outputs the transformed bytecode. The backend is effectively a small JIT compiler in itself, and takes either the bytecode or the intermediate code from the frontend as input, performs lower-level optimizations including transformation to register code, and outputs the native code for direct execution. The reason why there is a separate frontend and the backend is largely due to modularity and ease of development, especially for higher-level transformations, as well as defaulting to the backend when execution speed is not of premium concern. In particular, we strive for the possibility of the two modules being able to run as independent components.

OpenJIT will be invoked just as a standard Java JIT compiler would, using the standard JIT API on each method invocation. A small OpenJIT C runtime is dynamically linked onto the Java VM, disguised as a full-fledged C-based JIT compiler. Upon initialization, it will have set the CompiledCodeLinkVector within the Java VM so that it calls the neces-
CHAPTER 3. DESIGN AND IMPLEMENTATION OF OPENJIT

sary OpenJIT C stub routines. In particular, when a class is loaded, Java VM calls the OpenJIT_InvokeForCompiler() C function, which redirects the invoker functions for each method within the loaded class to OpenJIT_Invoke(). OpenJIT_Invoke(), in turn, upcalls the appropriate Java compile() method in the org.OpenJIT.Compile class, transferring the necessary information for compilation of the specific method. It is possible to specify, for each method, exactly which portion of the compiler is to be called; by default, it is the OpenJIT backend compiler, but for sophisticated compilation, OpenJIT frontend is called. After compilation, the upcall returns to OpenJIT_Invoke(), which calls the just compiled code through mb->invoker (mb means methodblock). Thus, the heart of OpenJIT compiler is written in Java, and the C runtime routines merely serve to “glue” the Java VM and the Java portion of OpenJIT. The details will be presented in Section 3.4.

Upon invocation, the OpenJIT frontend system processes the bytecode of the method in the following way: The decompiler recovers the Abstract Syntax Tree (AST) of the original Java source from the bytecode, by recreating the control-flow graph of the source program. At the same time, the annotation analysis module will obtain any annotating info on the class file, which will be recorded as attribute info on the AST.

Next, the obtained AST will be subject to optimization by the (higher-level) optimization module. Based on the AST and control-flow information, we compute the data and control dependency graphs, etc., and perform program transformation in a standard way with modules such as flowgraph construction module, program analysis module, and program transformation module using template matching. The result from the OpenJIT frontend will be a new bytecode stream, which would be output to a file for later usage, or an intermediate representation to be used directly by the OpenJIT backend.

The OpenJIT backend system, in turn, performs lower-level optimization over the output from the frontend system, or the bytecodes directly, and generates native code. It is in essence a small JIT compiler in itself.

Firstly, when invoked as an independent JIT compiler bypassing the frontend, the low-level IL translator analyzes and translates the bytecode instruction streams to low-level intermediate code representation using stacks. Otherwise the IL from the frontend is utilized. Then, the RTL Translator translates the stack-based code to intermediate code using registers (RTL). Here, the bytecode is analyzed to divide the instruction stream into basic blocks, and by calculating the depth of the stack for each bytecode instruction, the operands are generated with assumption that we have infinite number of registers. Then, the peephole optimizer would eliminate redundant

\footnote{In the current implementation, the existence of annotation is a prerequisite for frontend processing; otherwise, the frontend is bypassed, and the backend is invoked immediately.}
instructions from the RTL instruction stream, and finally, the native code generator would generate the target code of the CPU, allocating physical registers. Currently, OpenJIT supports the SPARC and the x86 processors as the target, but could be easily ported to other machines. The generated native code will be then invoked by the Java VM, as described earlier.

3.2 OpenJIT Frontend System

As described in Section 3.1, the OpenJIT architecture is largely divided into the frontend and the backend processors. In this section, we will describe the overall architecture of the OpenJIT frontend system, which is a high-level program optimizer class framework in the spirit of systems such as SUIF (Stanford University Intermediate Format) Compiler System.

3.2.1 Overview

The OpenJIT frontend system provides a Java class framework for higher-level, abstract analysis, transformation, and specialization of Java programs which had already been compiled by javac: (1) The decompiler translates the bytecode into augmented Abstract Syntax Tree (AST), (2) analysis, optimizations, and specialization are performed on the tree, and (3) the AST is converted into the low-level IL of the backend system, or optionally, a stream of bytecodes is generated.

Transformation over AST is done in a similar manner to SUIF (Stanford University Intermediate Format) Compiler System, in that there is a method which traverses the tree and performs update on a node or a subtree when necessary. There are a set of abstract methods that are invoked as a hook. The OpenJIT frontend system, in order to utilize such a hook functionality according to user requirements, extends the class file (albeit in a conformable way so that it is compatible with other Java platforms) by adding annotation info to the classfile. Such an info is called “classfile annotation”.

The overall architecture of the OpenJIT frontend system is as illustrated in Figure 3.2, and consists of the following four modules:

1. **OpenJIT Bytecode Decompiler**

   Translates the bytecode stream into augmented AST. It utilizes a new algorithm for systematic AST reconstruction using dominator trees.

2. **OpenJIT Class Annotation Analyzer**

   Extracts classfile annotation information, and adds the annotation information onto the AST.

3. **OpenJIT High-level Optimizer Toolkit**
The toolkit to construct “compilets”, which are modules to specialize the OpenJIT frontend for performing customized compilation and optimizations.

4. Abstract Syntax Tree Package

Provides construction of the AST as well as rewrite utilities.

We first describe the classfile annotation, which is a special feature of OpenJIT, followed by descriptions of the four modules.

3.2.2 Classfile Annotation

Classfile annotation in OpenJIT is additional information or directive added to the classfile to direct OpenJIT to perform classfile-specific (or application-specific, platform-specific) optimization and customization. Here are examples of directives possible with classfile annotations:

- Support for User-defined Optimizers and Specializers
OpenJIT allows user-level definitions and customizations of its optimizer and specializer classes in the frontend. The classfile annotation allows the user to specify which of the classes the user-defined compiler classes to employ, by means of naming the class directly, or encoding the classfile itself as an annotation.

- **Support for Memory Models e.g., Distributed Shared Memory (DSM)**

  As mentioned in Section 1.1.1, the support for various memory models including DSM requires insertion of appropriate Read/Write barriers for access to shared objects. However, there are algorithms to statically determine that objects are immutable or do not escape such as [15, 21, 20, 16, 136], which allow such barriers to be compiled away, eliminating runtime overhead.

- **Optimizing Numerical Optimizations**

  Numerical performance of Java is known to suffer due to array bounds checks, non-rectangular multidimensional storage allocation, etc. By marking the loops that can be statically determined to use the array in regular ways, we can apply traditional Fortran-style optimizations such as loop transformation, cache blocking, etc.

In order to implement the classfile annotation feature, we employ the attribute region of each method in the classfile. According to the Java VM specs, any attributes that the Java VM does not recognize are simply ignored; thus, classfiles with OpenJIT annotations can be executed on platforms without OpenJIT, achieving high portability (save for the programs that do not work without OpenJIT). One caveat is that there is no simple way to add extra information in the attribute field of classes themselves, due to the lack of appropriate JIT interface in the Java VM; thus, one must employ some convention, say, defining a “dummy” null method that is called by the constructor, whose sole purpose is to supply class-wide annotation information that would be cached in the OpenJIT compiler.

In order to create a classfile with annotation information, we either employ an extended version of source-to-bytecode compilers such as javac; for classfiles without source, we could use a tool to add such annotation in an automated way; in fact the tool we are currently testing is a modified version of the OpenJIT frontend system.

### 3.2.3 OpenJIT Bytecode Decompiler Module

OpenJIT Bytecode Decompiler inputs the bytecode stream from the classfile, and converts it into an augmented AST. The module processes the bytecode in the following way:
CHAPTER 3. DESIGN AND IMPLEMENTATION OF OPENJIT

1. Converts the bytecode stream into an internal representation of Java VM instruction, and marks the instructions that become the leading instruction of basic blocks.

2. Construct a control flow graph (CFG) with basic block nodes.

3. Construct a dominator tree that corresponds to the CFG.

4. Reconstruct the Java AST by symbolic execution of the instructions within the basic block.

5. Discover the control flow that originated from the short-circuit optimizations of the Java conditional expressions such as \&\& or || and \((x \ ? \ a \ * \ b)\), and recover the expressions.

6. Reconstruct the Java control structure using the algorithm described in \[75\].

7. Output the result as an AST, augmented with control-flow and dominator information.

All the above steps except (6) are either simple, or could be done with existing techniques, such as that described in \[1\]. Step (6), is quite difficult; most previous techniques published so far analyzed the CFG directly, and used pattern matching to extract valid Java control structures \[1, 98\]. Instead, we have proposed an algorithm which walks over the dominator tree, and enumerates over every possible patterns of dominance relation, which has a corresponding Java control structure. Compared to existing techniques such as Krakatoa\[98\], our method was shown to be faster, and more robust to code obfuscation. Some preliminary details can be found in \[75\].

3.2.4 OpenJIT Class Annotation Analyzer Module

The OpenJIT Class Annotation Analyzer module extracts the class annotation from a classfile, and adds the annotation information to the AST. The added annotations are typically compilets that modify the compiler more concretely, it processes the annotation in the following way:

1. First, it access the attribute region of the method. This is done by parsing the method block region extracted from the Java VM.

2. We process this byte array assuming that the annotation object has been serialized with writeObject(), constructing an annotation object.

3. We attach the annotation object to the AST as annotation information.
Because what kind of information is to be embodied in the classfile annotation differs according to its usage, the OpenJIT_Annotation is actually an abstract class, and the user is to subclass a concrete annotation class. The abstract superclass embodies the identifier of the annotation, and the AST node where it is to be attached. This is similar in principle to SUIF, except that the annotation must be extracted from the classfile instead of being given a priori by the user.

3.2.5 OpenJIT High-level Optimizer Toolkit

OpenJIT High-level Optimizer Toolkit is used to construct OpenJIT compilets, that are a set of classes that customizes the compiler. The toolkit provides means of utilizing the augmented AST for implementing traditional compiler optimizations, and is largely composed of the following three submodules:

- **Flowgraph Constructor**
  Flowgraph Constructor creates various (flow) graphs from the augmented AST, such as dataflow graph, FUD chains, control dependence graph, etc. The Flowgraph class is an abstract class, and Factory Method pattern is employed to construct user-defined flowgraphs.

- **Flowgraph Analyzer**
  The Flowgraph Analyzer performs general computation over the flowgraph, i.e., dataflow equation solving, handling merges, fix point calculation, etc. We employ the Command Pattern to subclass the Analyzer class for each algorithm, and each subclass triggers its own algorithm with the execute() method. The user can subclass the Analyzer class to add his own flowgraph algorithm.

- **Program Transformer**
  The Program Transformer employs declarative pattern matching and rewrite rules to transform the augmented AST. One registers the rule using the following API:

  - `register_pattern(Expression src, Expression dst)`
  - `register_pattern(Statement src, Statement dst)`

  Registers the transformation rule that transforms the src pattern to the dst pattern. The pattern can be constructed using the Abstract Syntax Tree Package described next.

---

2In the current version, compilets are not downloadable; this is primarily due to the fact OpenJIT itself is not yet entirely downloadable due to a few restrictions in the Java VM. We are currently working to circumvent the restrictions, and a prototype is almost working. Meanwhile, the Toolkit itself is available, and a custom version of OpenJIT can be created with “static” compilets using standard inheritance.
substitution(Expression root)
substitution(Statement root)

Searches the subtree with the designated root node depth-first, and if a match is found with the registered patterns, we perform the transformation.

Initial use of the current pattern matching technique proved to be somewhat too low-level; in particular, generation and registration of the transformation rule is still cumbersome. The next version of OpenJIT will have APIs to generate patterns and transformation rules from higher-level specifications, in particular for well-known program transformations (such as code motion, loop transformation, etc.)

3.2.6 Abstract Syntax Tree Package

The Abstract Syntax Tree Package is a utility package called from other parts of the OpenJIT frontend to implement low-level construction of the augmented AST, patterns for transformation rules, etc. The AST essentially implements the entire syntactic entities of the Java programming language. Each node of the AST corresponds to the expression or a statement in Java. The class hierarchy for the package is organized with appropriate subclassing of over 100 classes: (Figure 3.3). We show typical Expression and Statement classes in Figure 3.4 and Figure 3.5, respectively.

A typical Expression subclass for a binary operator (MultiplyExpression in the example) consists of the operator ID, left-hand and right-hand expressions, and reference to an annotation object. The code() method either generates the low-level IL for the backend, or a Java bytecode stream. The code() method walks over the left- and right-hand expressions in a recursive manner, generating code. When a node has non-null reference to an annotation object, it calls the execute() method of the annotation, enabling customized transformations and compilations to occur.

As a typical Statement subclass, IfStatement recursively generates code for the conditional in a similar manner to Expressions.

As such, the current OpenJIT is structured in a similar manner to OpenC++ [19], in that syntactic entities are recursively compiled. The difference is that we provide annotation objects that abstracts out the necessary hook to the particular syntax node, in addition to customization of the syntax node themselves. Thus, it is possible to perform similar reflective extensions as OpenC++ in an encapsulated way.

On the other hand, experience has shown that some traditional optimizations are better handled using SSA form, such as dataflow analysis, constant propagation, common subexpression elimination, loop transformation, and code motion. In the next version of OpenJIT, we plan to support SSA form
Figure 3.3: Class hierarchy of the Abstract Syntax Tree Package
3.3 Application-Specific Optimization using OpenJIT Frontend

In this section, we demonstrate how reflective JIT compilers could be useful class-specific or application-specific customization and optimization by providing an important reflective “hook” into a Java system, with the notion of *compilets*.

As an example of application-specific optimization, we tested loop transformation of the program in Figure 3.7 into an equivalent one as shown in Figure 3.8. In this example, we have added a compilet called LoopTransformer using the class annotation mechanism in the attribute region of the matmul() method by using a tool mentioned in Section 3.2.2.

---

Note that although we are using the Java source to represent the program, in reality the program is in bytecode form, and transformation is done at the AST level.
public class IfStatement extends Statement {
    int op; // Construct ID
    Expression cond; // Condition expression
    Statement ifTrue; // Statement of Then-part
    Statement ifFalse; // Statement of Else-part
    Annotation ann; // Embedded Annotation (default: null)

    // Convert AST to backend-IR form (or bytecodes)
    void code() {
        if (ann)
            ann.execute(this); // call-back for metacomputation
        codeBranch(cond); // generate code for Condition
        ifTrue.code(); // generate code for Then-part
        ifFalse.code(); // generate code for Else-part
        addLabel(); // add label for BREAK statement
    }

    // Simplify statement form (e.g. if (true) S1 S2 => S1)
    Statement simplify() {
        ...
    }
}

Figure 3.5: Statement class for the “If” statement
The execute() method of the LoopTransformer class searches the AST of the method it is attached to for the innermost loop of the perfect trinested loop. There, if it finds a a 2-dimensional array whose primary index is only bound to the loop variable of the outermost loop, it performs the necessary transformation. The overview of the LoopTransformer is shown in Figure 3.6; the real program is actually about 200 lines, and is still not necessarily easy to program due to relatively low level of abstraction that the tree package provides, as mentioned earlier. We are working to provide a higher level API by commonizing some of the operations as a compilet class library.

Also, one caveat is that the IL translator is still incomplete, and as such we have generated the bytecode directly, which is fed into the OpenJIT backend. Thus we are not achieving the best performance, due to the compilation overhead, and the information present in the frontend is not utilized in the backend. Nevertheless, we do demonstrate the effectiveness to some degree.

3.3.1 Benchmarking Environment

As an Evaluation Environment, we employed the following platform, and pitted OpenJIT against Sun Microsystems’ original JIT compiler (sunwjit) on JDK 1.2.2 (ClassicVM).

- Sun Ultra60 (UltraSpare II 300MHz×2, 256MB)
- Solaris 2.6-J
- JDK 1.2.2 (ClassicVM)

3.3.2 Benchmarking Result

For OpenJIT, we compared the results of executing Figure 3.7 directly, and also transforming at runtime using the OpenJIT frontend into Figure 3.8. For sunwjit, we performed the transformation offline at source level, and compiled both programs with javac. The size of the matrices (SIZE) are set to 200 × 200 and 600 × 600. Table 3.1 shows the results, before and after the transformation, and the setup time required for JIT compilation.

We see that the execution time of OpenJIT and sunwjit are within 10% of each other. This similar to SPEC JVM98 where OpenJIT and sunwjit for SPARCs. So, for the purposes of this benchmark, we can regard both systems to be essentially equivalent, and thus the benefits of reflection can be judged in a straightforward way.

The setup time for OpenJIT without frontend transformation is approximately 1.09 seconds, compared to 0.49 seconds for sunwjit. This verifies

---

4The overhead of for sunwjit is zero as it had been done offline.
public class LoopTransformer extends Annotation {
    int loop_nest = 0;
    LocalField index;
    LoopTransformer() {}
    boolean isRegularForm(Statement init, Expression cond, Expression inc) {
        // Check the initializer and the conditions of the ‘‘For’’
        // statement to verify that it is in a normal form.
    }
    void execute(Node root) {
        if (root instanceof CompoundStatement) {
            for (int i = 0; i < root.args.length; i++) {
                execute(root.args[i]);
            }
        }
        // Test whether the loop is a perfect tri-nested loop
        else if (root instanceof ForStatement &&
            root.body instanceof ForStatement &&
            root.body.body instanceof ForStatement) {
            if (isRegularForm(root.init, root.cond, root.inc) &&
                isRegularForm(root.body.init, root.body.cond, root.body.inc) &&
                isRegularForm(root.body.body.init, root.body.body.cond, root.body.body.inc)) {
                // (1) Record the loop variable of the root
                // (2) Verify that root.body.body does not include a
                //     ForStatement
                // (3) If it doesn’t then scan the RHS for a
                //     2-dimensional array of the form ([ ] ( [ ] index ) _ )
                // (4) If found then perform the appropriate
                //     transformation
            }
        }
        else return;
    }
}

Figure 3.6: Overview of LoopTransformer
public int[][] matmul(int[][] m1, int[][] m2) {
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                T[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }
    return T;
}

Figure 3.7: Matrix multiply method (original)

public int[][] matmul(int[][] m1, int[][] m2) {
    int tmp[] = m1[0];
    for (int j = 0; j < SIZE; j++) {
        for (int k = 0; k < SIZE; k++) {
            T[i][j] += tmp[k] * m2[k][j];
        }
    }
    return T;
}

Figure 3.8: Matrix multiply method (transformed)
CHAPTER 3. DESIGN AND IMPLEMENTATION OF OPENJIT

Table 3.1: Results of OpenJIT frontend optimization

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>200 before</th>
<th>200 after</th>
<th>600 before</th>
<th>600 after</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenJIT</td>
<td>2.52</td>
<td>2.26</td>
<td>85.22</td>
<td>77.74</td>
</tr>
<tr>
<td>OpenJIT setup-time</td>
<td>1.09</td>
<td>2.68</td>
<td>1.09</td>
<td>2.67</td>
</tr>
<tr>
<td>sunwjit</td>
<td>2.34</td>
<td>2.06</td>
<td>80.19</td>
<td>73.55</td>
</tr>
<tr>
<td>sunwjit setup-time</td>
<td>0.49</td>
<td>0.49</td>
<td>0.49</td>
<td>0.49</td>
</tr>
</tbody>
</table>

(all times are seconds)

our benchmarks in Chapter 4 where the compiler bootstrap overhead was quite small. The 1.59 seconds difference between the original and transformed is the overhead of frontend execution. The overhead consists of the process described in Section 3.2. We believe we can improve this overhead substantially, as the frontend has not been tuned as much as the backend, especially regarding generation of numerous small objects.

Still we see that, although when the matrix size is small (200 × 200), the overhead of frontend processing with a compilet exceeds that of the speed gain, for larger problem (600 × 600) this overhead is amortized for 7% improvement. Moreover, we expect to further amortize this as the transformation is done only once, and as a result, multiple execution of the same method will not pay the overhead allowing us to essentially ignore the setup overhead for 9% gain.

