



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Ph.D. DISSERTATION

# Java Virtual Machine Optimizations for Java and Dynamic Languages

자바와 동적언어를 위한 자바 가상 머신 최적화 기법

BY

YANG BYUNG-SUN

FEBRUARY 2017

DEPARTMENT OF ELECTRICAL ENGINEERING AND  
COMPUTER SCIENCE  
COLLEGE OF ENGINEERING  
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

# Java Virtual Machine Optimizations for Java and Dynamic Languages

자바와 동적언어를 위한 자바 가상 머신 최적화 기법

BY

YANG BYUNG-SUN

FEBRUARY 2017

DEPARTMENT OF ELECTRICAL ENGINEERING AND  
COMPUTER SCIENCE  
COLLEGE OF ENGINEERING  
SEOUL NATIONAL UNIVERSITY

# Java Virtual Machine Optimizations for Java and Dynamic Languages

자바와 동적언어를 위한 자바 가상 머신 최적화 기법

지도교수 문수목  
이 논문을 공학박사 학위논문으로 제출함

2017년 2월

서울대학교 대학원

전기 컴퓨터 공학부

양병선

양병선의 공학박사 학위 논문을 인준함

2017년 2월

위원장: \_\_\_\_\_  
부위원장: \_\_\_\_\_  
위원: \_\_\_\_\_  
위원: \_\_\_\_\_  
위원: \_\_\_\_\_

# Abstract

Java virtual machine (JVM) has been introduced as the machine-independent runtime environment to run a Java program. As a 32-bit stack machine, JVM can execute bytecode instructions generated through compilation of a Java program on any machine if the JVM runtime was correctly ported on it. The machine-independence of JVM brought about the huge success of both the Java programming language and the Java virtual machine itself on various systems encompassing from cloud servers to embedded systems including handsets and smart cards.

Since a bytecode instruction should be interpreted by the JVM runtime for execution on top of a specific underlying system, a Java program runs innately slower due to the interpretation overhead than a C/C++ program that is compiled directly for the system. Java just-in-time (JIT) compilers, the *de facto* performance add-on modules, are employed to improve the performance of a Java virtual machine (JVM) by translating Java bytecode into native machine code on demand.

One important problem in Java JIT compilation is how to map stack entries and local variables of the JVM runtime to physical registers efficiently and quickly, since register-based computations are much faster than memory-based ones, while JIT compilation overhead is part of the whole running time. This paper introduces LaTTe, an open-source Java JIT compiler that performs fast generation of efficiently register-mapped RISC code. LaTTe first maps “all” local variables and stack entries into pseudo registers, followed by real register allocation which also coalesces copies corresponding to pushes and pops between local variables and stack entries aggressively. In addition to the efficient register allocation, LaTTe is equipped with various traditional and object-oriented optimizations such as CSE, dynamic method inlining, and specialization. We also devised new mechanisms for Java exception handling and monitor handling in LaTTe, named *on-demand exception handling and lightweight monitor*,

respectively, to boost up the JVM performance more.

Our experimental results indicate that LaTTe’s sophisticated register mapping and allocation really pay off, achieving twice the performance of a naive JIT compiler that maps all local variables and stack entries to memory. It is also shown that LaTTe makes a reasonable trade-off between quality and speed of register mapping and allocation for the bytecode. We expect these results will also be beneficial to parallel and distributed Java computing 1) by enhancing single-thread Java performance and 2) by significantly reducing the number of memory accesses which the rest of the system must properly order to maintain coherence and keep threads synchronized.

Furthermore, Java virtual machine (JVM) has recently evolved into a general-purpose language runtime environment to execute popular programming languages such as JavaScript, Ruby, Python, or Scala. These languages have complex non-Java features including dynamic typing and first-class function, so additional language runtimes (engines) are provided on top of the JVM to support them with bytecode extensions. Although there are high-performance JVMs with powerful just-in-time (JIT) compilers, running these languages efficiently on the JVM is still a challenge.

This paper introduces a simple and novel technique for the JVM JIT compiler called *exceptionization* to improve the performance of JVM-based language runtimes. We observed that the JVM executing some non-Java languages encounters at least 2 times more branch bytecodes than Java, most of which are *highly biased* to take only one target. Exceptionization treats such a highly-biased branch as some implicit exception-throwing instruction. This allows the JVM JIT compiler to prune the infrequent target of the branch from the frequent control flow, thus compiling the frequent control flow more aggressively with better optimization. If a pruned path was taken, it would run like a Java exception handler, i.e., a catch block. We also devised de-exceptionization, a mechanism to cope with the case when a pruned path is actually executed more often than expected.

Since exceptionization is a generic JVM optimization, independent of any specific language runtime, it would be generally applicable to any language runtime on the JVM. Our experimental result shows that exceptionization accelerates the performance of several non-Java languages. The JavaScript-on-JVM runs faster by as much as 60%, and by 6% on average, when running the Octane benchmark suite on Oracle's latest Nashorn JavaScript engine and HotSpot 1.9 JVM. Additionally, the Ruby-on-JVM experiences the performance improvement by as much as 60% and by 6% on average, while the Python-on-JVM by as much as 6%. We found that exceptionization is most effectively applicable to the branch bytecode of the language runtime itself, rather than the bytecode corresponding to the application code or the bytecode of the Java class libraries. This implies that the performance benefit of exceptionization comes from better JIT compilation of the non-Java language runtime.

**keywords:** Java, Javascript, Java Virtual Machine, Just-in-Time Compiler, Dynamic Compiler, Dynamic Languages, Non-Java Languages, Optimization, Compiler

**student number:** 99420-841

# Contents

<b>Abstract</b>	<b>i</b>
<b>Contents</b>	<b>iv</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Java and Java Virtual Machine . . . . .	1
1.2 Java Virtual Machine and JVM Languages . . . . .	3
1.3 Outline of the Dissertation . . . . .	5
<b>2 Java Virtual Machine Optimization for Java</b>	<b>6</b>
2.1 Java Virtual Machine and SPARC . . . . .	8
2.2 Bytecode Translation with Aggressive Register Mapping . . . . .	9
2.2.1 Issues in Register Mapping for Bytecodes . . . . .	9
2.2.2 Translation of Bytecode into Pseudocode . . . . .	10
2.3 Fast Register Allocation . . . . .	16
2.3.1 Tree Regions . . . . .	16
2.3.2 Backward Sweep and Forward Sweep . . . . .	18
2.3.3 Reconciling <code>h_map</code> at Region Join Points . . . . .	20
2.3.4 Register Spill . . . . .	22

2.3.5	A Register Allocation Example . . . . .	23
2.4	Comparison with Previous JIT Compilation Techniques . . . . .	25
2.5	More Optimizations in the LaTTe JVM JIT Compiler . . . . .	30
2.5.1	Optimizations in the JIT Compiler . . . . .	30
2.5.2	On-demand Exception Handler Translation . . . . .	30
2.5.3	Lightweight Monitor for Synchronization . . . . .	31
2.5.4	Memory Management . . . . .	31
2.6	Experimental Results . . . . .	32
2.6.1	Experimental Environment . . . . .	32
2.6.2	Evaluation of LaTTe’s JIT Compilation Techniques . . . . .	32
2.6.3	Speed and Quality of LaTTe’s Register Mapping and Allocation	34
<b>3</b>	<b>Java Virtual Machine Optimization for Dynamic Languages</b>	<b>39</b>
3.1	Non-Java Languages on JVM . . . . .	41
3.1.1	JVM Extensions for Multi-Languages . . . . .	41
3.1.2	Nashorn JavaScript Engine on HotSpot JVM . . . . .	42
3.1.3	Other Non-Java Languages on HotSpot JVM . . . . .	45
3.2	Branch Bytecode Behaviours Of Non-Java Languages . . . . .	46
3.2.1	Branch Bytecode Statistics . . . . .	46
3.2.2	Behavioral Characteristics of Branch Bytecodes . . . . .	49
3.3	Exceptionization . . . . .	53
3.3.1	Idea of <i>Exceptionization</i> . . . . .	53
3.3.2	Selection of <i>Exceptionization</i> Targets . . . . .	55
3.3.3	Branch Exception Handling . . . . .	57
3.3.4	De-exceptionization . . . . .	58
3.4	Experimental Results . . . . .	61
3.4.1	Performance Impact on Non-Java Languages . . . . .	62
3.4.2	Performance Impact on Java . . . . .	66
3.4.3	Performance Variation by <i>Exceptionization</i> Threshold . . . . .	67