3.4 OpenJIT Backend System

We have seen in Section 3.2 that OpenJIT frontend is a high-level program optimizer class framework in the spirit of systems such as SUIF [135, 113]. There are some new technical issues involved in its architecture, such as the decompiler algorithm, and how compilets are merged into the system, the issues do not relate directly to that of being able to making a the reflective JIT system work. In particular, how is the circularity resolved? How does the JIT bootstrap? How does the JIT interface into the Java VM? How do multiple and self-compiling JIT threads interact with each other? These issues mostly has to do with the OpenJIT backend system. We now describe the architecture of the backend system, and the problems associated with constructing a reflective JIT compiler, and their solutions. The viability of the solutions, how they do (or rather do not) affect performance will be given in the detailed performance benchmark in Chapter 4.

3.4.1 Overview

As a JIT compiler, the high-level overview of the workings of OpenJIT backend is standard. The heart of the low-level IL translator is the
parseBytecode() method of the ParseBytecode class, which parses the bytecode and produces an IL stream. The IL we defined is basically an RISC-based, 3-operand instruction set, but is tailored for high affinity with direct translation of Java instructions into IL instruction set with stack manipulations for later optimizations. There are 36 IL instructions, to which each bytecode is translated into possibly a sequence of these instructions. Some complex instructions are translated into calls into run-time routines. We note that the IL translator is only executed when the OpenJIT backend is used in a standalone fashion; when used in conjunction with the frontend, the frontend directly emits IL code of the backend.

Then, RTL converter translates the stack-based IL code to register based RTL code. The same IL is used, but the code is restructured to be register-based rather than encoded stack operations. Here, a dataflow analyzer is then run to determine the type and the offset of the stack operands. We assume that there are infinite number of registers in this process. In practice, we have found that 24–32 registers are sufficient for executing large Java code without spills when no aggressive optimizations are performed. Then, the peephole optimizer would eliminate redundant instructions from the RTL instruction stream.

Finally, the native code generator would generate the target code of the CPU. It first converts IL restricting the number of registers, inserting appropriate spill code. Then the IL sequence is translated into native code sequence, and ISA-specific peephole optimizations are performed. Currently, OpenJIT supports the SPARC and x86 processors as the target, but could be easily ported to other machines. Our experience has been proven that it has not been too difficult to port from SPARC to x86, save for its slight peculiarities and small number of registers, due in part being able to program in Java. We expect that porting amongst RISC processors to be quite easy.

The generated native code will be then invoked by the Java VM, upon which the OpenJIT runtime module will be called in a supplemental way, mostly to handle Java-level exceptions.

The architectural outline of the OpenJIT backend is illustrated in Figure 3.9. Further details of the backend system can be found in Appendix A.

3.4.2 Technical Challenges in a Reflective Java JIT Compiler

As most of OpenJIT is written in Java, the bytecode of OpenJIT will be initially interpreted by the Java VM, and gradually become compiled for faster, optimized execution. Although this allows the JIT compiler itself to adapt to the particular execution environment the JIT optimizes for, it could possibly give rise to the following set of problems:
Figure 3.9: Overview of the OpenJIT backend system
• Invoking the Java-based JIT compiler from within the Java VM

As the JIT compiler is invoked in the midst of a call chain of the base Java program, there must be a smooth way to massaging the Java VM into invoking a JIT compiler in Java in a separate context.

• Recursive Compilation

The current OpenJIT is designed to be entirely bootstrapped in “cold” mode, i.e., no parts of the JIT compiler are precompiled. Thus, as is with any reflective system, there must be some mechanism to stop the infinite recursive process, and “bottom out”. This is a little more subtle than conventional compiler bootstrapping, as compilation occurs at runtime coexisting with compilation of applications; furthermore, the mechanism must be safe with respect to Java multi-threading, i.e., no deadlocks should occur.

• Speed and Memory Efficiency of the JIT compiler

A JIT compiler is beneficial only if the combined (compilation time + execution time) is smaller than the interpretation time under Java VM. In more practical terms, OpenJIT must compete with traditional C-based JIT compilers for performance. Here, because of the interpretation and possible slowness of JIT execution even if itself were JIT compiled due to quality of generated native code, it is not clear if such goals could be satisfied. Moreover, memory efficiency is of primary concern, especially for embedded systems. In this regard, there is a particular issue not present in C-based JIT compilers.

• Lack of appropriate API for Java-written JIT compilers in standard Java VM

A JIT compiler must be able to introspect and modify various data structures within the Java VM. Unfortunately, Java VM does not have any APIs for that purpose, primarily because it is likely that JIT compilers were assumed to be written with a low-level language such as C. For this purpose, there must be appropriate Java-level APIs which must be reasonably portable for Java VM introspection in OpenJIT.

• Multi-threading Problems with self-modifying code

OpenJIT will involve self-modifying code on its self-compilation. This will require appropriate locking to be performed, and care must be taken to avoid race conditions and deadlocks, as Java is fundamentally multi-threaded, and multiple compilations and run-time patching could occur concurrently. Also, under multiprocessor environment, appropriate code cache flushing to maintain cache consistency.
For the sake of brevity, we only cover the most salient technical features here: for complete technical details readers are referred to Appendix A.

**Invoking the Java-based JIT Compiler from within the Java VM**

In a “classic” Java VM, for each method, both JIT compilation and transfer of control to the native method happens at the point of the subject method invocation. There is no so-called on-stack replacement, that is, transfer of control between interpreted and compiled or native methods in the midst of method execution.

The Java VM interpreter loop is structured as follows. When a method is invoked, the invoker() function of the methodblock structure (a structure internal to the Java VM which embodies various info pertaining to a particular method) mb is called. Under interpretive execution, this in turn calls the Java VM to generate a new Java stack frame. The first argument of invoker() function o is the class object for static method calls, and the invoked object on normal method calls. The second argument mb is a pointer to the methodblock structure, and the third argument args_size indicates the types of the arguments. The 4th ee is a pointer to the execution environment structure ExecEnv.
while(1) {
    get opcode from pc
    switch(opcode) {
        ...(various implementation of the JVM bytecodes)
        callmethod:
            mb->invoker(o, mb, args_size, ee);
            frame = ee->current_frame; /* setup java frame */
            pc = frame->lastpc; /* setup pc*/
            break;
    }break;
}

We substitute the value of the invoker in methodblock structure
of every method to OpenJIT_invoke() when a class is loaded. The
OpenJIT_invoke() function is defined as follows in C:

```c
bool_t OpenJIT_invoke(JHandle *o, struct methodblock *mb,
                        int args_size, ExecEnv *ee);
```

This function in turns calls the OpenJIT_compile() in the C runtime
to dynamically compile the method. Thereafter, the control is transferred
in mb->invoker, transferring control to the just compiled method. The called
OpenJIT_compile() performs the following functions:

- **Mutual exclusion to prevent simultaneous compilation of the
  same method**
  
  We must prevent multiple threads from compiling the same method
  at the same time with proper mutual execution using a compile lock. We
  reserve a bit in the methodblock structure as a lock bit.

- **Setup of invoker and CompiledCode fields in the method-
  block structure**
  
  When a method is invoked, and subject to compilation, we reset the
  invoker and other fields in the methodblock so that any subsequent
  invocation of the method will have the method run by the interpreter
  during compilation. This allows natural handling of recursive self-
  compilation of OpenJIT compiler classes.

- **Invocation of the body of the JIT compiler**
  
  The Java method to invoke the compiler is then upcalled. An instance
  of a new JIT compiler in Java (to be more specific, its upcall entry
  class) is allocated and initialized for each JIT compiler invocation.
  Then, the compile() method of the instantiated entry class is up
with \texttt{do\_execute\_java\_method\_vararg()}. Note that the current call context is preserved in the stack; that is to say, the same thread is utilized to make the upcall.

- **Postprocessing of JIT compilation**
  
  After compilation, control returns to the C runtime. At this point, most of the compiler becomes garbage, except for the persistent information that must be maintained across method compilations. This is to facilitate dynamic change in the compiler with compilets, and also to preserve space, directly exploiting the memory management feature of Java. If the compilation is successful, we set the invoker field of the methodblock structure to the compiled native code. When compilation fails: The methodblock field values are restored to their original values.

  In practice, the invoker field is not directly substituted for the compiled native method, but rather we invoke a native code stub, depending on the type of the return argument. This is done to handle exceptions, java reflection, calls between native and interpreted code, etc.

  In this manner, the JIT compiler in Java is smoothly invoked on the same execution thread. In practice it is much more complicated, however, due to possibility of exceptions, JIT compilation occurring even on calls from native methods, advanced features such as backpatching, inlining, and adaptive compilation. Some of the issues are further discussed below, while for the rest refer to Appendix A.

**Recursive Compilation**

Recursive compilation is handled at the C runtime level of OpenJIT with simple locking mechanism, as we see in the following simplified code fragment (in practice, it would include more code such as support for adaptive compilation):
CHAPTER 3. DESIGN AND IMPLEMENTATION OF OPENJIT

```
COMPILE_LOCK(ee);
if (COMPILE_ON_THE_WAY(mb)) {
    /* now compiling this method.
       avoid from double compiling */
    COMPILE_UNLOCK(ee);
    return;
}
START_COMPILE(mb);
/* reset invoker temporarily */
mb->invoker = (mb->fb.access & ACC_SYNCHRONIZED) ?
    invokeSynchronizedJavaMethod : invokeJavaMethod;
/* reset dispatcher temporarily */
mb->CompiledCode = (void *)dispatchJVM;
COMPILE_UNLOCK(ee);
```

This is essentially where the compilation “bottoms out”; once the method starts to be compiled, a lock is set, and further execution of the method will be interpreted. In fact, in Java we actually obtain this behavior for free, as mutual exclusion of multi-threaded compilation has to be dealt with in any case, defaulting to interpretation.

However, in the case of recursive compilation, there are some issues which do not exist for C-based JIT compilers:

- **Possibility of Deadlocks**

  We must be assured that, as long as JIT compiler obeys the locking protocol, recursive multi-threaded compilation does not cause any deadlocking situations. This is proven by showing that cyclic resource dependencies will not exist between the multi-threaded compilations. Let the dependencies between the methods be denoted $m_1^c \rightarrow m_2^c$, where for execution of compiled method $m_1^c$ we need to execute a compiled method $m_2^c$. We further distinguish compiled and interpreted execution of methods with $m^c$ and $m^i$, respectively. Then, starting from the entry method as a node, graph of dependency relations will clearly form a tree for single-threaded case. For multi-threaded case, however, it must be shown that arbitrary interleavings of the tree via possible self compilation will only create DAGs. Informally this simply holds because all $m_i^i$'s will not be dependent on any other nodes, and thus the cycle will have to be formed amongst $m_c$'s, which is not possible.

  We also note that, in practice, deadlocks could and does occur not only between the JIT compiler and the Java VM. One nasty bug which took a month to discover was in fact such a deadlock bug. As it turns out, the “classic” Java VM locks the constant pool for a class when its
finalizer is run. This could happen just when OpenJIT tries to compile
the finalizer method, resulting in a deadlock.

• **Speed and Memory Performance Problems**

Aside from the JIT compiler merely working, we must show that the
JIT compiler in Java could be time and memory efficient. The issue
could be subdivided into cases where the OpenJIT is compiling (1)
application methods, and (2) OpenJIT methods.

The former is simply shown by extensive analysis of standard bench-
marks in Chapter 4, where it is shown that OpenJIT achieves good
time and memory performance and despite being constrained by the
limitations of the “classic” Java VM, such as handle-based mem-
ory systems implementation, non-strict and non-compacting GC, slow
monitor locking, etc.

The latter is much more subtle: because of recursive compilation, two
undesirable phenomena occur:

1. Compilation of a single application bytecode will set off a chain
   of recursive compilations, due to the dependency just discussed.
   This has the effect of accumulating compiling contexts of almost
   the entire OpenJIT system, putting excessive pressure on the
   memory system.

2. We could prevent the situation by employing adaptive compila-
   tion and defaulting back to interpretation earlier, but this will
   have the effect of slowing down the bootstrap time, as long as
   possibly having some residual effect on application compilation
   due to some OpenJIT compiler methods still being interpreted.

Both phenomena are strongly interrelated; in the worst case, we will
be trading speed, especially the bootup time, for space. On the other
hand, one could argue that little penalty is incurred by adaptive means,
not because of the typical execution frequency argument, but rather,
that because of recursive compilation, much of the OpenJIT system
could be compiled under interpretation in the first place. We perform
extensive performance analysis in order to investigate this issue in
Chapter 4.

**Lack of Appropriate API for Java-written JIT Compilers in Stan-
ard Java VM**

None of the current Java VMs, including the “classic” VM for which Open-
JIT is implemented, have sufficient APIs for implementing a JIT compiler
in Java. In particular, Java VM basically only provides APIs to *invoke* a
C-based JIT compiler, but does not provide sufficient APIs for generalized introspection or intersession features. Note that we cannot employ the Java reflection API either, for it abstracts out the information required by the JIT compiler.

Instead, we define a set of native methods as a part of the OpenJIT runtime. The Compile class declares the following native methods, which are defined in api.c of the distribution. There are 17 methods in all, which can be categorized as follows:

- Constant pool introspection methods (Figure 3.11)
- Native method allocation and reflection (Figure 3.12)
- Class resolution methods (used for inlining) (Figure 3.13)

As one can see, majority of the methods are such that either introspective or intercessive operations being performed on the Java VM.
One issue is modification of the constant pool; with the APIs we have defined, all the constant pool accesses are introspective, as there is no simple way to update the constant pool within the Java VM, such as adding a new identifier. The current OpenJIT thus prohibits change to the constant pool in the frontend. The new version will have cached indirect access to the constant pool, using the low-level APIs for direct access, and caching the results. Change to the constant pool will occur in this cache and not the constant pool in the Java VM. One drawback is that this will not be applicable OpenJIT classes, as the interpreter will observe the original value instead of the modified value.

The current API is sufficient, but admittedly too low level of abstraction, in that it exposes too much of the underlying VM design; indeed, our goal is to allow JIT compilers to be a customizable and portable hook to the Java system, and thus, have OpenJIT be portable across different kinds of VMs. For this purpose, in the next version of OpenJIT, we plan to design a substantially higher-level API, abstracting out the requirements of the different VMs. The implementation of the API for “classic” VM will sit on the current APIs, but other VMs will have different implementations of native methods.

Another issue is the safety of the API. In the current implementation, the OpenJIT native method APIs are accessible to all the classes, including the application classes. It is easy to restrict the access to just the compiler classes (those with path org.OpenJIT.), but this will preclude user-defined compiles. Some form of security/safety measures with scope control, such as restricting access only to signed classfiles, might be necessary. We are currently investigating this possibility to utilize the security API in JDK 1.2.

Still, our concern is that, the current trend in Java VM construction is to make the JIT compiler increasingly monolithic with the Java VM, eliminating the JIT interface altogether. We feel that such a trend is eliminating the possibility of increasingly interesting possibilities offered by an open-ended, reflective JIT compilers. By proposing an effective API for OpenJIT and other reflective JIT compilers, we hope that such trend towards monolithic architecture be reversed.
Multi-threading Problems with Self-Modifying Code

OpenJIT utilizes self-modifying code technique for optimization. We will only briefly describe this here, as the issue is relevant for all JIT compilers, not just reflective JIT compilers. In order to efficiently implement multi-word locking of the modified code region, OpenJIT employs a fast, lock-free algorithm that does not involve usage of Java monitors. The algorithm is not a general locking mechanism as in [3, 90], but is only applicable to code regions, and is very efficient as no atomic instructions are involved. Details are found in Appendix A.

3.5 Chapter Summary

In this chapter, we have described the entire design and implementation of OpenJIT. OpenJIT has some characteristic requirements and technical challenges that were previously not seen in traditional reflective systems as well as JIT compilers. In order to better describe the technical challenges, we have first overviewed the OpenJIT framework, which is largely divided into the frontend and the backend systems.

And, we have shown the overall architecture of the OpenJIT frontend system. The OpenJIT frontend system provides a Java class framework for higher-level, abstract analysis, transformation, and specialization of Java programs which had already been compiled by the Java bytecode compiler (javac).

Also, we demonstrated how reflective JIT compilers could be useful class-specific or application-specific customization and optimization by providing an important reflective “hook” into a Java system, with the notion of compilets. In particular, we have built a “loop transformation” compilet which performed loop transformation on the target program, and taken the benchmark using this compilet. The result has shown that it allows us to achieve 8-9% performance gain without changing the base-level code. Although the current examples are small, we have nevertheless presented a possibility of larger-scale deployment of OpenJIT for uses in class-specific or application-specific customization and optimization.

Moreover, we have described the architecture of the OpenJIT backend system, and identified the technical challenges associated with constructing a reflective JIT compiler, and shown our solutions to the challenges. The challenges exist for several reasons, that the JIT compiler is reflective, and also the characteristics of Java, such as its pointer-safe execution model, built-in multi-threading, etc. In particular, we considered the following set of challenges:

1. Invoking the Java-based JIT compiler from within the Java VM
2. Recursive Compilation
3. Speed and Memory Efficiency of the JIT compiler

4. Lack of appropriate API for Java-written JIT compilers in standard JVM

5. Multi-threading Problems with self-modifying code

In particular, we have shown an API that adds to the existing JIT compiler APIs in “classic” Java VM to allow reflective JIT compilers to be constructed. Although still early in its design, and requiring definitions of higher-level abstractions as well as additional APIs for supporting JIT compilers on more modern VMs, we nonetheless presented a minimal set of APIs that were necessary to be added to the Java VM in order to facilitate a Java JIT compiler in Java, contrasting to similar work such as Jikes RVM\[^5\], \[^8\].

The viability of our design and implementation, how they do (or rather do not) affect performance will be given in the detailed performance benchmark in Chapter \[^4\].
Chapter 4

Performance Analysis of OpenJIT

To give the viability of our solutions, we now analyze the behavior of OpenJIT with detailed benchmarks. As mentioned earlier, our concern is both execution speed and memory usage. The former is obvious, as the execution overhead of the JIT compiler itself as well as quality of generated code will have to match that of conventional JIT compilers. Memory usage is also important, especially in areas such as embedded computing, one of major Java targets.

All the OpenJIT objects, except for the small C runtime system, coexists in the heap with the target application. The necessary working space includes that of various intermediate structures of compiler metaobjects that the OpenJIT builds, including various flowgraphs, intermediate code, etc., and persistent data, such as the resulting native code. Standard C-based JIT compilers will have to allocate such structures outside the Java heap; thus, memory usage is fragmented, and efficient memory management of the underlying Java VM is not utilized. For OpenJIT, since both the application and compiler metaobjects will coexist in the heap, it might seem that we would obtain the most efficient usage of heap space.

On the other hand, the use of Java objects, along with automated garbage collection, could be less memory efficient than C-based JIT compilers. Moreover as mentioned in Section 3.4, there could be a chain of recursive compilations which will accumulate multiple compilation contexts, using up memory. It is not clear what kind of adaptive compilation techniques could be effective in decreasing the accumulation, while not resulting in substantial execution penalty.
Table 4.1: SPEC JVM98 benchmark information

<table>
<thead>
<tr>
<th>Name</th>
<th>Information</th>
<th>Allocated objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>File compression program using modified Lempel-Ziv method (LZW).</td>
<td>334MB</td>
</tr>
<tr>
<td>_202_jess</td>
<td>The Java Expert Shell System is based on NASA’s CLIPS expert shell system.</td>
<td>748MB</td>
</tr>
<tr>
<td>_209_db</td>
<td>DBMS simulator which performs multiple database functions on memory resident database.</td>
<td>224MB</td>
</tr>
<tr>
<td>_213_javac</td>
<td>The Java compiler from the JDK 1.0.2.</td>
<td>518MB</td>
</tr>
<tr>
<td>_227_mtrt</td>
<td>Multi-threaded raytracer.</td>
<td>355MB</td>
</tr>
<tr>
<td>_228_jack</td>
<td>A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS).</td>
<td>481MB</td>
</tr>
</tbody>
</table>

4.1 Benchmarking Environment

As an evaluation environment, we employed the following platform, and pitted OpenJIT against Sun Microsystems’ original JIT compiler (sunwjit) on JDK 1.2.2 (ClassicVM).