3.4.4	Performance Impact from <i>De-exceptionization</i> . . . . .	68
3.4.5	Origin of Exceptionized Branches in JavaScript . . . . .	71
3.5	Related Work . . . . .	72
<b>4</b>	<b>Summary and Conclusion</b>	<b>76</b>
	<b>Abstract (In Korean)</b>	<b>84</b>
	<b>Acknowledgement</b>	<b>87</b>

# List of Tables

2.1	LaTTe JVM running time (seconds) with LaTTe JIT and Kaffe JIT. . .	33
2.2	Register mapping and allocation quality of the base LaTTe for top five frequent methods. . . . .	35
2.3	Register allocation quality of the base LaTTe for top five largest-locals methods. . . . .	37
3.1	C2 compilation and branch bytecodes on CLBG . . . . .	47
3.2	C2 compilation and branch bytecodes on Octane and SPECjvm2008 .	47
3.3	Raised branch exceptions during Octane execution . . . . .	69
3.4	<i>Exceptionizable</i> branches in JavaScript and Java encountered during compilation . . . . .	70

# List of Figures

1.1	The Java system. . . . .	2
1.2	JVM-hosted languages. . . . .	3
2.1	The object model of LaTTe . . . . .	12
2.2	A translation example from bytecode into pseudo SPARC code. (a) Java source, (b) bytecode, and (c) CFG of pseudo SPARC code. . . .	15
2.3	A CFG of basic blocks and tree regions. . . . .	16
2.4	The backward sweep algorithm. . . . .	18
2.5	The forward sweep algorithm. . . . .	19
2.6	Reconciling register allocation. . . . .	21
2.7	Reconciling register allocation. (a) Problem and (b) solution. . . . .	21
2.8	Register allocation process for the previous example. (a) Register allocation of Region A and (b) register allocation of Region B. . . . .	24
2.9	Translation by VTune and CACAO. (a) Bytecode, (b) VTune, (c) CACAO, and (d) LaTTe. . . . .	26
2.10	An inefficient translation example. . . . .	28
2.11	Translation overheads and translated bytes of LaTTe JIT and Kaffe JIT. . . . .	34
3.1	Nashorn on top of HotSpot in OpenJDK. . . . .	43
3.2	Bias distribution of branch bytecode execution frequency in C2-compiled methods (CLBG). . . . .	48

3.3	Bias distribution of branch bytecode execution frequency in C2-compiled methods (CLBG). . . . .	48
3.4	Bias distribution of branch bytecode execution frequency in C2-compiled methods. . . . .	48
3.5	C2 compilations over time (Octane). . . . .	51
3.6	C2 compilations over time (SPECjvm2008). . . . .	51
3.7	<i>Exceptionization</i> on a sample code segment. . . . .	54
3.8	Pseudo code for <i>exceptionization</i> . . . . .	56
3.9	Branch exception handling. . . . .	58
3.10	<i>De-exceptionization</i> with partial method compilation. . . . .	59
3.11	Pseudo code to trigger <i>de-exceptionization</i> . . . . .	60
3.12	Relative performance on Octane (higher is better). . . . .	63
3.13	Relative performance on Kraken (higher is better). . . . .	63
3.14	Relative performance on Ruby (higher is better). . . . .	65
3.15	Relative performance on Python (higher is better). . . . .	65
3.16	Relative performance on SPECjvm2008 (higher is better). . . . .	66
3.17	Relative performance on DaCapo (higher is better). . . . .	67
3.18	Relative performance with different thresholds on Octane. . . . .	67
3.19	Relative performance with different <i>de-exceptionizations</i> (higher is better). . . . .	68
3.20	Origin distribution of <i>exceptionized</i> branches in JavaScript and Java. .	71

# Chapter 1

## Introduction

*“Most people talk about Java the language, and this may sound odd coming from me, but I could hardly care less. At the core of the Java ecosystem is the JVM.”* (by James Gosling)

### 1.1 Java and Java Virtual Machine

For network computing on heterogeneous systems from cloud servers to embedded devices, the Java system, consisting of the Java language and the Java virtual machine, has become a prominent runtime environment. Many programs are written in Java to achieve various purposes and they are compiled into bytecodes by the Java compiler. The bytecode programs are executed by the Java virtual machine that is a software layer residing on top of multiple physical machines such as handsets, TVs, desktop computers, and servers, as illustrated in Figure 1.1.

The Java virtual machine provides a *machine independent* execution environment and materializes the key design policy of the Java system: “write once, use everywhere”. This machine-independence imposes some overheads on the system performance, because a bytecode instruction should be mapped or interpreted to machine instructions of underlying physical machines. The Java Just-In-Time (JIT) compiler is

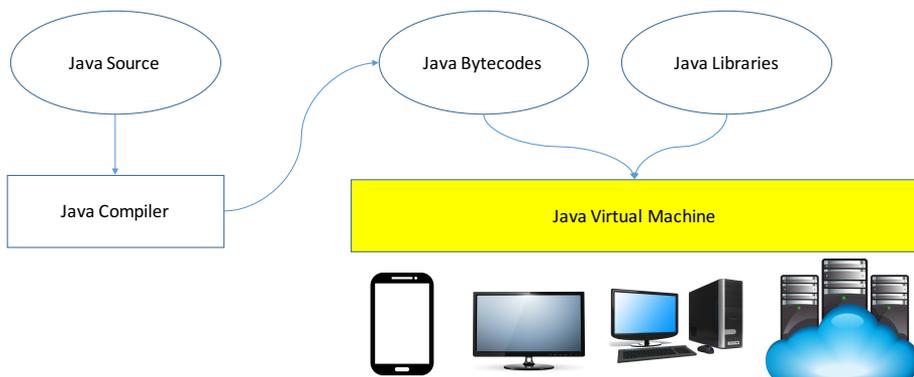


Figure 1.1: The Java system.

a component of the Java virtual machine which translates Java’s bytecode into native machine code on-the-fly prior to its execution so that the Java program runs as a real executable. Since the machine code is *cached* in the JVM, it can be used repetitively without re-translation. The idea of JIT compilation contrasts with static, off-line translation of bytecode and is more suitable for a dynamically loaded system mandated by the Java language specification.

A major issue of the Java JIT compilation is how to generate efficient code, especially by allocating stack entries and local variables of the JVM into physical registers effectively. One constraint is that since the JIT compilation time is part of the whole running time, the register allocation and accompanying optimizations should be done quickly, i.e., graph-coloring register allocation with copy coalescing would be the last technique to use, if an adaptive compilation framework can not be used. This requires a trade-off between quality and speed of register allocation, which poses a challenging engineering and research problem.

The first part of this paper introduces **LaTTe**, an open-source Java VM with a JIT compiler that performs fast and efficient register allocation and optimizations for RISC machines. Java bytecodes are translated into pseudo code with symbolic registers where many copies corresponding to pushes and pops between local variables and the operand stack are generated. Most of these copies are coalesced by fast allo-

cation of physical registers with a local lookahead. Additionally, the Java exception handling mechanism and the monitor synchronization mechanism are presented. They are carefully streamlined with the JIT compilation to provide better runtime performance through clever engineering designs.

The LaTTe JVM is operational on the SPARC platform and it achieves a performance better than or comparable to other product-level JVMs without using complex adaptive compilation.