- Sun Ultra60 (UltraSparc II 300MHz×2, 256MB)
- Solaris 2.6-J
- JDK 1.2.2 (ClassicVM)

We took six programs from the SPEC JVM98 benchmark, as well as the simple “Hello World” benchmark. The six — _201_compress, _202_jess, _209_db, _213_javac, _227_mtrt, and _228_jack (Table 4.1 shows the characteristics of these benchmarks) — have been chosen as they are relatively compute intensive, do not involve mere simple method call loops, and not reliant on runtime native calls such as networks, graphics, etc. “Hello World” benchmark superficially only makes a call to System.out.println(), but actually it will have executed almost the entire OpenJIT system, the Java packages that OpenJIT employs, as well as the constructors of system classes. This allows us to observe the bootstrap overhead of the OpenJIT system.
In order to obtain the precise profile information for memory allocation, we employed the JVMPI (Java Virtual Machine Profiler Interface)\textsuperscript{[12]} of JDK 1.2.2.

This profiler reports the time, the class name or the type, and the amount of heap usage for events such as:

- **JVMPI\_EVENT\_CLASS\_LOAD** (class load time)
- **JVMPI\_EVENT\_OBJECT\_ALLOC** (object allocation time)
- **JVMPI\_EVENT\_OBJECT\_FREE** (object deallocation time)

The profile is output on **JVMPI\_EVENT\_GC\_FINISH** (GC completion time) and **JVMPI\_EVENT\_JVM\_SHUT\_DOWN** (VM shutdown time), and compensates for the overhead required for data collection.

Additionally, we extended OpenJIT to output its own profile information. This is because it is difficult to determine with JVMPI whether the allocated compiler metaobject is being used to compile application methods, or used for recursive compilation, because JVMPI merely reports both to be of the same class (say, merely as instances of ILnode, etc.). By combining JVMPI and OpenJIT profile information, we obtain precise information of how much space the live OpenJIT compiler metaobjects occupy, how much native code is being generated, how much of the native code is that of OpenJIT, along the execution timeline. Also, how much classfiles are being loaded, how many methods are being compiled, and what is the percentage of the OpenJIT classes, can be profiled as well.

Such profiling is done in real-time, in contrast to the simulation based profiling of SPEC JVM98 memory behavior in [33]. Such an approach is difficult to apply for our purpose, as JIT compilation is being directly involved, resulting in code not directly profiable with Java VM simulation. Our compiler-assisted profiling allows us to obtain almost as precise an information as that of [33] at a fraction of time. Nevertheless, the profile information generated is quite large, reaching several hundred megabytes for each SPEC JVM98 run.
4.2 Benchmarking Contents

4.2.1 The Size of OpenJIT

We first show the code size of OpenJIT compared to sunwjit (Table 4.2). As we can see, the frontend is approximately 3 times the size of backend in terms of number of lines, and factor of approximately 8–10 larger in terms of number of classes and methods. This is because the frontend contains numerous small classes representing syntactic entities of Java, whereas the backend has much larger method size, and the backend IL does not assign a class for each instruction. We also see that the combined size of OpenJIT backend and C runtime is smaller than sunwjit, but when it is self-compiled, the x86 version could get larger. Thus, this raises an interesting issue of what happens if we run the compiler always interpreted in embedded situations; in the subsequent benchmark, we will also investigate this possibility.

4.2.2 Baseline Performance

We next observe the baseline execution time of OpenJIT. We set the heap limit to 32MBytes (as mandated by the SPEC JVM98 benchmarks) comparing the execution on the following runtime environments:

- ClassicVM interpreter (interpreter)
- Sun JIT Compiler equipped with ClassicVM (sunwjit)
- OpenJIT with self compilation (OpenJIT)
- OpenJIT without self compilation (OpenJIT-int)
- HotSpot (HotSpot)\[121\]
- ResearchVM (ResearchVM)\[31\]
In Figure 4.1, we see that the overhead is well amortized, and “OpenJIT” is competitive with “sunwjit”, sometimes superior. The running time of programs range between 56–172 seconds, so the overhead of JIT compilation is well amortized, even for “OpenJIT-int”, given the relative expense of OpenJIT compilation over sunwjit.

The runtime comparison of execution times of each program depends on each program. For _201_compress, OpenJIT was 20% superior, whereas sunwjit was faster by about 18%. Other benchmarks are quite similar in performance. Even small but unnegligible difference in compilation overhead, OpenJIT is likely producing slightly superior code on average.

We observe an anomaly, however. In many cases OpenJIT-int was faster than OpenJIT with self-compilation. This somewhat contradicts our observation that compilation DOES incur some overhead, as difference between interpreted and compiled executions of OpenJIT itself should manifest, but doesn’t.

Comparing with Sun Microsystems’ HotSpot and ResearchVM, which have production level JIT compilers, we see that OpenJIT runs about half the speed of theirs for most benchmarks. It may be caused by the difference in the underlying VM structures and the efficient GC mechanisms.

We have also taken the same benchmarks on the x86 version of OpenJIT, and compared it against IBM’s JDK 1.1.8 JIT compiler, which is reputed to be the fastest JIT compiler for x86, in neck to neck with Sun Microsystems’ HotSpot. Figure 4.2 shows the results: we see that, for most benchmarks, OpenJIT x86 is superior to sunwjit, and runs about half the speed of IBM’s JIT compiler, despite being constrained by the “classic” Java VM.

On both of Solaris SPARC and Linux x86 environments, we have also performed the SciMark 2.0 benchmarks [9] (Figure 4.3, Figure 4.4), which consist of typical numerical applications such as Fast Fourier Transform, Jacobi Successive Over-Relaxation, Monte Carlo Integration, Sparse Matrix Multiply, and LU Matrix Factorization. We observe that, SciMark 2.0 results were almost similar to SPEC JVM98 for the most part. However, especially for SPARC architecture, OpenJIT is not always superior to sunwjit. This is because the current version of OpenJIT cannot utilize floating-point registers as much as possible.
CHAPTER 4. PERFORMANCE ANALYSIS OF OPENJIT

Figure 4.1: SPEC JVM98 on Solaris (SPARC)

Figure 4.2: SPEC JVM98 on Linux (x86)
CHAPTER 4. PERFORMANCE ANALYSIS OF OPENJIT

Figure 4.3: SciMark 2.0 on Solaris (SPARC)

Figure 4.4: SciMark 2.0 on Linux (x86)
Table 4.3: Heap behavior statistics for Hello

<table>
<thead>
<tr>
<th>Hello</th>
<th>interpreter</th>
<th>sunwjit</th>
<th>OpenJIT</th>
<th>OpenJIT-int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class #</td>
<td>167</td>
<td>172</td>
<td>185</td>
<td>185</td>
</tr>
<tr>
<td>Alloc Object #</td>
<td>4.890</td>
<td>5.244</td>
<td>90.600</td>
<td>37.059</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(74.831)</td>
<td>(31.149)</td>
</tr>
<tr>
<td>Alloc Size [MB]</td>
<td>0.273</td>
<td>0.285</td>
<td>2.906</td>
<td>1.265</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(2.316)</td>
<td>(0.941)</td>
</tr>
<tr>
<td>GC #</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Exec Time [sec]</td>
<td>0.38</td>
<td>0.45</td>
<td>1.27</td>
<td>1.28</td>
</tr>
</tbody>
</table>

Table 4.4: Heap behavior statistics for _201_compress

<table>
<thead>
<tr>
<th>_201_compress</th>
<th>interpreter</th>
<th>sunwjit</th>
<th>OpenJIT</th>
<th>OpenJIT-int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class #</td>
<td>224</td>
<td>226</td>
<td>241</td>
<td>241</td>
</tr>
<tr>
<td>Alloc Object #</td>
<td>15,547</td>
<td>9,399</td>
<td>136,328</td>
<td>81,910</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(107,197)</td>
<td>(62,742)</td>
</tr>
<tr>
<td>Alloc Size [MB]</td>
<td>110.640</td>
<td>110.266</td>
<td>114.330</td>
<td>112.662</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(3.318)</td>
<td>(1.918)</td>
</tr>
<tr>
<td>GC #</td>
<td>20</td>
<td>16</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>Exec Time [sec]</td>
<td>673.91</td>
<td>89.62</td>
<td>72.53</td>
<td>74.46</td>
</tr>
</tbody>
</table>

4.2.3 Bootup Performance and Heap Behavior

We next observe the memory usage characteristics of OpenJIT. We set the heap limit to 32MBytes (as mandated by the SPEC JVM98 benchmarks) comparing the execution of Java VM interpreter, sunwjit, OpenJIT with self compilation (OpenJIT), and OpenJIT without self compilation (OpenJIT-int).

Table 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, and 4.9 shows for each execution, how many classes are loaded and their sizes, how many objects are allocated (parenthesis indicates how many OpenJIT compiler metaobjects), how much memory size are allocated (and that of OpenJIT compiler metaobjects), wallclock execution time (same as showed in Section 4.2.2), and number of GCs.

Figure 4.3, 4.4, 4.5, and 4.6 additionally show consumed overall heap space, live OpenJIT object heap space, amount of generated native code, and the amount of OpenJIT self-compiled native code, along the time axis. This shows the process of compiler bootstrapping. The compilation in OpenJIT was set to be most aggressive i.e., all the methods are JIT compiled on their first invocations, and the entire frontend had been turned off, and are not loaded.
### Table 4.5: Heap behavior statistics for _202_jess

<table>
<thead>
<tr>
<th></th>
<th>interpreter</th>
<th>sunwjit</th>
<th>OpenJIT</th>
<th>OpenJIT-int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class #</td>
<td>373</td>
<td>375</td>
<td>390</td>
<td>390</td>
</tr>
<tr>
<td>Alloc Object #</td>
<td>7,951,562</td>
<td>7,936,214</td>
<td>8,103,973</td>
<td>8,049,403</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(142,626)</td>
<td>(98,045)</td>
</tr>
<tr>
<td>Alloc Size [MB]</td>
<td>221.919</td>
<td>221.190</td>
<td>226.383</td>
<td>224.710</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(4.402)</td>
<td>(2.998)</td>
</tr>
<tr>
<td>GC #</td>
<td>547</td>
<td>565</td>
<td>528</td>
<td>532</td>
</tr>
<tr>
<td>Exec Time [sec]</td>
<td>148.55</td>
<td>64.65</td>
<td>51.91</td>
<td>51.55</td>
</tr>
</tbody>
</table>

### Table 4.6: Heap behavior statistics for _209_db

<table>
<thead>
<tr>
<th></th>
<th>interpreter</th>
<th>sunwjit</th>
<th>OpenJIT</th>
<th>OpenJIT-int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class #</td>
<td>218</td>
<td>220</td>
<td>235</td>
<td>235</td>
</tr>
<tr>
<td>Alloc Object #</td>
<td>3,218,293</td>
<td>3,213,851</td>
<td>3,343,820</td>
<td>3,289,249</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(109,778)</td>
<td>(65,197)</td>
</tr>
<tr>
<td>Alloc Size [MB]</td>
<td>63.249</td>
<td>63.095</td>
<td>67.104</td>
<td>65.431</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(3.398)</td>
<td>(1.994)</td>
</tr>
<tr>
<td>GC #</td>
<td>33</td>
<td>32</td>
<td>39</td>
<td>37</td>
</tr>
<tr>
<td>Exec Time [sec]</td>
<td>307.48</td>
<td>142.16</td>
<td>172.83</td>
<td>182.08</td>
</tr>
</tbody>
</table>

### Table 4.7: Heap behavior statistics for _213_javac

<table>
<thead>
<tr>
<th></th>
<th>interpreter</th>
<th>sunwjit</th>
<th>OpenJIT</th>
<th>OpenJIT-int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class #</td>
<td>386</td>
<td>388</td>
<td>403</td>
<td>403</td>
</tr>
<tr>
<td>Alloc Object #</td>
<td>5,972,713</td>
<td>5,936,663</td>
<td>6,181,295</td>
<td>6,126,571</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(208,562)</td>
<td>(164,145)</td>
</tr>
<tr>
<td>Alloc Size [MB]</td>
<td>147.288</td>
<td>145.458</td>
<td>154.486</td>
<td>151.531</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(6.478)</td>
<td>(5.080)</td>
</tr>
<tr>
<td>GC #</td>
<td>80</td>
<td>69</td>
<td>77</td>
<td>67</td>
</tr>
<tr>
<td>Exec Time [sec]</td>
<td>200.94</td>
<td>94.850</td>
<td>102.11</td>
<td>108.85</td>
</tr>
</tbody>
</table>
CHAPTER 4. PERFORMANCE ANALYSIS OF OPENJIT

Table 4.8: Heap behavior statistics for _227_mtrt

<table>
<thead>
<tr>
<th></th>
<th>_227_mtrt interpreter</th>
<th>sunwjit</th>
<th>OpenJIT</th>
<th>OpenJIT-int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class #</td>
<td>239</td>
<td>241</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Alloc Object #</td>
<td>6,382,222</td>
<td>6,376,266</td>
<td>6,524,115</td>
<td>6,469,545</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(124,549)</td>
<td>(79,968)</td>
</tr>
<tr>
<td>Alloc Size [MB]</td>
<td>84.118</td>
<td>83.902</td>
<td>88.467</td>
<td>86.794</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(3.855)</td>
<td>(2.451)</td>
</tr>
<tr>
<td>GC #</td>
<td>90</td>
<td>90</td>
<td>96</td>
<td>93</td>
</tr>
<tr>
<td>Exec Time [sec]</td>
<td>173.51</td>
<td>57.56</td>
<td>55.16</td>
<td>55.50</td>
</tr>
</tbody>
</table>

Table 4.9: Heap behavior statistics for _228_jack

<table>
<thead>
<tr>
<th></th>
<th>_228_jack interpreter</th>
<th>sunwjit</th>
<th>OpenJIT</th>
<th>OpenJIT-int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class #</td>
<td>270</td>
<td>272</td>
<td>287</td>
<td>287</td>
</tr>
<tr>
<td>Alloc Object #</td>
<td>6,878,777</td>
<td>6,868,951</td>
<td>7,046,695</td>
<td>6,992,109</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(152,625)</td>
<td>(108,001)</td>
</tr>
<tr>
<td>Alloc Size [MB]</td>
<td>150.755</td>
<td>150.353</td>
<td>155.818</td>
<td>154.144</td>
</tr>
<tr>
<td>(for OpenJIT)</td>
<td></td>
<td></td>
<td>(4.707)</td>
<td>(3.302)</td>
</tr>
<tr>
<td>GC #</td>
<td>451</td>
<td>465</td>
<td>286</td>
<td>276</td>
</tr>
<tr>
<td>Exec Time [sec]</td>
<td>196.33</td>
<td>66.67</td>
<td>66.97</td>
<td>68.01</td>
</tr>
</tbody>
</table>
Figure 4.5: Timeline behavior of heap usage and live object allocated by interpreter

Figure 4.6: Timeline behavior of heap usage and live object allocated by sunwjit
Figure 4.7: Timeline behavior of heap usage and live object allocated by OpenJIT

Figure 4.8: Timeline behavior of heap usage and live object allocated by OpenJIT-int
CHAPTER 4. PERFORMANCE ANALYSIS OF OPENJIT

Figure 4.9: Timeline behavior of heap usage with SPEC JVM98 compress (sunwjit vs. openjit)

Figure 4.10: Timeline behavior of heap usage with SPEC JVM98 jess (sunwjit vs. openjit)
Figure 4.11: Timeline behavior of heap usage with SPEC JVM98 db (sunwjit vs. openjit)

Figure 4.12: Timeline behavior of heap usage with SPEC JVM98 javac (sunwjit vs. openjit)
Figure 4.13: Timeline behavior of heap usage with SPEC JVM98 mtrt (sunwjit vs. openjit)

Figure 4.14: Timeline behavior of heap usage with SPEC JVM98 jack (sunwjit vs. openjit)
The Hello benchmark exemplifies the overhead of bootstrapping OpenJIT and OpenJIT-int; compared to sunwjit, we see approximately 2.8 times increase in startup time, indicating that compilation with OpenJIT incurs approximately $\times 3$ overhead over sunwjit. On the other hand, different between OpenJIT and OpenJIT-int is negligibly small; this indicates that overhead of self compilation is almost negligible, but rather, the overhead of system and library classes are substantial (we observe approximately 457 methods compiled, as opposed to 128 methods for OpenJIT).

For the six SPEC JVM98 benchmarks, we see that, since method-specific openjit compiler metaobjects are mostly thrown away on each compilation, we do not occupy memory compared to sunwjit (Figure 4.5, 4.6, 4.7, 4.8) in principle. In fact, we may be utilizing memory better due to sharing of the heap space with the application.

We do observe some anomalies, however. For most cases OpenJIT had increased invocations of GCs due to heap coexistence; but for _228_jack and _202_jess, OpenJIT had less GC invocations, by approximately 40% and 5%, respectively. This is attributable to unpredictable behavior of conservative GC in the “classic” Java VM; it is likely that by chance, the collector happened to mistake scalars for pointers on the stack. Neither really contributes significantly to performance differences.

Figure 4.5, 4.6, 4.7, and 4.8 show the timeline track of the amount of heap usage by the Hello benchmark, for interpreter, sunwjit, OpenJIT, and interpreted OpenJIT (OpenJIT-int), respectively. Again, we observe that during bootstrapping, OpenJIT and OpenJIT-int require approximately 700Kbytes of heap space, which is about 2.6 times the heap space as sunwjit and pure interpreter. Since OpenJIT-int does not allocate metaobjects to compile itself, and the amount being consumed to compile methods of other classes are small, we attribute the consumption to the system objects with the libraries being called from OpenJIT, and immediately released.

The Hello benchmark also verifies that there are two phases of execution for OpenJIT. Firstly, there is a bootstrap phase where the entire OpenJIT is aggressively compiled, accumulating multiple compilation contexts in the call chain of the JIT compiler. Thus, the space required is proportional to the critical path in the call chain. Then, it quickly falls off, and transcends into a stable phase where most parts of OpenJIT have been compiled, and only application methods are being compiled and executed.

Figure 4.9, 4.10, 4.11, 4.12, 4.13, and 4.14 show the timeline track of the amount of heap usage for OpenJIT (OpenJIT) and sunwjit, by executing _201_compress, _202_jess, _209_db, _213_javac, _227_mtrt, and _228_jack, respectively.

No matter how the memory is being used, the amount of additional heap space required for recursively compiling OpenJIT will not be a problem for modern desktop environments, in some situations. A typical desktop applications consumes orders of magnitude more space: for example, our
measurements in Figure 4.12 for _213_javac shows it consumes more than 20 Mbytes\(^1\). However, for embedded applications, such an overhead might be prohibitive. As discussed earlier, this could be suppressed using less aggressive, adaptive compilation similar to the SELF compiler\(^56,57\), but it is not clear what strategy will achieve good suppression while not sacrificing performance. In the next section we consider several adaptive strategies for suppression.

4.3 Adaptive OpenJIT Compilation Strategies

There are several criteria in the design space for adaptive compilation in OpenJIT for memory suppression of the bootstrapping phase.

- **Alteration of JIT Compilation Frequency**
  
  The most aggressive strategy will compile each method on its first invocation. We reduce the frequency of compilation using the following strategies, with \(p\) as a parameter (\(p = 2, 4, 8, 16\))
  
  - JIT compile on \(p\)th invocation, deferring to interpretation for the first \(p - 1\) invocations (constant delay).
  - Assign each method a random number between \([0, p-1]\), and compile when the number of invocation reaches that number (random delay).
  - Compile with probability \(1/p\) on each invocation (probabilistic). Reduced probability may increase the execution time.

- **Restriction of Methods Subject to Adaptation**
  
  We could delay compilation for all the methods, or alternatively, only those of the OpenJIT compiler metaclasses. The former obviously will likely consume less space, but the former may be sufficient and/or desirable, as it will not slow down the application itself. We verify this by comparing altering compilation frequency changes to all classes, versus only altering the frequency of OpenJIT method. For the latter, all other methods are compiled on first invocation except for class initializers, which are interpreted.

- **Restriction of number of simultaneous compilations**
  
  We put global restriction on how many compilations can occur simultaneously. This can be done safely without causing deadlocks. Attempt to compile exceeding this limit (\(L = 1, 8\)) will default back to the interpreter. Note that, although simultaneous compilation could occur

\(^1\)\(\text{[33]}\) reports that with exact GC, the actual usage is approximately 6MBytes. The difference is likely to be an artifact of conservative GC, our close examination has shown.
for application methods under multi-threading, this primarily restricts the simultaneous occurrence of deep recursive compilation chains on bootstrapping.