## 1.2 Java Virtual Machine and JVM Languages

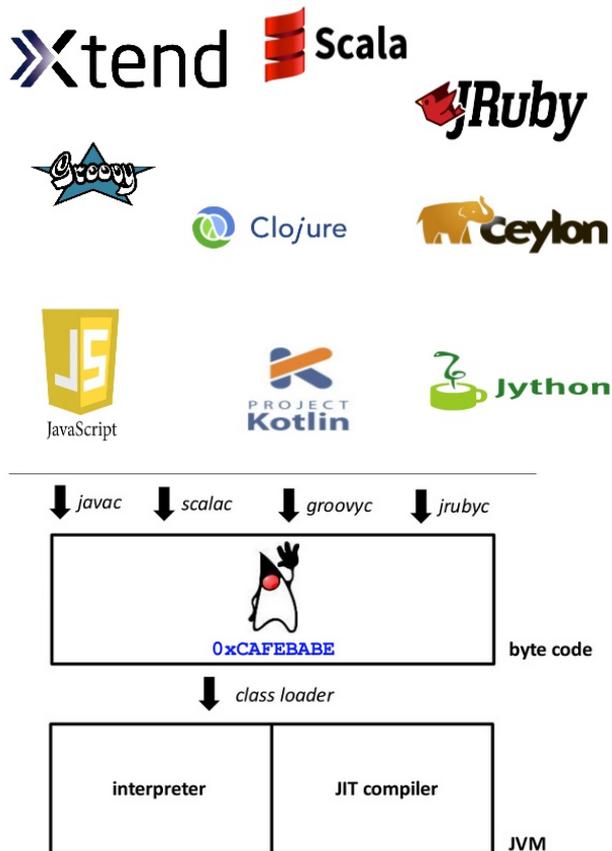


Figure 1.2: JVM-hosted languages.

For last two decades, the JVM has been so rigorously engineered that the current JVM performance is regarded as sufficiently mature in space of Java applications. And the proliferation of Java has given rise to the abundance of useful Java libraries that are building blocks to tackle various real-world problems. Consequently, the JVM started to become a runtime environment for dynamically-typed script languages including JavaScript, Python, Ruby and Groovy, and for functional languages including Clojure, Haskell and Scala. Figure 1.2 shows such JVM-hosted non-Java languages.

An existing language can be easily implemented on top of the JVM and the selected set of libraries with little performance degradation compared to a native language implementation. (*e.g.* JavaScript, Python, Ruby) And the language functionality can be easily extended with existing Java libraries, by exploiting the underlying JVM as the communication hub between the hosted language and the Java libraries. Furthermore, a new language tends to be quickly prototyped and tested with the JVM and Java libraries. (*e.g.* Groovy, Clojure, Scala)

Since algorithms and heuristics used for the JVM implementation are highly tailored for Java applications, recent researches found that the runtime performance for a JVM-hosted non-Java language is inferior to that of the native language implementation. So, novel optimization techniques are necessary to accelerate the non-Java language execution on top of the JVM.

The second part of this paper proposes a JVM optimization that is effective to the non-Java language execution, called *exceptionization*. Motivated by 1) observations on the non-Java language execution behavior and 2) on-demand exception handling in LaTTe, the JVM treats high-biased branches as implicit branch-exception throwing instructions during its JIT compilation. The frequently-executed path are dealt with and highly optimized, while the uncommon branch targets are excluded/pruned from the JIT compilation. If a pruned target is taken, then a trap-handling mechanism is invoked to take care of the taken path.

*Exceptionization* boosts up the start-of-art JVM performance significantly when

executing multiple non-Java applications on top of the JVM. As a JVM-level optimization, it provides the performance improvement to any JVM-hosted languages that use their language runtime on the JVM.

### **1.3 Outline of the Dissertation**

As a whole, the paper consists of two parts. Chapter 2 introduces the JVM optimizations for Java that are used in LaTTe. LaTTe's JIT compiler with efficient register allocation and other optimizations is explained and its experimental results are also presented. Chapter 3 presents *exceptionization*, a JVM optimization for non-Java dynamic languages. Since the optimization is integrated with the state-of-art HotSpot JVM and non-Java language runtimes including JavaScript, Python, and Ruby, the overall JVM runtime architecture is explained and the experimental results are given. The summary and conclusion comes in Chapter 4.