- **Restricting Compilation of `parseBytecode()`**

  This is a special case, as preliminary benchmarks indicated that `org.OpenJIT.ParseBytecode.parseBytecode()`, is quite large for a single method, (1576 lines of source code and 6861 Java VM bytecode instructions), and thus single compilation of this method creates a large structure in the heap space once it is subject to compilation, irrespective of the strategies used. In order to eliminate the effect, we test cases where compilation of `parseBytecode()` is restricted. In the next version of OpenJIT we plan to factor the method into smaller pieces.

According to the Hello benchmark, in when adaptaion is applied to all the methods, combinations of other schemes effectively yielded reduction in the number and size of objects that are allocated during bootstrapping, without significant increase in bootstrap time. On the other hand, restricting compilation of OpenJIT method only did not yield significant results, except for the case when the entire OpenJIT was interpreted, or when `parseBytecode()` was restricted, again, without significant loss of performance.

The table only shows the total memory allocated. In order to characterize the peak memory behavior, we present the timeline behavior in Figure 4.15 and Figure 4.16. Here, for each scheme, the parameter with lowest peak is presented. We observe that, (1) probabilistically lowering the frequency helps reduce the peak usage, and (2) `parseBytecode()` dominates the peak. We are currently conducting further analysis, but it is conclusive that naive frequency adjustment does not help to reduce the peak; rather, the best strategy seems to be to estimate the heap usage based on bytecode length, and suppressing compilation once a prescribed limit is exceeded.
### Table 4.10: Alteration of JIT compilation frequency for all methods

<table>
<thead>
<tr>
<th>criteria</th>
<th>param</th>
<th>method# (openjit)</th>
<th>alloc obj# (openjit)</th>
<th>alloc size[MB] (openjit)</th>
<th>GC#</th>
<th>time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td></td>
<td>457 (128)</td>
<td>90,594 (74,831)</td>
<td>2.905 (2.316)</td>
<td>5</td>
<td>1.270</td>
</tr>
<tr>
<td>constant</td>
<td>2</td>
<td>225 (126)</td>
<td>64,827 (52,194)</td>
<td>2.117 (1.629)</td>
<td>3</td>
<td>1.190</td>
</tr>
<tr>
<td>delay</td>
<td>4</td>
<td>180 (116)</td>
<td>57,568 (46,133)</td>
<td>1.898 (1.445)</td>
<td>3</td>
<td>1.280</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>161 (114)</td>
<td>55,743 (44,470)</td>
<td>1.843 (1.395)</td>
<td>4</td>
<td>1.250</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>151 (113)</td>
<td>54,069 (43,064)</td>
<td>1.793 (1.352)</td>
<td>4</td>
<td>1.810</td>
</tr>
<tr>
<td>random</td>
<td>2</td>
<td>412 (127)</td>
<td>84,739 (69,746)</td>
<td>2.729 (2.162)</td>
<td>5</td>
<td>1.270</td>
</tr>
<tr>
<td>delay</td>
<td>4</td>
<td>301 (123)</td>
<td>72,686 (59,272)</td>
<td>2.357 (1.843)</td>
<td>4</td>
<td>1.190</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>231 (120)</td>
<td>62,628 (50,589)</td>
<td>2.050 (1.578)</td>
<td>4</td>
<td>1.240</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>196 (115)</td>
<td>61,923 (49,790)</td>
<td>2.031 (1.558)</td>
<td>3</td>
<td>1.350</td>
</tr>
<tr>
<td>probability</td>
<td>2</td>
<td>170 (115)</td>
<td>56,383 (45,041)</td>
<td>1.862 (1.412)</td>
<td>4</td>
<td>1.280</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>190 (115)</td>
<td>58,493 (46,877)</td>
<td>1.924 (1.466)</td>
<td>4</td>
<td>1.900</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>149 (115)</td>
<td>53,710 (42,735)</td>
<td>1.780 (1.341)</td>
<td>3</td>
<td>2.010</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>112 (99)</td>
<td>27,066 (21,399)</td>
<td>0.955 (0.664)</td>
<td>1</td>
<td>0.920</td>
</tr>
</tbody>
</table>
### Table 4.11: Alteration of JIT compilation frequency for OpenJIT methods only

<table>
<thead>
<tr>
<th>criteria</th>
<th>param</th>
<th>method# (openjit)</th>
<th>alloc obj# (openjit)</th>
<th>alloc size[MB] (openjit)</th>
<th>GC#</th>
<th>time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td>–</td>
<td>457</td>
<td>90,594</td>
<td>2.905</td>
<td>5</td>
<td>1.270</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(128)</td>
<td>(74,831)</td>
<td>(2,316)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>constant</td>
<td>2</td>
<td>457</td>
<td>90,526</td>
<td>2.907</td>
<td>5</td>
<td>1.240</td>
</tr>
<tr>
<td>delay</td>
<td></td>
<td>(128)</td>
<td>(74,757)</td>
<td>(2,314)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>451</td>
<td>89,616</td>
<td>2.877</td>
<td>5</td>
<td>1.260</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(122)</td>
<td>(73,988)</td>
<td>(2,291)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>449</td>
<td>88,535</td>
<td>2.845</td>
<td>5</td>
<td>1.280</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(120)</td>
<td>(73,179)</td>
<td>(2,265)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>446</td>
<td>85,699</td>
<td>2.760</td>
<td>5</td>
<td>1.800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(117)</td>
<td>(71,112)</td>
<td>(2,200)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>2</td>
<td>457</td>
<td>90,534</td>
<td>2.904</td>
<td>5</td>
<td>1.320</td>
</tr>
<tr>
<td>delay</td>
<td></td>
<td>(128)</td>
<td>(74,765)</td>
<td>(2,314)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>455</td>
<td>90,236</td>
<td>2.895</td>
<td>5</td>
<td>1.220</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(126)</td>
<td>(74,491)</td>
<td>(2,306)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>452</td>
<td>89,780</td>
<td>2.882</td>
<td>5</td>
<td>1.230</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(123)</td>
<td>(74,115)</td>
<td>(2,295)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>450</td>
<td>89,299</td>
<td>2.868</td>
<td>5</td>
<td>1.330</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(121)</td>
<td>(73,763)</td>
<td>(2,284)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>probability</td>
<td>2</td>
<td>450</td>
<td>89,490</td>
<td>2.873</td>
<td>5</td>
<td>1.270</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(121)</td>
<td>(73,884)</td>
<td>(2,288)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>429</td>
<td>83,831</td>
<td>2.703</td>
<td>4</td>
<td>1.980</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(116)</td>
<td>(69,523)</td>
<td>(2,152)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>446</td>
<td>85,722</td>
<td>2.760</td>
<td>5</td>
<td>2.020</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(117)</td>
<td>(71,136)</td>
<td>(2,201)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>441</td>
<td>84,637</td>
<td>2.728</td>
<td>4</td>
<td>0.910</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(112)</td>
<td>(70,295)</td>
<td>(2,117)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>limit</td>
<td>1</td>
<td>457</td>
<td>90,590</td>
<td>2.906</td>
<td>4</td>
<td>2.530</td>
</tr>
<tr>
<td>simultaneity</td>
<td></td>
<td>(128)</td>
<td>(74,821)</td>
<td>(2,316)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>457</td>
<td>90,514</td>
<td>2.903</td>
<td>4</td>
<td>1.410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(128)</td>
<td>(74,751)</td>
<td>(2,314)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>except</td>
<td>–</td>
<td>456</td>
<td>69,463</td>
<td>2.248</td>
<td>3</td>
<td>1.190</td>
</tr>
<tr>
<td>parseBytecode</td>
<td></td>
<td>(127)</td>
<td>(58,430)</td>
<td>(1,788)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenJIT-int</td>
<td>–</td>
<td>329</td>
<td>37,059</td>
<td>1.265</td>
<td>2</td>
<td>1.280</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0)</td>
<td>(31,149)</td>
<td>(0.941)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.15: Timeline behavior of heap usage for adaptive compilation for all methods (best cases).

Figure 4.16: Timeline behavior of heap usage for adaptive compilation for OpenJIT methods only (best cases).
CHAPTER 4. PERFORMANCE ANALYSIS OF OPENJIT

4.4 Chapter Summary

In this chapter, we have performed extensive analysis of the performance characteristics of OpenJIT, both in terms of execution speed and memory consumption. In fact, as far as we know, there have not been any reports on any self-descriptive JIT compilation performance analysis, nor memory consumption reports for any JIT compilers.

In particular, we have shown that (1) JIT compilation speed does not become a performance issue, especially during the bootstrap process when much of the OpenJIT compiler is run under interpretation, (2) memory consumption of reflective JIT compilers, however, could be problematic due to recursive compilation, especially in embedded situations, (3) that there are effective strategies to solve the problems, which we investigate extensively, and (4) that the solutions do not add significant overhead to overall execution, due to (1). In fact, the self-compilation time of OpenJIT is quite amortizable for real applications.
Chapter 5

OMPI: Optimizing C + MPI Programs

Message Passing Interface (MPI) is gaining acceptance as a standard for message-passing in high-performance computing, due to its powerful and flexible support of various communication styles. However, the complexity of its API poses significant software overhead, and as a result, applicability of MPI has been restricted to rather regular, coarse-grained computations. Our system removes much of the excess overhead by employing partial evaluation techniques, which exploit static information of MPI calls. Because partial evaluation alone is insufficient, we also utilize template functions for further optimization. To validate the effectiveness for our OMPI system, we performed baseline as well as more extensive benchmarks on a set of application cores with different communication characteristics, on the 64-node Fujitsu AP1000 MPP. Benchmarks show that OMPI improves execution efficiency by as much as factor of two for communication-intensive application core with minimal code increase. It also performs significantly better than previous dynamic optimization technique.

5.1 Overview

With the proliferation of MPPs, NOWs, and PC clusters, standardized message passing interfaces such as PVM and MPI are becoming increasingly popular. They are used not only for writing new parallel applications, but also as an effective tool for parallelizing existing applications, as well as serving as a runtime message passing library for implementations of parallel languages, such as High Performance Fortran (HPF).

Since MPI was announced at 1994, it is gaining increasing popularity, thanks to its powerful and flexible support of communication, such as different communication contexts via communicators, various synchronous / asynchronous communication modes, derived datatypes, group communicatio-
There have been a number of recent implementations of MPI as well. But, due to the inherent design of its API, the incurred software overhead is large, even compared to previous message passing libraries such as P4 or PVM. This is especially problematic when the hardware latency is low, because much of the benefits of fast networks are lost because of software overhead. This phenomenon not only applies to MPPs, but also to NOWs, where the availability of low-cost, low-latency networks such as the Myrinet is making low-latency communication possible.

As a result, application area of MPI has been somewhat restricted to regular, coarse-grained, and computation-intensive applications. In other words, attaining efficiency in fine-grained, irregular problems using MPI has been difficult. This is unfortunate, since standard message-passing libraries should encompass a wide variety of platforms and applications, including non-numerical applications, which are typically irregular and communication intensive.

There has also been a string of work that has focused on reducing software overhead in message passing as much as possible\[130\]. Notably, with Berkeley Active Messages, the incurred software overhead is in the order of several microseconds. The drawback is the relative lack of power and flexibility, and portability to some extent. Programming with native Active Messages library is much more difficult compared to programming with MPI, because primitives are “lower-level”. Furthermore, current Active Message does not support OS-level multi-threading nor network heterogeneity well\[1\].

The question then is, would it be possible to have the best of both approaches, i.e., would it be possible to have a low-latency, high-performance message passing library, while retaining the flexibility and power of MPI? The answer is effectively yes — in this chapter, we present our OMPI (Optimizing MPI) system, where much of the software overhead is eliminated with partial evaluation techniques, attaining performance which approaches that of Active Messages. C programs that contain MPI function calls are statically analyzed in order to determine which arguments are static, and specialized with respect to those arguments. Because the current partial evaluation techniques are not sufficiently powerful to eliminate all the software overhead, we propose a technique where partial evaluation is combined with selection of pre-optimized template functions. As a result, OMPI guarantees generality and portability of MPI programs, while allowing architecture-specific optimizations to be incorporated at compile-time. OMPI itself is also semi-portable, in that only template functions need to be reimplemented for a particular architecture. This is in contrast to traditional research on MPI implementation, where optimizations were highly architecture-specific.

To validate the effectiveness of our OMPI system, we performed some

---

1Some of these issues will be addressed with Active Messages 2.0.
baseline benchmarks, and more extensive benchmarks on a set of application cores with different communication characteristics, on the 64-node Fujitsu AP1000 MPP. The basic point-to-point latency improved from 338 microseconds to 76 microseconds, for communication intensive CG solver core, speedup of over a factor of two has been achieved. Even compared to traditional run-time optimization employing dynamic caching techniques, our OMPI system was consistently faster. The results show that our system is effective for various patterns of communication, significantly reducing the software overhead.

The rest of this chapter is as follows: Section 5.2 analyzes the source of software overhead in MPI and opportunities of elimination. Section 5.3 describes our OMPI system. Section 5.4 presents results of the benchmark. Finally we summarize this chapter in Section 5.5.

5.2 Analysis of Software Overhead in MPI

We first analyze the source of the software overhead, identifying problems pertinent to message passing libraries in general, and those that are specific to MPI. We then investigate the opportunities for optimization by removing the overhead when static information is exploited.

5.2.1 General Overhead

For most message passing libraries, information on messages can only be obtained at run-time. As a result, the receiver’s buffer must either be somehow allocated dynamically, and the received message must be copied into the buffer. More specifically, the receiver has only two alternatives:

- If `receive` is posted before the actual message arrives, then the message is directly written into the receiver’s buffers and not copied.
- Otherwise, the system allocates a buffer, and messages are written into the buffer; when `receive` is subsequently called, the contents of the system buffer is copied into the user’s buffer.

Techniques to reduce the overhead typically has the sender specify the buffer address directly, such as is possible with the original Active Messages. However, software flexibility is lost as a result; for example, it will not be possible to filter message reception with message tags, as is possible with most message passing libraries. Also, programs must be strictly SPMD, in that addresses of functions must be the same among all the nodes.
CHAPTER 5. OMPI: OPTIMIZING C + MPI PROGRAMS

5.2.2 MPI-Specific Overhead

The general philosophy of MPI design is to provide a rich set of features applicable to a variety of parallel applications. The characteristics of the resulting API of MPI is to have numerous arguments which are known to MPI only at run-time. For example, even the simplest point-to-point send/receive has six arguments, namely: (1) the address of the send/receive buffer, (2) message size, (3) message type, (4) ranks of sender/receiver processes, (5) message tag, and (6) communicator. While such API complexity allows sophisticated support of different application communication patterns, the MPI library embodies the overhead of dynamic allocation of work area, error checking and handling, etc., in addition to the general overhead of message passing libraries.

Here we identify the 4 sets of parameters, which we call communication sets, that are necessary for message passing. As described below, MPI incurs additional overhead compared to traditional, simpler message passing interfaces such as PVM, in order to obtain the communication sets. Later on we describe how OMPI utilizes the communication sets to optimize MPI code.
• **CommBuf**
  Obtained directly.

• **CommSize**
  Obtained from `datatype` and `count`. When `datatype` is a derived datatype, the processing becomes more complex because the datatype structure must be traversed to obtain the exact size of each unit of a datatype.

• **CommNode**
  The physical node-id is obtained from the rank of the given process group specified by the communicator `comm`. When the process group is not the default, which contains all the nodes, some table lookup must be performed to obtain the rank ↦ node-id mapping.

• **CommTag**
  Because MPI can specify both the `tag` argument and the tag specified by the communicator `comm`, the two must be combined to generate a unique tag.

### 5.2.3 Opportunities for Eliminating Overhead

The problems with previous techniques have been that there are limitations in attempting to eliminate software overhead relying only on dynamic information available at run-time. By exploiting static information available at compile time, we can eliminate much of the software overhead as we will show in Section 5.3. Here, we analyze the various types of software overhead in detail, and discuss the opportunities of their elimination by utilizing static information.

The source of software overhead due to lack of exploiting static information can be categorized into the followings:

- Inapplicability of inter-buffer copy elimination techniques as seen in Active Messages.
- Cost of dynamic buffer allocation/deallocation management.
- Necessity of executing error checking and other dynamic conditionals.
- Cost of computing the communication sets themselves.

Thus, static determination of various MPI parameters will allow us to reduce the overhead by eliminating the computations costs incurred above. We further relate the cases when static information is known on the arguments and which of the costs above can be eliminated:
Case 1: When CommSize (datatype and count) are known

- **Elimination of Run-time Checks and Computation of Message Size**
  
  This is especially effective for derived datatypes. Even for primitive datatypes, error checking can be eliminated.

- **Static Allocation and Re-use of Message Buffers**
  
  When the DMA controller is employed for non-blocking transfers other than the ready mode, the system will require its own message buffer. By allocating or scheduling the re-use of such buffers statically, overhead involving buffer management can be drastically reduced.

- **Selection of Optimal Message Passing Procedures**
  
  Various optimization techniques could be employed. As an example, packetization of small messages can be simplified; selection of push-based vs. pull-based messaging can be determined by message size. When fast message passing hardware is available, such as Line-sending in AP1000[104], sender buffering could be eliminated, etc.

- **Simplification/Elimination of Error Checking/Handling**
  
  There are other benefits besides elimination of error checking/handling code. For example, argument range errors could be detected at compile time, increasing system robustness. When there is reliable transport for short messages, the heavyweight transport that handles all message sizes could be bypassed.

Case 2: When CommNode (the sender/receiver rank and the communicator comm) are known

- **Elimination of Runtime Computing of Node-id**
  
  Compared to simpler message passing libraries such as PVM, taking the correspondence of communication contexts is a significant part of MPI message passing, as mentioned earlier. Much of the overhead can be eliminated if both rank and the communicator are known. Even if the rank is not known, search procedures could be specialized to a fixed rank number, etc.

- **Conversion of Self-sending into Local Memory Copy / Elimination of Handlers**
  
  In order to take the correspondence between the sender and the receiver, communication contexts (usually called handlers) are created.

---

2This optimization has been suggested by Marc Snir, but has not been implemented yet in our current system.
For self-sending, which typically occurs in an SPMD program, message passing could be converted into local memory copy, and such handlers need not be created.

- **Simplification/Elimination of Error Checking/Handling**
  For the same reasons mentioned above.

## 5.3 Optimizing MPI Programs Using Partial Evaluation

In order to exploit the optimization opportunities analyzed in Section 5.2, we propose OMPI: a system which optimizes MPI programs using partial evaluation techniques. OMPI works as a preprocessor to programs written in C + MPI, is semi-portable, and do not require customized C compilers, operating systems, or hardware. As the benchmarks will later show, our proposal eliminates much of the overhead analyzed so far, achieving the speed approaching Active Messages, while retaining the generality, flexibility, and portability of the MPI.

### 5.3.1 Architectural Overview of OMPI

Optimization architecture of OMPI, which already has been implemented as a prototype on AP1000, performs the following optimizations automatically, without end-user interventions:

1. Static analysis is performed on the end-user source program written with C + MPI, obtaining static information of the arguments passed onto MPI.

2. Using partial evaluation techniques, the source program is specialized with respect to the MPI library functions.

3. The specialized source program is further optimized so that static information is fully exploited.

Unfortunately, we have found that the static analysis/partial evaluation techniques available today are not sufficiently powerful to effectively eliminate software overheads. Thus, we propose an alternative approach that requires machine-specific optimizations to be prepared by the MPI implementor for the particular machine. To be more specific, pre-specialized functions, called *template functions*, that correspond to each case of possible static/dynamic arguments, are prepared. Then, instead of automatic specialization of MPI functions, we instantiate the template functions with current function argument information, and inline expand the instantiated
function. As we shall see, this technique works quite well in practice. Furthermore, it has the added benefit of being able to incorporate optimizations that are not possible with straightforward partial evaluation techniques. The drawback is that template functions must be tailored for each specific architecture; however, the burden of doing so could be substantially resolved by re-use of existing code and employing tool support.