## Chapter 2

### Java Virtual Machine Optimization for Java

Recently, Java became a prominent programming language for parallel and distributed computing, due to its support for multithreading, networking, CORBA, and remote method invocation [1]. Unfortunately, parallel and distributed Java still has the same performance issue as sequential Java, related to executing Java bytecode. Indeed these performance issues are magnified by the additional synchronization and coherence overhead required in multi-processor environments. Efficient register allocation, as described in this paper, helps mitigate some of those problems by reducing the number of memory accesses which the rest of the system must properly order.

The Java Virtual Machine (JVM), a software layer to execute bytecode, while providing desirable features such as a “write-once, run anywhere” model for software developers, and security and portability for end-users, does not immediately lend itself to high performance. In order to circumvent the JVM overhead, a technique called Just-in-Time (JIT) compilation [2] is used to implement a JVM. Through JIT compilation, a bytecode method is translated into a native method on the fly, so as to remove the interpretation overhead.

The most important issue in Java JIT compilation is generating efficient code. A critical part of this is how to map and allocate stack entries and local variables into registers effectively. One constraint is that since the JIT compilation time is part of

the whole running time, this job should be done quickly. This requires a trade-off between quality of the generated code and speed of mapping and allocating registers for the bytecode, which poses a challenging research and engineering problem beyond a simple register allocation problem.

LaTTe is a freely available JVM and JIT compiler. LaTTe aggressively maps registers for the bytecode, and performs fast register allocation. LaTTe first translates bytecode into pseudocode by mapping all stack entries and local variables to symbolic registers. There will be many copies corresponding to pushes and pops between local variables and the stack in the pseudocode. LaTTe removes most of these copies via efficient register allocation with a local lookahead.

The contribution of this paper is twofold. First, since LaTTe is a working, high-performance JIT compiler whose source code is publicly available, this paper, together with the source code, can be helpful to readers interested in designing JIT compilers. Second, the present paper shows that LaTTe has made a reasonable trade-off between the quality and the speed of register mapping and allocation: the performance impact and the translation overhead of LaTTe's approach to register allocation are evaluated in detail in the paper. As already noted, these techniques contribute to improved performance in parallel environments by significantly reducing the number of memory accesses which the rest of the system must properly order. These techniques also contribute to parallel and distributed Java computing environments by improving the performance of individual threads.

The rest of Chapter 2 is organized as follows: Section 2.1 briefly reviews the Java VM and our target RISC machine, SPARC, focusing on calling conventions. Section 2.2 describes the register mapping and the translation of bytecode into pseudo SPARC code. Section 2.3 describes the real register allocation technique of LaTTe for the pseudocode. A comparison with previous JIT compilation techniques is given in Section 2.4. Section 2.5 overviews the LaTTe JVM, its JIT compiler, and other performance optimizations. Section 2.6 presents our experimental results.

## 2.1 Java Virtual Machine and SPARC

The Java VM is a typed stack machine [3]. Each thread of execution has its own Java stack where a new activation record is pushed when a method is invoked and is popped when it returns. An activation record includes state information, local variables, and the operand stack. All computations are performed on the operand stack and temporary results are saved in local variables, so there are many pushes and pops between the local variables and the operand stack.

The calling conventions for a Java method are as follows: The actual parameters are pushed on the operand stack of the caller method before a call is made. In the case of a virtual method call `invokevirtual`, the `this` reference is also pushed as the first parameter. The JVM pops those parameters and moves them into local variables of the callee method in order, starting from local variable zero. When a (nonvoid) Java method returns, the return value is pushed on top of the caller's operand stack.

SPARC is a 32-bit RISC machine with a register-based instruction set [4]. A function has its own register window which consists of 24 consecutive integer registers: eight in registers (`%i0-%i7`), eight local registers (`%l0-%l7`), and eight out registers (`%o0-%o7`).<sup>1</sup> When a method is called, the register window is rotated, such that the callee gets a new register window, where the callee's in registers overlap the caller's out registers. This facilitates argument passing: the caller passes arguments in `%o0-%o5`, which can be retrieved by the callee in `%i0-%i5`. The callee saves the return value in `%i0` which can be retrieved by the caller in register `%o0` when the called method returns. In addition, each method has its own C stack frame in memory, with a reserved 64-byte register-window save area for saving the local registers when a trap is raised; LaTTe uses this for exception handling.

---

<sup>1</sup>LaTTe uses 20 registers for allocation (excluding `%i6`, `%i7`, `%o6`, `%o7`). In the SPARC notation, the destination is the last operand, e.g., “`add %i1, %i2, %i3`” means “`%i3 = %i1 + %i2`” and “`mov %i1, %i2`” means a copy “`%i2 = %i1`”.

## 2.2 Bytecode Translation with Aggressive Register Mapping

When a method is called for the first time, LaTTe translates its bytecode into SPARC code. In LaTTe, there are two issues in translating bytecode into register-based code. One is converting stack entries and local variables into symbolic registers, which we call register mapping. The other is assigning symbolic registers to real registers, which we call register allocation. This section deals with register mapping. We will first discuss some JIT compiler design issues pertaining to register mapping, and we will then show how each bytecode is translated.

### 2.2.1 Issues in Register Mapping for Bytecodes

There are a few JIT compiler design issues related to register mapping for bytecodes. The JIT compiler designer first needs to decide if registers will be used for stack entries only, or for local variables only, or for both. Obviously, mapping both the stack entries and local variables to registers would be better, but it would require a nontrivial but fast register allocation scheme, which must also be able to remove register copies corresponding to pushes and pops between stack entries and local variables. The JIT compiler designer also needs to decide whether to generate register-allocated code directly from the bytecode in a single pass, or to have a separate pass to generate pseudocode with symbolic registers, followed by real register allocation. The former approach would be faster, yet may constrain register allocation by preallocating fixed registers to some stack entries or local variables, to reduce allocation complexity. The latter would be more versatile in terms of allocating registers and eliminating copies, but it could be slower. LaTTe uses registers for all stack entries and local variables. It also has a separate pass to generate pseudocode in order to allocate registers and remove copies in a highly flexible way. The translation process is composed of four stages. In the first stage, LaTTe identifies all control join points and subroutines (finally blocks) in the method's bytecode via a depth-first traversal. In the second stage, the

bytecode is translated into a control flow graph (CFG) of pseudo SPARC instructions with symbolic registers. In the optional third stage, LaTTe optimizes the pseudocode. In the fourth stage, LaTTe performs fast register allocation, generating a CFG of real SPARC instructions, which is finally converted into SPARC code. In the remainder of this section, we focus on the second stage and the next section focuses on the fourth stage.

## 2.2.2 Translation of Bytecode into Pseudocode

This section describes the translation of key bytecode instructions into SPARC primitives with symbolic registers. The translation rule for each bytecode instruction is determined based solely on the operand types and the opcode of the instruction itself. When this independently generated SPARC code fragment for each bytecode is concatenated with others, the resulting code becomes correct because consistent formats are used for symbolic registers, especially for those corresponding to stack elements; their format includes information on the current operand stack status, called TOP (explained shortly). A symbolic register in the pseudo SPARC code is composed of three parts:

- The first character indicates the type: a = address (object reference), i = integer, f = float, l = long, and d = double.
- The second character indicates the location: s = operand stack, l = local variable, t = temporaries generated by LaTTe for translation purposes.
- The remaining number further distinguishes the symbolic register.