5.3.2 Static Analysis and Partial Evaluation Using SUIF

OMPI is currently built by extending the SUIF compiler toolkit proposed by Monica Lam et al. at Stanford University\[139, 113\]. The toolkit consists of a definition of an intermediate program representation called SUIF (from where the name is derived), and a set of extensible compiler passes. Each pass takes SUIF intermediate representation of the program, and outputs SUIF intermediate representations with additional information regarding the performed analysis, and/or a result of performing program transformation. The library is implemented as an object-oriented class framework using C++, and is extensible using inheritance. One could add new information to the SUIF node objects, and implement new passes by building on top of existing passes.

As mentioned above, OMPI is built as a preprocessor which passes the optimized MPI program to the backend C compiler of the target machine. We have found SUIF to be well-suited for the purpose, saving us considerable development time and achieving portability at the same time. Furthermore, as other research groups develop relevant optimizers, they could be integrated easily into our system to further optimize MPI programs.

5.3.3 Optimization Passes

Here we describe the optimization passes in more detail, as shown in Figure 5.3:

1. The source program written in C + MPI is transformed into the SUIF intermediate representation by a tool called scc provided by the SUIF system.

2. An optimization pass called peval is applied to the SUIF intermediate representation, performing various static analysis and partial evaluation. peval is OMPI’s customized version of porky of SUIF, adding some features such as constant expression calculation and several specialized interprocedural analyses.

3. A template selection pass tsel is applied, selecting and instantiating the appropriate template functions.
4. **peval** is further applied to the specialized SUIF intermediate representation, propagating the static information inside the instantiated templates, and performing partial evaluation thereof.

5. The optimized SUIF intermediate representation is converted back into a C program by using the **s2c** of SUIF.

**MPI Program Analysis/Optimization Using peval**

**peval** is our core analysis/optimization pass. It is implemented by re-using and extending upon many of the scalar optimizers available in SUIF. Traditional optimizations, such as constant folding, constant propagation, calculation of constant expression, forward propagation, induction variable detection, common sub-expression elimination, dead code elimination, unused symbols elimination, unused type definitions elimination, loop invariant expressions, loop invariant conditionals, control simplification, if hoisting, etc. are performed as source-to-source transformations.

In our current system, we have already implemented interprocedural data-flow analysis\(^3\) for basic datatypes, but array analysis is still in the works (array analysis is important for determining *CommBuf* and *CommSize* inside loops.).

As noted earlier, the **peval** pass is run twice, before and after the **tsel** (template instantiation) pass. The former is run to gather as much information as possible for **tsel**; because all the necessary arguments are passed to the MPI library via arguments and not through globals, we have found that current analysis and optimizations available in **peval** to be often sufficient for making as good a selection of templates as possible.

The post-**tsel** **peval** pass optimizes the internals of instantiated templates, which will be explained in the following sections.

**Template Instantiation with tsel**

The **tsel** pass scans the source program from the beginning, and when it encounters an MPI call, it performs the following actions:

1. Determine whether the arguments to the MPI call has been analyzed as a constant.

2. Select a template function depending on static/dynamic argument information.

3. Replace the MPI call with the instantiated template function call.

\(^3\)Interprocedural analyses can be easily implemented, using the Global Symbol Table of SUIF, which provides method to access symbols and procedures across program hierarchy.
CHAPTER 5. OMPI: OPTIMIZING C + MPI PROGRAMS

Figure 5.3: Overview of optimization passes
4. Append the instantiated template function.

In practice, however, forcing the MPI implementor to create template functions for every possible combinations of static/dynamic arguments is impractical even if we have semi-automated tools, due to combinatorial explosion. Instead, we employ a simpler policy for building and selecting template functions, based on communication sets described in Section 5.2. We group the arguments into those that determine CommSize, and CommNode and others, and only consider the entire communication set to be static if all the member arguments are static; otherwise, the communication set is considered to be dynamic. We further partition the static case of CommSize into short and long messages, and CommNode into local and remote nodes. By eliminating meaningless combinations, we obtain 9 cases of template functions as seen in Figure 5.4, where MPI_Func is the name of an MPI function.

We note that not all the template functions need not be created. In fact, any function can be substituted by a conservative version with respect to its arguments being dynamic. For example, MPI_Send_long could be safely substituted for MPI_Send_long_remote.

The algorithm of selecting a template function depending on static / dynamic argument information is as below:

1. Identify whether CommSize and CommNode is static or dynamic.

CommSize is determined to be static if both datatype and count are constants, otherwise dynamic. Likewise, CommNode is determined to
be static if both rank and comm are constants, otherwise dynamic.

2. If CommSize is static, calculate the size of datatype. The actual CommSize value is obtained by the product of the size and count.

3. Identify CommSize with long, if the value is bigger than the architecture specific threshold. Otherwise, identify with short.

4. If CommNode is static, the physical node-id is obtained from the rank of the given process group specified by the communicator comm. In OMPI runtime, the rank is numbered in order of the physical node-id, therefore, we can obtain the actual CommNode value easily.

5. Identify CommNode with local or remote.

6. According to the abstract value of CommSize (long or short or dynamic) and CommNode (local or remote or dynamic), select a template function from Figure 5.4.

Instantiation of template functions selected by the above algorithm results in inline expansion of the function. In the expansion, all the static arguments appeared in the original MPI call is embedded inside of the instantiated template function definition. We should also note that even with conservative selections, we still may get the benefits of partial evaluation because the static value of the arguments will be propagated within the instantiated template functions in the subsequent peval pass.

5.3.4 An Example of MPI Optimization

As an example, consider MPI_Send in Figure 5.5. Suppose that the preceding peval has output SUIF program which includes the instantiated template call in Figure 5.5(a), where the message count is statically determined to be 10, type to be MPI_INT, rank to be 2, and communicator to be the default MPI_COMM_WORLD. In this case, tsel selects the template function MPI_Send_Short_Remote, which is optimized for sending short messages to a fixed node-id. The library call is transformed into code as shown in Figure 5.5(b), where 0001 is an ID which identifies each instantiation. Finally, the macro and the template function is expanded as seen in Figure 5.5(c). (Macro definition is for readability purposes.) The inline expanded template function has all the constants embedded within the function, which will be subsequently passed onto the next peval pass.

Instantiation of specialized template functions does increase code size. However, since the expansion of MPI functions is not recursive, the only
(a) Original MPI call

```c
MPI_Send(buf, 10, MPI_INT, 2, tag, MPI_COMM_WORLD)
```

(b) Transformed MPI call

```c
MPI_Send_Short_Remote_0001(buf, 10, MPI_INT, 2, tag, MPI_COMM_WORLD)
```

(c) Specialized and expanded MPI call

```c
#define MPI_Send_Short_Remote_0001(buf, count, type, dest, tag, comm)  
  MPI_Send_short_remote_0001(buf, _tag)  

MPI_Send_short_remote_0001(buf, _tag)  
{
  /* preamble */
  const int _count = 10;  
  const MPI_Datatype _type = MPI_INT;  
  const int _dest = 2;  
  const MPI_Comm _comm = MPI_COMM_WORLD;  
  
  /* body */
  ....
}
```

Figure 5.5: An example of MPI optimization via partial evaluation
potential problem is expansion of loops and recursive functions with embedded MPI calls. OMPI avoids this problem by limiting loop expansions to 4 iterations and not expanding mutually recursive functions more than once; as a result code increase is almost proportional to the original code size, and is minimal in practice so far.

5.4 Performance Measurements

In order to validate the effectiveness of our OMPI system, we performed some baseline benchmarks, and also more extensive benchmarks on application cores. The results show that our system is effective for various patterns of communication, and significantly reduces the software overhead, even compared with traditional optimization techniques.

We chose Fujitsu AP1000 as a target of our prototype implementation. As mentioned earlier, communication performance of AP1000 relative to its processor performance is considerably higher (25MBytes/sec node-to-node vs. 25Mhz SPARC IU + FPU). Thus, small software overhead will dominate loss in communication performance. AP1000 facilitates two modes of message communication. One is interrupt-based, and employs the DMA controller. Although the startup overhead is large, the send is nonblocking. The other is Line-sending, where values contained in a cache line could be sent directly with explicit cache flushing, eliminating the need for copying, interrupts, and DMA setup. On the other hand, the sender must block until the entire line is sent. Also, although the receive is transparently done into a ring buffer without interrupts, the value must be copied.

As a baseline for comparison, we chose MPICH, which is a public domain MPI implementation by Argonne National Laboratory and Mississippi State University. In order to port MPICH onto AP1000, one only needs to implement is lower-level communication interface, called Abstract Device Interface (ADI). Since MPICH relies heavily on the performance of ADI functions, it is critical for performance comparisons that ADI functions are implemented efficiently. We reused the source code of tuned native AP1000 message passing libraries to achieve this requirement.

5.4.1 Comparison against Dynamic Caching Optimization

As mentioned earlier, dynamic caching of arguments is a more orthodox and simpler implementation technique compared to ours. More specifically, each MPI function would have its own cache that holds the arguments of its previous calls, and when there is a cache hit, the following optimizations become possible in order to eliminate the overhead:

- Error checking could be eliminated.
Parameter checks for other dynamic optimizations could also be eliminated.

- Message buffers could be reused.

In order to validate the effectiveness of our approach against dynamic caching techniques, we also customized MPICH to incorporate such optimization for comparison purposes. To effectively implement dynamic cache in MPI calls, we employed the Persistent Communication Request (PCR) feature of MPI. The intended use of PCR is in inner loops, where the same MPI function is invoked repeatedly; there, instead of specifying the arguments each time in the iteration, a set of arguments could be registered with PCR using calls such as `MPI_Send_Init` and `MPI_Recv_Init`, and could be invoked repeatedly with `MPI_Start`. By using PCR, dynamic cache can be easily implemented as follows:

- **Preamble upon MPI function call**
  
  Check if the arguments are cached; this is achieved with a cache manager handle as illustrated in Figure 5.6 and checking whether the stored arguments match or not. If it matches, it is a hit, and is a miss otherwise.

- **Cache Miss**
  
  Register the called MPI function and the arguments with PCR. Create the communication handle, and store the arguments, the function ID, and a pointer to the PCR. Finally, invoke the communication with PCR.

- **Cache Hit**
  
  Obtain the PCR from the cache manager handle, and invoke the communication.

For derived datatypes, the argument comparison in the preamble could be costly, as the entire dynamic data structure must be traversed. The simple solution is not to cache such arguments; an alternative strategy is to use a fast but conservative matching function; for example, for systems where the operating system could trap on writes, any writes to a page containing a buffer could invalidate a match by setting some flag.

### 5.4.2 OMPI Implementation on AP1000

In order to customize our system on AP1000, we need to create the template functions, and tailor the selection heuristics of `tsel`.

In the current prototype, template functions for the 9 cases involving the communication sets were hand-created. Because we do
not yet have tool support for creating template functions, only a small subset of MPI has so far been implemented. Here describe the specifics for `MPI_Send` and `MPI_Recv`. For `MPI_Send`, the templates for five cases have the specializations/optimizations described in Table 5.1. The remaining 4 cases (`MPI_Func_long_remote`, `MPI_Func_short_remote`, `MPI_Func_long_local`, `MPI_Func_short_local`) were created by combining the optimizations. Similarly, for `MPI_Recv`, the specializations/optimizations are described in Table 5.2, and likewise the remaining 4 cases were derived by their combination. Optimizations in other MPI functions are similar, which has given us confidence that at least some semi-automated tool could be created to greatly ease the task of template creation.

We also make a note here that, `MPI_Func_generic` (i.e. unoptimized) template functions of OMPI are essentially the same as the corresponding MPICH implementations both in the robustness (e.g. error checking) and execution time.

We also tailor the selection heuristics of `tsel`. When there are multiple message transports, selection of the transport is dependent on `CommSize` and the characteristics of the underlying message passing architecture. On AP1000, in general it is better to employ Line-sending for short messages, whereas DMA+Interrupts is preferable for long messages. For the 64-node platform we employed, our tests showed that the threshold is at 60 Kbytes. This threshold is used for determining whether either `_short_` or `_long_` would be faster.
### Table 5.1: List of building template functions for MPI_Send

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.generic</td>
<td>- do nothing (essentially same as MPICH)</td>
</tr>
</tbody>
</table>
| .short    | - use line-sending method  
|           |   - eliminate allocation/deallocation of send buffer  
|           |   - eliminate error handling and overflow checking  
|           |   - eliminate CommSize calculation |
| .long     | - use DMA+Interrupt method  
|           |   - eliminate CommSize calculation |
| .local    | - use local copying from user buffer to system receive buffer  
|           |   - eliminate allocation/deallocation of send buffer  
|           |   - eliminate error handling and overflow checking  
|           |   - eliminate CommNode calculation |
| .remote   | - eliminate CommNode calculation |

### Table 5.2: List of building template functions for MPI_Recv

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.generic</td>
<td>- do nothing (essentially same as MPICH)</td>
</tr>
</tbody>
</table>
| .short    | - use Buffer-receiving method correspond to Line-sending  
|           |   - eliminate allocation/deallocation of receive buffer  
|           |   - eliminate error handling and overflow checking  
|           |   - eliminate CommSize calculation |
| .long     | - use DMA+Interrupt method  
|           |   - eliminate CommSize calculation |
| .local    | - use local copying from system receive buffer to user buffer  
|           |   - eliminate allocation/deallocation of receive buffer  
|           |   - eliminate error handling and overflow checking  
|           |   - eliminate CommNode calculation |
| .remote   | - eliminate CommNode calculation |
5.4.3 Baseline Performance

We first compare the baseline performance by conducting the basic ping-pong benchmark. In order to obtain realistic figures for a multiprocessing environment, message reception is via interrupts and not polling. We tested both the latency and throughput; here, we only introduce the latency figures (Figure 5.7), as the throughput basically converges to be identical at approximately 60 Kbytes of message size (which was chosen as the threshold).

The leftmost two columns indicate the performance of native AP1000 message passing library for sending a null message. DMA requires 193 microseconds vs. 37 microseconds for Line-sending, and we can see, both hardware and software setup time of DMA is significantly greater. Line-sending latency is close to what one obtains from Active Messages (Latency for polling-based Active Messages on AP1000 has been reported to be approximately 9 microseconds[125]).

The middle three columns are dynamic cache optimized MPI. The software overhead for the initial setup time (i.e., cache miss) is significantly greater compared to DMA, but for cache hits involving basic datatypes, both software and hardware overhead is reduced significantly, coming close to that of DMA. Software overhead reduction is mainly due to elimination of error checks and dynamic computation of target nodes from communication set, as PCR allows such communication contexts to be cached and passed almost
directly to the underlying DMA routine. On the other hand, software over-
head for derived datatype is significant, due to the interpretation/traversal
overhead for cache comparison check mentioned earlier, nullifying the gains
obtained with caching.

The rightmost columns are OMPI results. The “Generic” column is when
there is no static information available; “CommNode” and “CommSize” in-
dicate the cases where the respective communication sets are identified to
be static; and “Both” means that both are known statically. Here, we see
that even with partial information, our optimizations result in significant
overhead reduction. In particular, when CommSize is known, since our mes-
gaes is below the 60Kbytes threshold, the Line-sending was selected, which
greatly reduces the mandatory hardware latency (from 112 microseconds to
24 microseconds). For the “Both” case, the results are dramatic: both hard-
ware and software overhead have been reduced to 1/4 of the unoptimized
generic case, down to 76 microseconds from 338 microseconds. Although
the software overhead is still larger than the native AP1000 message pass-
ing library, we strongly believe that we could close this gap with further
improvements in partial evaluation.

5.4.4 Numerical Application Core Benchmarks

We next compare the performance of MPI optimizations in a typical nu-
merical applications core. We chose three benchmarks with very different
communication patterns and computation vs. communication ratio. They
are as follows:

- **MM**
  1024 \times 1024 Matrix Multiply based on (Cyclic(16), Cyclic) sub-
  matrices distribution. Communication size is 16 Kbytes, communication
  pattern is regular, and is highly compute intensive.

- **LU**
  Linpack 1000x1000 matrix LU-factorization with single pivot row se-
  lection and (Cyclic(8), Cyclic) distribution. Communication size is
  1 Kbyte, and communication pattern cannot be entirely determined
  statically due to run-time pivot selection. Less compute intensive com-
  pared to **MM**.

- **CG**
  Conjugate Gradient solver over a 128 \times 128 sparse matrix. Dot dis-
  tribution is employed. Communication size is one floating number,
  communication pattern is regular, and is highly communication inten-
  sive.
We compared the execution times of the above benchmarks for generic (unoptimized), dynamic cache optimized, and OMPI. Furthermore, the execution times were categorized into time spent for computation, time spent within MPI executing library code, and time spent within MPI waiting for barrier and communication synchronization. Figure 5.8 presents the results.

For MM, speedup is minimal, because the speedup obtained within MPI library is only a very small fraction of the entire execution time. Still, OMPI won by a small margin. For CG, the other extreme, OMPI was able to win by a significant factor over both generic and cache implementations, with 53% reduction in execution time.

An interesting result was obtained for LU; we initially speculated that this benchmark would be disadvantageous for OMPI, as the communication pattern cannot be determined at compile time. The result surprisingly indicated otherwise, significantly improving over the generic MPI and winning over dynamic cache MPI by a notable factor. Closer analysis revealed that the PCR cache was being invalidated due to irregularity in the communication pattern of LU, and also that the cost of cache management was adding considerable overhead. By tuning the dynamic cache optimization, e.g., by not relying on the PCR, such overhead could be reduced. Still, the benchmark shows that, even for irregular communication, our partial evaluation strategy eliminates considerable portion of software overhead of MPI.

![Figure 5.8: Numerical application core benchmarks](image-url)
5.5 Chapter Summary

In this chapter, we have presented OMPI (Optimizing MPI), a compile-time optimizer for C + MPI programs that eliminates much of the communication overhead using partial evaluation techniques.

In particular, we analyzed the source of the software overhead, identifying problems pertinent to message passing libraries in general, and those specific to MPI. Then, we investigated the opportunities for optimization by removing the overhead when static information was exploited.

And, we proposed the OMPI system in order to exploit the optimization opportunities analyzed. It works as a preprocessor to programs written in C + MPI, is semi-portable, and do not require customized C compilers, operating systems, or hardware.

Moreover, to validate the effectiveness of our OMPI system, we performed some baseline benchmarks, and also more extensive benchmarks on numerical application cores. The results have shown that our system is effective for various patterns of communication, and significantly reduces the software overhead, even compared with traditional optimization techniques, such as dynamic caching. In other words, our proposal eliminates much of the overhead analyzed so far, achieving the speed approaching Active Messages, while retaining the generality, flexibility, and portability of the MPI standard.
Chapter 6

Conclusion

In this thesis, we have addressed our research and experience of designing and implementing OpenJIT, an open-ended reflective JIT compiler framework for Java, and OMPI, a compile-time optimizer for C + MPI programs.

OpenJIT is an open-ended, reflective JIT compiler framework for Java written almost entirely in Java itself. Although in general self-descriptive systems have been studied in various contexts such as reflection and interpreter/compiler bootstrapping, OpenJIT is a first system we know to date that offers a stable, full-fledged Java JIT compiler that plugs into existing monolithic Java VMs, and offer competitive performance to JIT compilers typically written in C or C++. At the same time, OpenJIT is designed to be a compiler framework in the sense of Stanford SUIF, in that it facilitates high-level and low-level program analysis and transformation framework for the users to customize.

OMPI, on the other hand, is a compile-time optimizer for C + MPI programs based on Stanford SUIF compiler framework, which eliminates much of the communication overhead using partial evaluation techniques which exploit static information of MPI calls in the target programs. Because partial evaluation alone is insufficient, we also utilize \textit{template functions} for further optimization. Benchmarks showed that OMPI improves execution efficiency by as much as factor of two for communication-intensive application core with minimal code increase.

These systems enable compile-time customizations and optimizations in Java and C + MPI, and will contribute to achieve both of runtime efficiency and maintainability of large-scale and complicated software, especially with dynamicity, portability, and concurrency.
6.1 Technical Contributions

OpenJIT

As for OpenJIT, an open-ended reflective JIT compiler framework for Java, we have presented the following technical contributions:

1. We proposed an architecture for a reflective JIT compiler framework on a monolithic "classic" Java VM, and identified the technical challenges as well as the techniques employed. The challenges exist for several reasons, that the JIT compiler is reflective, and also the characteristics of Java, such as its pointer-safe execution model, built-in multi-threading, etc.

2. We showed an API that added to the existing JIT compiler APIs in "classic" Java VM to allow reflective JIT compilers to be constructed. Although still early in its design, and requiring definitions of higher-level abstractions as well as additional APIs for supporting JIT compilers on more modern VMs, we nonetheless presented a minimal set of APIs that were necessary to be added to the Java VM in order to facilitate a Java JIT compiler in Java, contrasting to similar work such as Jikes RVM.

3. We demonstrated a small example of how reflective JIT compilers could be useful class-specific or application-specific customization and optimization by defining a compilet which allowed us to achieve 8-9% performance gain without changing the base-level code. Although the current examples are small, we nevertheless present a possibility of larger-scale deployment of OpenJIT for uses in the above-mentioned situations.

4. We performed extensive analysis of the performance characteristics of OpenJIT, both in terms of execution speed and memory consumption, using collaborative instrumentation technique between the Java VM and OpenJIT, which allowed us to instrument the JIT performance in real-time. In fact, as far as we know, there have not been any reports on any self-descriptive JIT compilation performance analysis, nor memory consumption reports for any JIT compilers.

And we showed that OpenJIT was quite competitive with existing, commercial JIT systems, and some drawbacks in memory consumption during the bootstrap process could be circumvented without performance loss. To be more particular, we showed that (1) JIT compilation speed does not become a performance issue, especially during the bootstrap process when much of the OpenJIT compiler is run under interpretation, (2) memory consumption of reflective JIT compilers,
However, could be problematic due to recursive compilation, especially in embedded situations, (3) that there are effective strategies to solve the problems, which we investigate extensively, and (4) that the solutions do not add significant overhead to overall execution, due to (1). In fact, the self-compilation time of OpenJIT was quite amortizable for real applications.

OpenJIT is based on the early implementation of Java VM (“classic" VM), which is a representative of the “JIT-only” VMs, as well as Smalltalk-80 and the first generation of SELF, as we have described in Section 2.1.1. Therefore, one might assume that OpenJIT provides a limited solution only applicable to the past and obsolete Java VMs. But, OpenJIT can also be considered to be a bridging technology from the first generation JIT compilers to the second generation JIT compilers. Due to the customizability of OpenJIT, we can extend and customize a JIT compiler itself to support runtime profiling of method invocations, adaptive compilation, recompilation, and so on. In fact, we introduced an experimental result to adapt OpenJIT several adaptive compilation strategies for memory suppression of the bootstrapping phase (in other words, we can determine hot methods needed to be compiled at JIT compilation time, and compile only them for faster execution.), as well as to perform extensive analysis to prove their effectiveness, in Section 4.3. Moreover, today, JavaScript implementations with the “JIT-only” VMs such as V8 or Tracing JIT, are gaining popularity as a faster browser-based application platform. In order to realize faster compilation on a limited compilation resource, those systems only perform a direct transformation from an AST or SSA form representation to native code. We believe that OpenJIT will provide them a practical and/or theoretical base of realizing the extensibility and customizability atop them.

There have been several work in self-descriptive object-oriented systems targeted to Java, such as JavaInJava, Rivet, MyJVM, Jikes RVM, and so on. Except Jikes RVM, all the other systems only realize Java VM in Java and demonstrate how such implementations are not so inefficient because of the use of the underlying Java VM and JIT compiler. Jikes RVM is an aggressive effort in building not only the JIT compiler, but the entire Java VM in Java. On surface, one might assume that building a self-descriptive Java system subsumes building only the JIT compiler in Java. But the technical goals and achievements, as well as the advantages and the disadvantages of OpenJIT and Jikes RVM, are different, and could be even considered complementary. Detailed discussion between these systems has been described in Section 2.1.2.

We also know of several work in implementing compiler frameworks targeted to Java, namely Marmot, BLOAT, SableVM and Soot, and Joeq VM. Except Joeq VM, all the other frameworks just work as static optimizing compilers or AOT compilers, and they don’t support
any program transformations at the JIT compile-time. While Joeq VM provides a full-fledged compiler framework and supports both of JIT and AOT compilations, OpenJIT provides a limited compiler framework compared to Joeq VM and supports only JIT compilation. To compromise this disadvantage, however, OpenJIT can cooperate with source-to-bytecode compile-time and/or bytecode load-time reflective systems. Moreover, while OpenJIT supports higher and lower intermediate representations such as AST, stack-based IL, and register-based RTL, Joeq VM supports only lower ones, such as Bytecode IR (similar to our stack-based IL) and Quad IR (similar to our RTL). Thereby, OpenJIT could be better in realizing higher-level program transformations than Joeq VM, but further studies are required to prove this observation.

There have also been a number of work in practical reflective systems targeted to Java, such as OJ (OpenJava)\(^{123}\), AspectJ\(^{70}\), jContractor\(^{66}\), Javassist\(^{124}\), and so on, as we described in Section 2.1.4. These systems realize compile- or load-time program transformations on Java source codes or bytecodes. OpenJIT, on the other hand, is the sole instance that can reflectively customize and extend the behavior of the JIT compiler. OpenJIT can also perform bytecode-to-bytecode structural program transformation using OpenJIT frontend system. The disadvantage in OpenJIT is that structural reflection to the bytecodes never occurs when they are interpreted and executed by the Java VM. Therefore, the effectiveness of the OpenJIT frontend system is considered to be limited. But, OpenJIT can cooperate with structural rewriting toolkits such as OJ or Javassist and perform reflective modifications to the target programs at pre-JIT compile-time, namely at source-to-bytecode compile-time or at bytecode load-time.

**OMPI**

Also, as for OMPI, a compile-time optimizer for C + MPI programs, we have presented the following technical contributions:

1. We first analyzed the source of the software overhead, identifying problems pertinent to message passing libraries in general, and those that were specific to MPI. We then investigated the opportunities for optimization by removing the overhead when static information was exploited.

2. In order to exploit the optimization opportunities analyzed, we proposed OMPI, a system which optimizes MPI programs using partial evaluation techniques. OMPI works as a preprocessor to programs

---

\(^{1}\)Actually, Joeq VM can use the SUIF intermediate representation as an input program. Therefore, higher-level program transformations is considered to be performed using SUIF compiler toolkit.
written in C + MPI, is semi-portable, and do not require customized C compilers, operating systems, or hardware.

3. In order to validate the effectiveness of our OMPI system, we performed some baseline benchmarks, and also more extensive benchmarks on application cores. The results showed that our system was effective for various patterns of communication, and significantly reduced the software overhead, even compared with traditional optimization techniques. In other words, our proposal eliminates much of the overhead analyzed so far, achieving the speed approaching Active Messages, while retaining the generality, flexibility, and portability of the MPI.

As far as we know, all the efforts to lower communication latency in MPI have been to tune the libraries so that their software overhead becomes minimal, given the fact that all the arguments are dynamic. None have employed static compiler techniques to improve performance, other than our research.

Today, MPICH and Open MPI are known as the most popular and widely portable implementations of MPI standard. Even in BlueGene/L and SiCortex with special network hardware, MPI implementations start with MPICH as a basis and specialize it to their network fabrics. These implementations have more or less employed component architectures, which allow users and MPI developers to select and employ several network device drivers simultaneously in a single MPI process at run-time. These architectures are convenient to support special network hardware or hierarchical network environments with heterogeneity, and to distribute third party software supporting only binary distribution if possible. However, they introduces still more software overhead needed to be eliminated, as we eliminated software overhead from the AP1000 MPI implementation with using OMPI.

Utilizing template functions, as we proposed, will benefit “separation of concerns” in implementing complex network drivers to some extent. While component architectures that MPICH and Open MPI support are useful for implementing each drivers for each network devices, OMPI template mechanism will ease the burden to implement network device drivers that have various communication modes, such as Line-sending and DMA in AP1000.

One might think that software overhead in communication libraries is no more a big issue in today’s hardware, because the performance of the current mainstream or server CPUs is far higher and richer than that of AP1000, 25Mhz SPARC IU + FPU. This observation is partially true, but in order to utilize multicores and/or multisockets architectures fully with using message-passing programming model, software overhead is still a big issue needed to be eliminated by compiler-based approach as we showed in this thesis.
6.2 Future Directions

OpenJIT

Possible future directions of our OpenJIT system are as follows:

- **Supporting High-performance Parallel Programs**
  One of the main objectives of the OpenJIT is to realize portable high-performance computing across a wide-variety of machines over the Internet. There, high-performance parallel programs (written in Java and compiled into Java bytecode) will be downloaded and executed on diverse platforms, from single-node computers to large-scale parallel clusters and MPPs. For this purpose, different parallel programming models will have to be supported. In particular, since Java’s natural model of parallel programming is threads in shared memory space, one needs to support such a model for diverse set of architecture, while preserving *performance portability*. This is not just a slogan; in fact as a pilot study precursor to OpenJIT, we have implemented a set of portable compiler metaclasses with OpenC++\[19\] (a reflective compiler for C++) that implements DSM in a portable way\[110\], called OMPC++. Early benchmark results for OMPC++ using numerical core programs written in shared-memory SPMD-style programs (a fast parallel CG-kernel, and a parallel FFT from SPLASH2) has so far shown that, our reflective DSM implementation scales well, and achieves performance competitive with that of high-performance SMPs, such as SparcServer 4000 which has dedicated and expensive hardware for maintaining hardware memory consistency.

Additionally, since the current version of OpenJIT cannot utilize floating-point registers as much as possible, especially for SPARC architecture, OpenJIT is not always superior to sunwjit on typical numerical applications represented by SciMark 2.0 benchmarks. To support high-performance parallel programs sufficiently, we need to brush up on our native code generator.

- **Porting to Modern Java VMs**
  Unfortunately, the current version of OpenJIT is based on the early implementation of Java VM (“classic” VM). Therefore, one might assume that OpenJIT provides a limited solution only applicable to the past and obsolete Java VMs. To maximize the feasibility of our system, we should also investigate the port of OpenJIT to other systems, including more modern Java VMs such as Sun Microsystems’ research JVM (formerly EVM) or OpenJDK\[2\]. In the due process we need to in-

---

1 In November 2006, Sun Microsystems Inc. open sourced the Java programming language compiler (javac) and the Java HotSpot Client VM under the GPL version 2 ([118]).

---
VESTIGATE AND IDENTIFY THE HIGH-LEVEL, GENERIC API FOR PORTABLE INTERFACE TO VARIOUS VMs.

WE ARE CURRENTLY REDESIGNING THE BACKEND SO THAT IT WILL BE SUBSTANTIALLY EXTENSIBLE, AND BETTER PERFORMING. THE FRONTEND REQUIRES SUBSTANTIAL WORK, INCLUDING SPEEDING UP ITS VARIOUS PARTS AS WELL AS ADDING HIGHER-LEVEL PROGRAMMING INTERFACES. DYNAMIC LOADING OF NOT ONLY THE COMPLETS, BUT ALSO THE ENTIRE OPENJIT SYSTEM, IS ALSO A MAJOR GOAL, FOR LIVE UPDATE AND LIVE CUSTOMIZATION OF THE OPENJIT. WE ARE ALSO WORKING ON SEVERAL PROJECTS USING OPENJIT, INCLUDING A PORTABLE DSM SYSTEM [109], NUMERICAL OPTIMIZER, AND A MEMORY PROFILER Whose EARLY PROTOTYPE WE EMPLOYED IN THIS WORK. THERE ARE NUMEROUS OTHER PROJECTS THAT OTHER PEOPLE HAVE HINTED; WE HOPE TO SUPPORT THOSE PROJECTS AND KEEP THE DEVELOPMENT GOING FOR THE COMING YEARS, AS OPEN-ENDED JIT COMPILERS HAVE PROVIDED US WITH MORE CHALLENGES AND APPLICATIONS THAN WE HAD INITIALLY FORESEEN WHEN WE STARTED THIS PROJECT ALMOST TEN YEARS AGO.

OMPI

And, possible future directions of our OMPI system are as follows:

- **Making Use of Compiler and Static Optimization Techniques**
  Some issues we share in common with elaborate compilation techniques, such as separate compilation and debugging of optimized code. We believe that solution techniques in advanced compilers could be applied. Furthermore, since the user can always fall back to non-optimized version of MPI, it is possible for the user to fully debug his code before applying OMPI.

  There are other static optimization techniques that could be applied. For example, we could perform more extensive static analysis, such as variable range checking, which would be effective in eliminating many of the checks even if we do not have full static information. Another is communication rescheduling; even a simple algorithm would be effective in grouping the communication, and applying techniques such as message vectorization and piggybacking [63]. More elaborate communication rescheduling techniques will allow further optimizations. We are also considering combining our techniques with dynamic optimization techniques.

- **Ease of the Effort to Implement Template Functions**
  One of the technical challenges with our MPI optimization is how to ease the effort of implementation of template functions. Currently, we are taking three approaches to this problem. One is classic software
engineering, that is to separate out the machine-independent optimizations from machine-specific optimizations. Another is semi-automated tool support: a software tool could aid the user in specializing his code, by semi-automatically generating the code the user starts out with, given the static / dynamic distinctions of the arguments. The tool could also be supplied with the characteristics of the underlying hardware and operating system (latency / bandwidth of different network interfaces, polling / interrupt-driven / buffered, single / multiprocessing, etc.) and further select or eliminate parts of code, in a similar manner as the current partial evaluator.

Another interesting approach is to implement the core functionality of a subset of MPI, and implement more sophisticated functionalities be implemented in terms of the core subset, and optimized via our MPI optimizer by expanding them all out with partial evaluation. By taking care not to implement MPI functions to be mutually recursive, such recursive expansion via partial evaluation should terminate in a few iterations. Indeed, the MPI standard defines an official subset, whereby other MPI functionalities could be implemented — we must investigate whether the official subset will be just enough for our purpose, in terms of its functionality and the speed of the resulting implementation.

- **Adapting to MPI 2.0 and MPI 3.0**

  It is an interesting research and design issue how much of the new features introduced by MPI 2.0 and currently proposed for MPI 3.0 could be superseded by optimization techniques such as ours. Indeed, some of the new proposals are fundamentally beneficial, such as threads, but there could be some features which might not be necessary, and would otherwise will have unsatisfactory effect on the current execution model and/or the MPI API.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


Appendix A

OpenJIT Runtime Specification

This internal specification covers the runtime portion of the OpenJIT backend compiler for SPARC V8 and Intel x86 CPU. We outline the structure of this specification document below. The OpenJIT Backend compiler largely consists of the compiler part and the runtime part; this document covers the runtime part.

A.1 Java Native Code API

The Java Development Kit (JDK) has several APIs for JIT compilers. OpenJIT is plugged into a given JDK using this API. We first explain the API, and describe the implementation in OpenJIT.

A.1.1 JIT Compiler Initialization

When JDK starts up, it first reads the Java system class (java.lang.System), and then loads the java.lang.Compiler class. java.lang.Compiler is defined as follows:
APPENDIX A. OPENJIT RUNTIME SPECIFICATION

public final class Compiler {
    private Compiler() {} // don't make instances
    private static native void initialize();
    static {
        try {
            String library = System.getProperty("java.compiler");
            if (library != null) {
                System.loadLibrary(library);
                initialize();
            }
        } catch (Throwable e) {
            ...
        }
    }
    ...
}

When this class is loaded, the class initializer is executed, and looks for the property "java.compiler". On JDK 1.1.x, the user either specifies the compiler via a command-line option "-Djava.compiler=XXX", or sets the environment variable JAVA_COMPILER, allowing the system to dynamically link the library via System.loadLibrary. Then, the native method initialize() is invoked. This method is defined in C as follows:

```c
void java_lang_Compiler_initialize(Hjava_lang_Compiler *this) {
    void* address =
    (void*)sysDynamicLink("java_lang_Compiler_start");
    if (address != 0) {
        (*(void (*)(void **)) address)(CompiledCodeLinkVector);
    }
    compilerInitialized = TRUE;
}
```

By defining the function java_lang_Compiler_start() in the OpenJIT native library, this function is thereby invoked by the JVM, allowing proper initialization of OpenJIT. The argument CompiledCodeLinkVector passes the necessary values for JIT compilation; it essentially is a vector of hook functions for JIT compilation, and by modifying the vector appropriately, the JIT compiler is invoked appropriately by the JDK when needed. Some essential hook functions are described in Table A.1.
### Table A.1: Essential hook functions

<table>
<thead>
<tr>
<th>Function name</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitializeForCompiler</td>
<td>On class load</td>
</tr>
<tr>
<td>invokeCompiledMethod</td>
<td>On method invocation</td>
</tr>
<tr>
<td>CompiledCodeSignalHandler</td>
<td>On signal occurrence</td>
</tr>
<tr>
<td>CompilerFreeClass</td>
<td>When class is no longer needed</td>
</tr>
<tr>
<td>CompilerCompileClass</td>
<td>Compilation of specified class</td>
</tr>
<tr>
<td>CompilercompileClasses</td>
<td>Compilation of specified classes</td>
</tr>
<tr>
<td>CompilerEnable</td>
<td>Enable Compiler</td>
</tr>
<tr>
<td>CompilerDisable</td>
<td>Disable Compiler</td>
</tr>
<tr>
<td>ReadInCompiledCode</td>
<td>Load pre-compiled code</td>
</tr>
<tr>
<td>PCinCompiledCode</td>
<td>For exception handling</td>
</tr>
<tr>
<td>CompiledCodePC</td>
<td>For exception handling</td>
</tr>
<tr>
<td>CompiledFramePrev</td>
<td>For exception handling</td>
</tr>
</tbody>
</table>

### Figure A.1: Flow of OpenJIT compiler execution
A.1.2 Overview of JIT Compilation Process

Figure A.1 illustrates the overview of JIT compilation process. When Java is invoked with a JIT compiler specified, the native portion of the JDK is dynamically linked into the JVM. By setting the hook functions appropriately as described above, the class loader will always call the `InitializeForCompiler()` function on each class load.

`InitializeForCompiler`, in turn, modifies the invoker of all the methods defined for the class (except for the native and abstract methods) so that the JIT compiler is invoked for dynamic compilation from the bytecode to the native code. When the method is compiled, the method invocation flag is modified so that the compiled code is now executed directly, and passes on the control to the generated native code. Thereafter, JVM invokes the compiled native code directly by the virtue of the flag.

When a compiled native code calls another compiled native code, the `CompiledCode` field of the `methodblock` structure is read and the control is transferred directly via a jump instruction.

The compiled native method accumulates in memory. The space is reclaimed when the hook function `CompilerFreeClass` is called when the JDK actually deletes the class from memory. The hook function in turn frees the memory for the native code as well.

We have overviewed the JIT compilation process. As we can see, the unit of dynamic compilation is each individual method, and compilation happens on the first time the method is invoked. By all means one can have adaptive strategies to compile on nth invocation, etc.

A.1.3 JIT Compiler Initialization

In OpenJIT, the initialization routine `java_lang_Compiler_start` performs the following task:

1. Initialization of the compiler lock variable:
   In a multithreaded environment one must prevent OpenJIT from compiling the same method with multiple threads at the same time. We thus use a lock variable (`_compile_lock`), which is initialized here.

2. Load `org.OpenJIT.Sparc` class:
   In order to obtain the pointer to the OpenJIT class structure (`ClassClass` structure), the OpenJIT classes are loaded. OpenJIT instantiates the OpenJIT compiler classes before it starts compilation of a Java method; the pointer is required for this.

3. Loading of the `org.OpenJIT.ExceptionHandler` class:
   Similarly, in order to instantiate at compile time, we obtain the pointer to `org.OpenJIT.ExceptionHandler` class by loading it.
4. Obtain pointer to the `OpenJIT_compile()` function:

   We next obtain the pointer to the upcall entry point for OpenJIT Java classes by calling the `ResolveClassConstant()` function, obtaining the methodblock structure for `OpenJIT_compile()`.

5. Initialization of the `org.OpenJIT.Compile` class:

   We initialize several internal native variables such as debugging info variables, floating point constants, and class variables for entry point (address) of runtime routines.

6. Initialization of the `CompiledCodeLinkVector`:

   We set the essential hook functions as described in A.2.

7. Re-initialization Resetting of method invocation functions of the classes already loaded:

   Although not a problem for standard JIT compilers which are present from the point of invocation of the JDK, for OpenJIT several system classes are already loaded when this initialization function is called. Such classes if left alone are never compiled if we merely set the hook functions in the `CompiledCodeVector`. Instead, we must re-initialize the classes already loaded by resetting the method invocation functions etc. of the already loaded classes, in order to allow them to be compiled as well.

A.1.4 Class Initialization

```c
static void OpenJIT_InitializeForCompiler(ClassClass *cb)
```

Given a class, set the invoker functions of all the methods except for the native methods and the abstract methods so that they are dynamically compiled on invocation. Also, in order to allow calls to a compiled method from another compiled method, the compilation must set the `CompiledCode` field of the methodblock structure.

Also, we set the `CompiledCodeFlags` of the methodblock structure depending on the type of the return value of the method. This value is used in `dispatchVM()` function when the compiled native calls the JVM for interpretive execution.

Furthermore, by setting a command-line option, we can restrict the classes and methods to be compiled by calling the function `match_compile_methods`. We pass the methodblock structure to the function, and if the return value is TRUE then the method is subject to compilation, whereas if FALSE then the method is not to be compiled.
A.1.5 Signal Handler

```
static void OpenJIT_SignalHandler(int sig, siginfo_t *info,
                                 ucontext_t *uc)
```

Called when a signal occurs. OpenJIT generates Java exceptions with Unix signals. This function looks at the signal for Java exceptions, and routes control to an appropriate handler. If it receives a signal that is irrelevant to Java exceptions, it simply returns. For details of the exception handling please refer to Section A.3.

A.1.6 Freeing a Class

```
static void OpenJIT_CompilerFreeClass(ClassClass *cb)
```

When JDK decides that a certain class is no longer necessary, this function is called, freeing the space occupied by the compiled native code.

A.1.7 Compiling a Class

```
static bool_t OpenJIT_CompilerCompileClass(ClassClass *cb)
```

Called from a java user program with the following: `java.lang.Compiler.compileClass(clazz)`. This forces compilation of all methods that have not been compiled in the given class.

A.1.8 Enabling the JIT Compiler

```
static void OpenJIT_CompilerEnable()
```

Called from a Java user program with the following: `java.lang.Compiler.enable()`. The methods that are called following this call will be compiled.
A.1.9 Disabling the JIT Compiler

```c
static void OpenJIT_CompilerDisable()
```

Called from a Java user program with the following: `java.lang.Compiler.disable()`. The subsequent methods called after this call will not be compiled. Note that, for the default OpenJIT implementation where each method is compiled on its first invocation, the caller of this method will have been already compiled and thus will execute as compiled native code.

A.1.10 Compiled Code Execution Test

```c
static bool_t
OpenJIT_PCinCompiledCode(caddr_t *pc, struct methodblock *mb)
```

Judges whether the current execution is within a given method using the program counter and the `methodblock` structure. This function is used by the JDK when an exception occurs and it traces and displays the stack trace of execution. If the method is being executed, then it returns TRUE, otherwise FALSE.

A.1.11 The Value of the Program Counter

```c
static unsigned char *
OpenJIT_CompiledCodePC(JavaFrame *frame, struct methodblock *mb)
```

Returns the value of the program counter given the frame and the `methodblock`. This function is used by the JDK when an exception occurs and it traces and displays the stack trace of execution.

NOTE: For OpenJIT, for simplification this function does not return the correct value of the PC, but rather returns the entry address of the compiled native code of given `methodblock`. Everything seems to work fine under this simplification for JDK 1.1.x and JDK 1.2, but other JDKs might break this assumption.
A.1.12 Generating Java Stack Frame

```c
static JavaFrame *OpenJIT_CompiledFramePrev(JavaFrame *frame, JavaFrame *buf)
```

Converts the native compiled code stack frame into Java stack frame used by the JDK (`JavaFrame`). This function is used by the JDK when an exception occurs and it traces and displays the stack trace of execution.

The generated code by the OpenJIT follows the C stack frame convention, and this function performs the conversion under that assumption. For JDK 1.1.x, the `JavaFrame` structure only utilizes the current method and the vars frame; thus, in practice these are the only two fields set by the function. As the converted frame must use the buf memory region, the function sets the values and returns buf.

### A.2 Method Invocation

For each method, both JIT compilation and transfer of control to the native method happens at the point of the subject method invocation.

The JVM interpreter loop is structured as follows. When a method is invoked, the invoker function of the `methodblock` structure `mb` is called. Under interpretive execution, this in turn calls the JVM to generate a new Java stack frame. The first argument of `invoker()` is a pointer to a class object for static method calls, and is the pointer to the invoked object on normal method calls. The second argument `mb` is a pointer to the `methoblock` structure, and the third argument `args_size` indicates the types of the arguments. The 4th argument `ee` is a pointer to the execution environment structure `ExecEnv`.

```c
while(1) {
    get opcode from pc
    switch(opcode) {
        ...(various implementation of the JVM bytecodes)
        callmethod:
            mb->invoker(o, mb, args_size, ee);
            frame = ee->current_frame; /* setup java frame */
            pc = frame->lastpc; /* setup pc */
            break;
    }
}
```
A.2.1 Invoking the Compiler from the JDK

As mentioned earlier, we substitute the value of the invoker to OpenJIT_invoke when the class is loader. The OpenJIT_invoke function is defined as follows in C:

```
bool_t OpenJIT_invoke(JHandle *o, struct methodblock *mb, int args_size, ExecEnv *ee);
```

This function in turns upcalls the OpenJIT_compile to dynamically compile the method. Thereafter, the control is transferred to mb->invoker, transferring control to the just compiled method.

A.2.2 Compiling a Method

A method is compiled and the invoker as well as the CompiledCode fields of the methodblock structure are initialized.

```
void OpenJIT_compile(struct methodblock *mb)
```

This function performs the followings:

1. Mutual exclusion to prevent simultaneous compilation of the same method. As mentioned earlier, we prevent multiple threads from compiling the same method at the same time with proper mutual execution using a compile lock.

2. Setup of invoker and CompiledCode fields in the methodblock structure. When a method is invoked during compilation, we set the invoker and CompiledCode so that the interpreter is invoked. This allows natural handling of recursive self-compilation of OpenJIT compiler classes.

3. Invocation of the dynamic compiler (OpenJIT_Sparc_compile()). The Java method is upcalled to perform the actual compilation.

4. When the compilation is successful: We set the stub function in the invoker field of the methodblock structure so that the compiled native code is invoked.

5. When compilation fails: The methodblock field values are restored to their original values.
A.2.3 Invoking the Compiled Code from the JVM

The `CompileMethod` function sets the value of the invoker to one of the following stub functions according to the return type of the method. The last letter of the function indicates the return type: `V`: void, `I`: int, `J`: long, `F`: float, `D`: double. Other types such as `Object`, `short`, `char`, and `byte` that could be encoded in 1 word use `I` as a default.

```c
bool_t invokeCompiledCodeV(JHandle *o, struct methodblock *mb, int args_size, ExecEnv *ee)
bool_t invokeCompiledCodeI(JHandle *o, struct methodblock *mb, int args_size, ExecEnv *ee)
bool_t invokeCompiledCodeJ(JHandle *o, struct methodblock *mb, int args_size, ExecEnv *ee)
bool_t invokeCompiledCodeF(JHandle *o, struct methodblock *mb, int args_size, ExecEnv *ee)
bool_t invokeCompiledCodeD(JHandle *o, struct methodblock *mb, int args_size, ExecEnv *ee)
```

These stub function performs the following:

1. Saving of the Java frame.
2. Setting up of the equivalent native code stack frame.
3. Calling of the function `makeCompiledFrame`. A dummy JVM frame is created for exception handling and Java reflection, etc.
4. Necessary exception of the native frame by manipulating the stack pointer `%sp`.
   The `invokeCompiledCode` is four arguments; should there be more arguments in the call, then the save area for the arguments must be allocated on the stack by bumping the `%sp`.
5. Set pointer to the `methodblock` structure in register `%3`: Required for compiled code calling convention. For details, refer to the descriptions of the compiled code to compiled code details.

For SPARCs, we pass up to 6 arguments via registers. As such, we fetch 6 values from the JVM operand stack and store them into registers prior to the call. Note that for the current implementation, we ALWAYS pass six values, rather than case analyzing for fewer arguments. In practice this has proven to be effective.
APPENDIX A. OPENJIT RUNTIME SPECIFICATION

7. After method execution, the return value is pushed onto the JVM frame according to the return type. The JVM stack is adjusted as well, and the frame is restored.

8. Check for exceptions. If an exception has occurred, return TRUE otherwise FALSE.

Steps 2–4 are already packaged in the Macro `INVOKE_COMPILED_CODE`.

### A.2.4 Invoking the Compiled Code from Compiled Code

As mentioned earlier, control is passed to the `CompiledCode` field of the `methodblock`. The C-style description of the call is as follows:

```c
mb->CompiledCode(obj, arg0, arg1, arg2, ...)
```

For SPARC and Linux/x86 differ in the function call convention, we show their argument-passing mechanisms separately.

#### For SPARC

The arguments are passed via SPARC function call convention, i.e., the first 6 arguments are passed in the registers %o0-%o5. Note that, for Java native code, %o0 is reserved for the object or the class pointer of the call, so the register usage actually is shifted by one. As an example for the following Java program:

```java
obj.method(arg0, arg1, arg2, arg3, arg4, arg5)
```

%o0 will have obj, %o1 will be assigned arg0, ..., %o5 will be assigned arg4, whereas arg5 is allocated onto the stack as caller-save register.

Similarly, the method return values also follow the SPARC function call convention, i.e., for integers, the value is returned in %i0, for longs (64-bits) %0 and %1, float in %f0, and doubles are returned in %f0 and %f1.

What differs from C function calls is that, we must always set the `methodblock` of the method to be called into the %g3 register. This value is necessary for virtual function calls (OpenJIT_invokevirtual, OpenJIT_invokevirtualobject_quick), as well as for exception handling, upon which the value must be saved in the native stack.
For Linux/x86

The arguments are passed via UNIX/x86 function call convention, all arguments are passed through native stack. The return value of the method are treated just as same as C function call (i.e. return values are returned using registers).

%eax will have obj, int, or lower-half of long value, %edx will have higher-half of long value, and %f0 will have floating stack top of float or double value.

What differs from C function calls is that we must always set the methodblock of the method to be called onto native stack top, prior to the actual call. This value is necessary for virtual function calls (OpenJIT_invokevirtual, OpenJIT_invokevirtualobject_quick), as well as for exception handling, upon which the value must be saved in the native stack.

A.2.5 Invoking the Compiler from the Compiled Code

The methods to be compiled by the JIT compiler already have the value of the CompiledCode field in the methodblock changed to dispatchJITCompiler on class loading time. When this method is invoked for the first time from within the compiled code, the dispatchJITCompiler is invoked. This function, if written in C, would have the following interface:

```c
void dispatchJITCompiler(? arg0, ? arg1, ? arg2, ...)
```

The types of the arguments and the return value depends on the method, and is indeterminate at compile time. In practice, we use assembly for efficiency to code in the following way:

```assembly
dispatchJITCompiler:
    save %sp,-112,%sp
    call compileMethod,0 ! compileMethod(mb)
    mov %g3, %o0
    ld [%g3+.off_CompiledCode],%l1 ! jump to mb->CompiledCode
    jmp %l1
restore
```

Firstly, compileMethod is invoked. The compileMethod does the compilation and sets the CompiledCode field. Then, this field is re-invoked with the same arguments, effectively executing the compiled native code.
A.2.6 Invoking the JVM from the Compiled Code

When the CompiledCode field of the methodblock structure is set to be dispatchJVM, then the following function is called to transfer the control to the interpreter. This is used when compilation fails, or the compilation is restricted due to the compiler option.

```c
void dispatchJVM(? arg0, ? arg1, ? arg2, ...)
```

JVM facilitates the following function to call a Java method from native code in general:

```c
long do_execute_java_method(ExecEnv *ee,  
    void *obj,  
    char *method_name, 
    char *signature, 
    struct methodblock *mb, 
    bool_t isStaticCall, 
    ...);
```

DispatchJVM interfaces the calling convention of the compiled native methods and this function:

1. Extract the methodblock structure from %g3.
2. Save the argument registers onto the stack.
3. Set the methodblock structure of the caller into the dummy Java frame. This is required for reflection and exception handling.
5. Set the return values into registers according to the return type.

A.2.7 Invoking the Native Method from the Compiled Code

When the method is originally native to begin with, the code field of the methodblock points to the native code to be executed. A call can be thus made in the following way in C:

```c
optop =  
    (*(stack_item *(*(stack_item*, ExecEnv*)))mb->code)(optop, ee)
```
The first argument is a pointer to the operand stack, and the second argument is a pointer to the execution environment ExecEnv. As a result, when we call a native method from the compiled native code, we must assign the register arguments into operand stack similarly to the call to the JVM. Also, the returned value on the operand stack must be placed back in the register. For this purpose, we define 5 stub functions according to the return type:

\[
\begin{align*}
\text{void } &\text{dispatchNormNativeV}(\ldots) \\
\text{int } &\text{dispatchNormNativeI}(\ldots) \\
\text{int64_t } &\text{dispatchNormNativeJ}(\ldots) \\
\text{float } &\text{dispatchNormNativeF}(\ldots) \\
\text{double } &\text{dispatchNormNativeD}(\ldots)
\end{align*}
\]

When the native method is a synchronized method, it further requires the monitor lock/unlock operations. For efficiency, we further define a set of separate stub functions to cover this case:

\[
\begin{align*}
\text{void } &\text{dispatchSynchNativeV}(\ldots) \\
\text{int } &\text{dispatchSynchNativeI}(\ldots) \\
\text{int64_t } &\text{dispatchSynchNativeJ}(\ldots) \\
\text{float } &\text{dispatchSynchNativeF}(\ldots) \\
\text{double } &\text{dispatchSynchNativeD}(\ldots)
\end{align*}
\]

These functions perform the followings:

1. Obtain the value of the \texttt{methodblock} from \%g3 to obtain the starting address of the native method.
2. Save the register arguments onto the native stack (Not the Java operand stack). These will become the second argument of the native call.
3. Obtain the value of \texttt{ExecEnv}.
4. Obtain the value of the \texttt{methodblock} of the caller, and set to the dummy JVM frame. Again, this is required for reflection and exception handling.
5. For synchronized methods, call \texttt{MonitorEnter}.
6. Call the native method.
7. For synchronized methods, call \texttt{MonitorExit}. 

APPENDIX A. OPENJIT RUNTIME SPECIFICATION

8. Check whether an exception has occurred. If so, call handle_exception, which actually controls the transfer entirely and never returns.

9. Obtain the return value of mb->code from the operand stack and place it appropriately into the register according to the return type.

NOTE: Steps 2–8 are defined as a macro DISPATCH_NATIVE.

A.3 Self-Modifying Code

Java bytecodes refers to classes, instance variables, and methods via symbol names. Symbols are stored in a structure called constant pool. Thus, on bytecodes execution, one must search the constant pool with the given symbols as a key, and obtain the actual address. This search is quite costly, as one must lock the constant pool region for multithreaded execution. Moreover, constant pool references occur quite frequently, and the cost of the search could dominate the overall execution time.

The JVM implementation solves this problem by modifying the bytecode on the fly. That is to say, the bytecode for constant pool access is modified to an equivalent so-called quick bytecode, which refers to the absolute address after the name has been resolved. For example, the bytecode instruction:

\begin{verbatim}
getfield #22 <Field Obj var>
\end{verbatim}

pushes the object variable value onto the operand stack. When this instruction is first executed, the constant pool is always searched for the constant pool index #22. As a result, when we find that this variable can be accessed at a 4-byte offset from the object header, then the JVM modifies the code at run-time to the following quick version:

\begin{verbatim}
getfield_quick 4
\end{verbatim}

then subsequently re-executes the instruction. From that point on, the quick instruction is always used, since the constant pool does not change over the execution of the program, effectively eliminating the lookup cost.

However, for native code, it is more difficult to eliminate this cost of lookup by naive application of a similar technique. A simple method would be to search all the possible symbol values in the constant pool and resolve them at once at compile time. However, constant pool resolution is not mere simple symbol resolution, but rather incurs other processing such as class
loading and initialization; thus, this strategy could change the semantics of the program by changing the initialization order of classes.

The viable option is to change the native instruction code in the same manner as the interpreter. However, it is much more difficult to do for native code, which involves several native instructions per each bytecode. Since the length of the instruction sequence cannot change, this could involve insertion of several \texttt{NOP} instructions. Moreover, the change must be atomic, requiring some form of mutual exclusion. Moreover, the change must propagate across code caches on different processors in a multiprocessor environment.

The basic solution is as follows. For SPARC, we place the \texttt{CALL} instruction to the constant pool resolution routine preceding a delay slot which contains the \texttt{MOV} instruction to set the index number of the symbol onto a register:

\begin{verbatim}
call resolve
mov #22,%o0
\end{verbatim}

When this sequence is executed, the \texttt{resolve()} routine is called. There, after the constant pool is searched and the address corresponding to the index is found, the two instructions are rewritten so that now the instruction sequence places the the address (or the offset) value into a register:

\begin{verbatim}
sethi %hi(offset), %o0
or %o0, %lo(offset),%o0
\end{verbatim}

Then, the \texttt{resolve()} routine returns with the resolved address in the \%o0 register. The instruction immediately following the two rewritten instructions merely accesses the memory using \%o0. For SPARC, we could further optimize this as small offsets can be encoded into one instruction, and the load instruction contains a displacement field.

\textbf{A.3.1 Two Instructions Modification}

Although the basic idea was given, in practice for SPARC the \texttt{MOV} instruction only accepts signed 13 bits as index values. In JVM, the index value can be as large as 16 bits; so, we have employed the \texttt{SETHI} instruction instead of the \texttt{MOV} instruction, allowing the usage of 22 bits. One caveat is that the encoded index value is the value obtained by left shifting the bits by 10 bits.
call resolver   -> sethi %hi(offset),%o0
sethi (index<<10),%o0  -> or %o0,%lo(offset),%o0

The following functions actually searches the constant pool and resolves the offsets:

- OpenJIT_resolveField
- OpenJIT_resolveStaticField
- OpenJIT_resolveString

**OpenJIT_resolveField**

```c
int OpenJIT_resolveField(int index)
```

The JVM `getfield` and `putfield` instructions are translated to call this function, which performs the followings:

1. Check for self modification (`CHECK_SELF_MODIFYING`).
2. Right shift the index by 10 bits.
3. Extract the `methodblock` of the caller. We obtain the address of the constant pool from this `methodblock` structure. Also, we set the address value into the JVM dummy frame in case exception happens.
4. Search and resolve the constant in the pool (`RESOLVE_CLASS_CONST`).
5. Check whether we have access rights to the field. If the field is static then generate an exception.
6. Obtain the offset, and modify the instruction as stated above (`PATCH_SET_O0`).
7. Return the offset value.

**OpenJIT_resolveStaticField**

```c
int OpenJIT_resolveStaticField(int index)
```

The JVM `getstatic` and `putstatic` bytecodes are translated to call this function.
1. Check for self modification (CHECK_SELF_MODIFYING).

2. Right shift the index by 10 bits.

3. Extract the methodblock of the caller. We obtain the address of the constant pool from this methodblock structure. Also, we set the address value into the JVM dummy frame in case exception happens.

4. Search and resolve the constant in the pool (RESOLVE_CLASS_CONST).

5. Check whether we have access rights to the field. If the field is NOT static then generate an exception.

6. If the field type is long or double (64-bits);
   Obtain the offset, and modify the instruction as stated above (PATCH_SET_O0).
   Return the u.static_address of the fieldblock.

7. If the field type is not 64-bits;
   Obtain the offset, and modify the instruction as stated above (PATCH_SET_O0).

OpenJIT_resolveString

```c
int OpenJIT_resolveString(int index)
```

For JVM bytecodes ldc and ldc_w, if the constant pool type is CONSTANT_String, then the bytecodes are translated to call the function.

1. Check for self modification (CHECK_SELF_MODIFYING).

2. Right shift the index by 10 bits.

3. Extract the methodblock of the caller. We obtain the address of the constant pool from this methodblock structure. Also, we set the address value into the JVM dummy frame in case exception happens.

4. Search and resolve the constant in the pool (RESOLVE_CLASS_CONST).

5. Obtain the offset, and modify the instruction as stated above (PATCH_SET_O0).

6. Return the address value.
A.3.2 One Instruction Modification

We modify only the CALL instruction. This is only employed when a new class is loaded on method call. This series of instructions is a combination of setting the pointer to the method block into the %g3 register, and calling the invoker function. Here, only the CALL instruction is modified.

<table>
<thead>
<tr>
<th>Original Instruction</th>
<th>Modified Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>sethi %hi(mb),%g3</td>
<td>sethi %hi(mb),%g3</td>
</tr>
<tr>
<td>call old_invoker</td>
<td>call new_invoker</td>
</tr>
<tr>
<td>or %g3,%lo(mb),%g3</td>
<td>or %g3,%lo(mb),%g3</td>
</tr>
</tbody>
</table>

The functions below are subject to such one-instruction modification:

- OpenJIT_invokeinterface
- OpenJIT_invokeinterface_quick
- OpenJIT_invokespecial
- OpenJIT_invokesuper_quick
- OpenJIT_invokestatic
- OpenJIT_invokevirtual

OpenJIT_invokeinterface

```
int OpenJIT_invokeinterface(....)
```

The JVM invokeinterface bytecode is translated to call this function, which performs the followings:

1. Check for self modification (CHECK_SELF_MODIFYING).
2. Extract the methodblock of the caller.
3. Search and resolve the constant in the pool (RESOLVE_CLASS_CONST).
4. Patch the instruction to call OpenJIT_invokeinterface_quick.
5. Jump to OpenJIT_invokeinterface_quick.
OpenJIT_invokeinterface_quick

The JVM invokeinterface_quick instruction is translated to invoke this function. Also, as indicated in Section A.3.1, it is invoked subsequently to the invocation of OpenJIT_invokeinterface. The %g3 register must contain the predicted value of the method table. The procedure is almost same as when JVM processes the invokeinterface_quick bytecode, but differs in the following points:

- The predicated value is obtained by shifting %g3 right by 24 bits.
- When the method is found, we modify the instruction in order to set the predicted value.

We modify the instruction that sets the %g3 register, preceding the call instruction which called this function; the modification is such that the predicted value is shifted by 24 bits, and set to the upper 8-bits of the %g3 register. Since the predicated value could be old and stale, we do not lock the instruction upon modification.

OpenJIT_invokespecial

```c
void OpenJIT_invokespecial(...) {

    The JVM invokespecial, invokenonvirtual_quick bytecodes are translated to call this function. The function is effectively used when the method does not change for the given call site irrespective of the type of the object.

    1. Check for self modification (CHECK_SELF_MODIFYING).
    2. Extract the methodblock of the caller.
    3. Check the class and the methodblock structure, and find out whether we are invoking the methods in the ancestor classes (super).
    4. For super calls, modify the instruction to be a call to OpenJIT_invokesuper_quick, and jump to OpenJIT_invokesuper_quick().
    5. If it is not a super call, either compile the method to be called, or load it in the case it is a native method. (RESOLVE_NATIVE_OR_COMPILE).
    6. Modify the instruction to directly call mb->CompiledCode.
```
OpenJIT_invokesuper_quick

```java
void OpenJIT_invokesuper_quick(...)
```

The JVM invokesuper_quick bytecode is translated to call this function. Also, it might be called after a call to the OpenJIT_invokespecial function. It performs essentially the same procedure as the JVM for this instruction.

OpenJIT_invokestatic

```java
void OpenJIT_invokestatic(...)
```

The JVM invokestatic instruction (in case it does require constant pool resolution) and the invokestatic_quick instruction are translated to call this function, which performs the followings, allowing direct, fast calls to static methods:

1. Check for self modification (CHECK_SELF_MODIFYING).
2. Extract the methodblock of the caller.
3. Either compile the callee method or load the native method as specified in the classfile (RESOLVE_NATIVE_OR_COMPILE).
4. Modify the instruction to make a direct jump to mb->CompiledCode.
5. Make the jump to mb->CompiledCode.

OpenJIT_invokevirtual

```java
void OpenJIT_invokevirtual(...)
```

For invokevirtual bytecodes that does not require constant pool resolution, and the invokevirtual_quick instructions are translated to call this function. The procedure is similar to OpenJIT_invokestatic, but the call target of the self-modified code differs in the following way:

- When the method is private;
We rewrite the method in the same manner as the OpenJIT_invokestatic to make a jump to mb->CompiledCode. This allows direct jump to the target method.

- **java.lang.Object** methods;
  
  We rewrite the target of the call to OpenJIT_invokevirtualobject_quick.

- **Other cases**;
  
  We rewrite the target of the call to OpenJIT_invokevirtual_quick.

Subsequently, the case analysis becomes unnecessary, speeding up the virtual call.

### A.3.3 Three Instructions Modification

We modify the instruction sequence consisting of three instructions, which involves a method invocation with constant pool resolution. The general sequence is as follows:

```
call old_invoker        -> sethi %hi(mb),%g3
sethi %g3,index<<10     -> call new_invoker
illtrap                 -> or %g3,%lo(mb),%g3 /* delay slot */
```

The following functions are subject to 3 instruction modification:

- OpenJIT_invokespecial_resolve
- OpenJIT_invokekestatic_resolve
- OpenJIT_invokevirtual_resolve

#### OpenJIT_invokespecial_resolve

```
void OpenJIT_invokespecial_resolve(...)
```

The JVM invokespecial bytecode is translated to call this function. After constant pool resolution, we perform the same process as the OpenJIT_invokespecial function.
OpenJIT_invokestatic_resolve

```c
void OpenJIT_invokestatic_resolve(...)
```

The JVM `invokestatic` bytecode is translated to call this function. After constant pool resolution, we perform the same process as the `OpenJIT_invokestatic` function.

OpenJIT_invokevirtual_resolve

```c
void OpenJIT_invokevirtual_resolve(...)
```

The JVM `invokevirtual` bytecode is translated to call this function. After constant pool resolution, we perform the same process as the `OpenJIT_invokevirtual` function.

A.3.4 Three Instructions Modification (Modifying %o0)

We modify the instruction sequence consisting of three instructions, which involves an access to class object with constant pool resolution. The general sequence is as follows. For this kind of sequences, for the modified target of the call instruction, the first argument of the call is the resolved address of the class object.

```
call old_func     -> sethi %hi(mb),%o0
sethi %o0,index<<10 -> call new_func
illtrap        -> or %g3,%lo(mb),%o0 /* delay slot */
```

The following functions are subject to 3 instruction modification with %o0:

- OpenJIT_new
- OpenJIT_anewarray
- OpenJIT_multianewarray
- OpenJIT_checkcast
- OpenJIT_instanceof
OpenJIT_new

HObject *OpenJIT_new(int index)

The JVM new bytecode is translated to call this function. It performs the following steps:

1. Check for self modification (CHECK_SELF_MODIFYING).
2. Shift the index right by 10 bits.
3. Constant pool resolution (RESOLVE_CLASS_CONST).
4. Check the access rights to the class, and for illegal access generate an exception.
5. Self-modify the call so that the new target is OpenJIT_new_quick (PATCH_SET_O0_and_CALL).
6. Physically allocate an object from memory (same as OpenJIT_new_quick).
7. Return the pointer to the new object.

OpenJIT_anewarray

HArrayOfObject *OpenJIT_anewarray(int index, int size)

The JVM anewarray bytecode is translated to call this function. It performs steps similar to OpenJIT_new, self-modifies the target to OpenJIT_anewarray_quick, and jumps to the OpenJIT_anewarray_quick.

OpenJIT_multianewarray

HArrayOfObject *OpenJIT_multianewarray(int index, int dimensions, stack_item *optop)

The JVM multianewarray bytecode is translated to call this function. It performs steps similar to OpenJIT_new, self-modifies the target to OpenJIT_multianewarray_quick, and jumps to the OpenJIT_multianewarray_quick.
OpenJIT_checkcast

```c
void OpenJIT_checkcast(int index, JHandle *h)
```

The JVM `checkcast` bytecode is translated to call this function. It performs steps similar to `OpenJIT_new`, self-modifies the target to `OpenJIT_checkcast_quick`, and jumps to the `OpenJIT_checkcast_quick`.

OpenJIT_instanceof

```c
bool_t OpenJIT_instanceof(int index, JHandle *h)
```

The JVM `instanceof` bytecode is translated to call this function. It performs steps similar to `OpenJIT_new`, self-modifies the target to `OpenJIT_instanceof_quick`, and jumps to the `OpenJIT_instanceof_quick`.

A.3.5 Self-Modifying Code in OpenJIT

Since JVM is inherently multithreaded, caution is required for atomic updating of successive sequence of multiple instructions. If the self-modification is not atomic, other threads might try to execute the half-cooked instruction sequence, resulting in a critical error.

The current JVM supports two types of thread system. One is the green thread, and the other is the native thread. The green thread only works for uniprocessor machines, and the context switching occurs only at fixed, safe locations, and thus such problems do not occur. For native threads, however multiple threads might be executing on different processors, resulting in partially rewritten instruction sequences to be executed. Thus, it is extremely important to guarantee the atomicity of self-modification in an efficient manner. OpenJIT implements such an atomic update in the following way:

Macro `CHECK_SELF_MODIFYING()`

This macro checks whether the call instruction to the function which uses the macro has been modified or not. If it has been modified, then the control returns to the modified instruction of the call site, which is re-executed.
Macro PATCH_CODE(CODE, OFFSET)

This macro modifies the instruction whose offset is OFFSET from the call instruction which called the function which uses this macro. Subsequently, the instruction cache is flushed. For example, PATCH_CODE(code, 4) modifies the delay slot of the call site which called the function.

Modifying Multiple Instructions Atomically

Multiple instructions are modified atomically in the following way. We assume that the first instruction of the sequence of instructions to be modified is a CALL instruction, followed by NOP instructions. The function which had been called by the CALL instruction modifies the instruction sequence.

1. First, the CALL instruction is modified to unconditional branch instruction to effectively spin lock on the instruction. This modification is atomic, and any thread which executes the jump instruction goes into infinite spin.

2. Change the NOP instructions to desired instruction sequence.

3. Change the unconditional branch instruction to the desired instruction. The threads that had been spinning on the branch instruction will resume with the execution of the new instruction sequence.

We illustrate this scheme in Figure A.2.

One problem with this scheme is when multiple threads execute the CALL A instruction. However, since both threads will be modifying the
instruction sequence ($\text{INST1}$, ..., $\text{INST4}$) identically, this will not cause a problem.

### A.4 Exception Handling

For efficient execution, OpenJIT backend does not generally check for exceptions except for a few instances where explicit runtime checks are required. Instead, exceptions are checked and processed using the UNIX signaling mechanism.

Figure A.3 indicates how the compiled native code executes. We must check for exception occurrence when the transfer of control occurs between the compiled native code and other native code such as the JVM interpreter and runtime routines, and native methods. For example, in Figure A.3, we must check for exception for each point in the control flow marked by a star. On the other hand, for exceptions occurring with the compiled native code, we generally employ the UNIX signals, and do not explicit check for exception occurrence.

### A.4.1 Checking Exceptions Using UNIX Signals

By setting the Java Native Code API, the following function is called when a Unix signal occurs:

```java
static void OpenJIT_SignalHandler(int sig, siginfo_t *info, ucontext_t *uc)
```
Below are the possible exceptions that might occur in runtime. Other signals are not JVM exceptions, but rather a compiler or a JVM bug.

**SIGFPE** zero division

**SIGSEGV** null pointer, stack overflow

**SIGILL** array index out of bounds

Within the `OpenJIT_SignalHandler` function, in order to check that the signal was indeed generated by a Java exception and not a compiler or a JVM bug, we check the instruction that caused the exception, and its operand address. For each type of exception, we perform the check in the following way, and by calling the `setcontext()` system call, we setup the calling frame so that the instruction causing the exception behaves as if it had called the exception generation function.

### Zero Division

The signal SIGFPE is raised, and the exception code `info->si_code` is either `FPE_INTOVF` or `FPE_INTDIV`. If so, signal handler sets up the context so that it seems as if the following function had been called from the instruction that caused the exception.

```c
void catchZeroDivide(unsigned char *pc)
```

### Null Pointer

The signal SIGSEGV is raised, and the exception code `info->si_code` is `SEGV_MAPERR`. In addition, the base register of the instruction that caused the exception is 0. Here is the exception generation function:

```c
void catchNullPointer(unsigned char *pc)
```

### Stack Overflow

The signal SIGSEGV is raised, and the exception code `info->si_code` is `SEGV_MAPERR`. In addition, the instruction that caused the exception is `ld [%sp + constant]`. Here is the exception generation function:
Array Index Out of Bounds

The signal SIGILL is raised, and the instruction that caused the exception is a trap instruction, and the trap code is ST\_RANGE\_CHECK. Here is the exception generation function:

```c
void catchArrayIndexOutOfBounds(unsigned char *pc, int index)
```

This function is slightly different, in that the index of the array must be given as the second parameter. For this reason, before we perform `setcontext`, we must check the value of the register which was used as an operand to calculate the out-of-bounds condition.

### A.4.2 Finding an Exception Frame

FIND\_EXCEPTION\_FRAME(pc, ee) is a macro used by the exception generation functions described above in order to identify the method that caused the exception. It performs the following steps:

1. Flush the register window.
2. Trace the native stack.
   - Walk the stack until the frame for the compiled native code is found.
3. Set the pointer to the methodblock structure into the dummy JVM frame.
4. Setup for the `fillInStackTrace` (described later in Section A.4.4).

### A.4.3 Jumping into an Exception Handler

```c
bool_t handle_exception (ExecEnv *execEnv)
```

This function traces the compiled native code stack, and finds the corresponding exception handler, and jumps to the handler. As is with C `longjump()`, it makes a jump leapfrogging the nested function calls. Because SPARC has register windows, they must be restored appropriately during leapfrogging. Figure A.4 shows the steps.
while(1) {
    // delete the stack frame of the runtime routine
    while (%i7(address of the caller) is within the runtime
           routine) {
        restore // recover the register window
    }
    if (%i7(return address) is not a compiled native code) {
        // Return to the JVM interpreter loop
        return FALSE;
    }

    // Set the lastpc.
    // Needed when returning to the interpreter loop?
    ee->current_frame->lastpc = %i7

    Extract the pointer to the methodblock structure from %fp,
    and set it to the variable mb.

    // Find the exception handler for the caught exception
    // within mb
    new_pc = JITProcedureFindThrowTag(ee, mb, 
                                    ee->exception.exc, %i7)
    if (new_pc != 0) {
        // An exception handler is found!
        exceptionClear(ee) // Clear the exception flag

        // The exception handler for the compiled native code
        // assume that the pointer to the object that caused the
        // exception is in %i7
        %i7 = ee->exception.exc
        restore // restore the register window
        jump new_pc // Jump to the exception handler
    }

    // Exception handler is not found
    if (mb is a synchronized method) {
        // unlock the monitor lock
        // The monitor object is stored in %fp[-1]
        monitorExit(%fp[-1])
    }
    restore
}

Figure A.4: Jumping into an exception handler
A.4.4 Filling in Stack Trace

JDK calls the `SignalError` function when an exception occurs. This function in turns calls `fillInStackTrace()`. Also, `java.lang.Throwable` class has a method `fillInStackTrace`, allowing the user program to obtain the status of the current Java method, and the trace of the stack frame.

The code generated by the OpenJIT compiler does not generate a Java frame when compiled native code is called from another native code. As a result, JVM cannot trace the stack frame. To solve this problem, the JDK prepares the following API:

```c
JavaFrame *JITCompiledFramePrev(JavaFrame *frame, 
                JavaFrame *buf)
```

Other than `fillInStackTrace`, this function is used to obtain the trace of the stack frame. JVM basically uses the following algorithm to walk the stack to obtain the trace:

```c
{ 
    JavaFrame *frame, buf;
    frame = ExecEnv->current_frame;
    while(frame) {
        if (frame->current_method->fb.access & ACC_MACHINE_COMPILED) 
            frame = CompiledFramePrev(frame, &buf);
        else 
            frame = frame->prev;
    } 
}
```

Thus, before `JITCompiledFramePrev` is called, `ExecEnv` (the execution environment structure) `current_frame` must have the Java frame of the compiled native code. For this purpose, when there is a possibility that an exception may occur upon calling a JVM function from the OpenJIT runtime routine, we must also set the JVM frame in the `ExecEnv->current_frame`.

For OpenJIT, we judged that it is too expensive to generate a JVM frame each time this happens. Instead, we generate a dummy JVM frame only when the control flow transfers from the compiled native code into the internals of the JVM, and set it to `ExecEnv->current_frame`. When the OpenJIT runtime routine calls a JVM function, we merely set the `current_method` of the dummy frame.
A.5 Other Runtime Functions

We show the other OpenJIT runtime functions that are called from the compiled native code that the OpenJIT compiler generates (Table A.2). The compiled native code may also call a C library function or a JVM function. The table below indicates where the called functions are being defined.
<table>
<thead>
<tr>
<th>JVM Instruction</th>
<th>Runtime Function</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>anewarray_quick</td>
<td>HArrayOfObject *OpenJIT_anewarray_quick(ClassClass *array_cb, int size)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>athrow</td>
<td>void OpenJIT_athrow(HJava_lang_Object *obj)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>checkcast_quick</td>
<td>void OpenJIT_checkcast_quick(ClassClass *cb, JHandle *h)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>d2l</td>
<td>int64_t __dtoll(double d)</td>
<td>C</td>
</tr>
<tr>
<td>dcmpg</td>
<td>int OpenJIT_dcmpg(stack_item *p)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>dcmpl</td>
<td>int OpenJIT_dcmpl(stack_item *p)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>drem</td>
<td>double OpenJIT_drem(stack_item *p)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>f2l</td>
<td>int64_t __ftoll(float f)</td>
<td>C</td>
</tr>
<tr>
<td>fcmpg</td>
<td>bool_t OpenJIT_fcmpg(float *p)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>fcmpl</td>
<td>bool_t OpenJIT_fcmpl(float *p)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>frem</td>
<td>float OpenJIT_frem(float *args)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>instanceof</td>
<td>bool_t OpenJIT_instanceof(int index, JHandle *h)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>l2d</td>
<td>double OpenJIT_l2d(signed hi, unsigned lo)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>l2f</td>
<td>float OpenJIT_l2f(signed hi, unsigned lo)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>lcmp</td>
<td>bool_t OpenJIT_lcmp(long long x, long long y)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>ldiv</td>
<td>int64_t __div64(int64_t x, int64_t y)</td>
<td>C</td>
</tr>
<tr>
<td>lmul</td>
<td>int64_t __mul64(int64_t x, int64_t y)</td>
<td>C</td>
</tr>
<tr>
<td>lrem</td>
<td>int64_t __rem64(int64_t x, int64_t y)</td>
<td>C</td>
</tr>
<tr>
<td>lshl</td>
<td>uint64_t longOpenJIT_lshl(signed hi, unsigned lo, unsigned b)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>lshr</td>
<td>uint64_t longOpenJIT_lshr(signed hi, unsigned lo, unsigned b)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>monitorEnter</td>
<td>void monitorEnter(unsigned int key)</td>
<td>JDK</td>
</tr>
<tr>
<td>monitorExit</td>
<td>void monitorExit(unsigned int key)</td>
<td>JDK</td>
</tr>
<tr>
<td>multianewarray_quick</td>
<td>HObject *OpenJIT_multianewarray_quick(ClassClass *array_cb, int dimensions, stack_item *optop)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>new_quick</td>
<td>HObject *OpenJIT_new_quick(ClassClass *cb)</td>
<td>OpenJIT</td>
</tr>
<tr>
<td>newarray</td>
<td>JHandle *OpenJIT_newarray(int type, int size)</td>
<td>OpenJIT</td>
</tr>
</tbody>
</table>

Table A.2: Other runtime functions
A.6 Summary

We covered the runtime structure of the OpenJIT backend system. For the details of how the JVM instructions are translated, and runtime functions are called, the readers are referred to the files in org/OpenJIT/Sparc.java and org/OpenJIT/X86.java. The layout of the stack frame of the compiled native code is described in a companion document OpenJIT Backend Compiler Internal Specification.1157.