For example, al0 represents a local variable 0 whose type is an object reference. is2 represents the second item of the operand stack whose type is an integer. TOP is a translation-time variable used by LaTTe (not a value computed at runtime) which indicates the number of items on the operand stack just before translating the current bytecode instruction. For example, if the current value of TOP is 4, “add isTOP-1,

isTOP, isTOP- 1” means “add is3, is4, is3.” There is another translation-time array, type[1..TOP] which indicates the type of each item (one of a, i, f, l, d) currently on the stack (required for translating dup/pop). LaTTe traverses the bytecode of a method in depth-first order, starting at the beginning of the method with TOP set to zero. Following any path of the bytecode, when a bytecode instruction that pushes some item(s) on the stack is encountered, TOP is incremented by the number of pushed items. Similarly, when a bytecode instruction that pops some item(s) is encountered, TOP is decremented by the number of popped items. The type array type[] is appropriately updated by the type of pushed items. According to the JVM specification [3] paragraph 4.9.2, this translation-time computation of the operand stack status is justified, since if the number of items on the operand stack is N on one path from the beginning to a given point, the operand stack must have the same number of items N and the same types of items in the same order on any path arriving at the same point [3]. In fact, the JVM verifier checks if this property is violated during the class loading.

### Stack/Local Variable Manipulation Instructions

Due to the stack computation model, bytecode instructions that push a local variable onto the stack or pop the stack top into a local variable are executed frequently. These are translated into symbolic register copies as follows (\$ means a translation-time action, not a runtime action).

```

iload n                                // stack: ... => ..., (local variable n)
  mov    il{n}, is{TOP+1}              // means a copy "is{TOP+1} = il{n}"
  $TOP = TOP + 1                       // The stack now has one more item.
  $type[TOP] = 'I'                     // The type of the new item is integer.

astore n                               // stack: ..., (object reference) => ...
  mov    as{TOP}, al{n}
  $TOP = TOP - 1                       // The stack now has one less item.

```

It should be noted that these symbolic register copy instructions do not really gen-

erate code because they will be coalesced during the register allocation phase.

## Arithmetic/Logical/Shift Instructions

The arithmetic/logical/shift bytecode instructions that operate on the top items of the operand stack can be directly mapped to one or two pseudo instructions.

```

iadd                                // stack: ..., x, y => ..., (x+y)
  add      is{TOP-1}, is{TOP}, is{TOP-1}
                                           // means a copy "is{TOP+1} = il{n}"
$TOP = TOP + 1                        // The stack now has one more item.

```

## Object Access Instructions

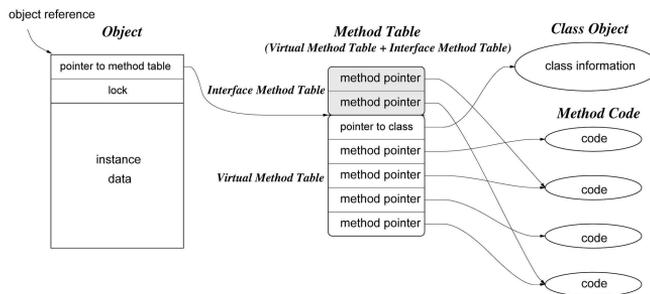


Figure 2.1: The object model of LaTTe

Figure 2.1 depicts the object model of LaTTe. An object includes two fields before the instance data: a pointer to the virtual/ interface method table and a 32-bit lock, which are for method invocation and for thread synchronization, respectively. The instance data can be accessed by a single memory access, compared to two accesses used in some implementations of the JDK [3]. Here is an example pseudocode for accessing the integer field `foo` of an object.

```

getfield <x.foo>                        // stack: ..., (object ref) => ..., (integer)
  ld      [as{TOP} + foo_offset], is{TOP}
                                           // "is{TOP} = load @[as{TOP}+foo_offset]"
$stype[TOP] = 'I'                        // foo_offset is a constant.

```

```

putfield <x.foo>          // stack: ..., (object ref), (integer) => ...
    st      is{TOP}, [as{TOP-1} + foo_offset]
$TOP = TOP - 2

```

The JVM is required to throw a `NullPointerException` if the object reference is `NULL`. LaTTe does not generate such check code here because if the object reference is `NULL`, a `SIGSEGV` or `SIGBUS` signal will be raised by the operating system during the execution of the load/store; the LaTTe JVM includes a signal handler where the `NullPointerException` is thrown.

### Method Invocation Instructions

The LaTTe JVM maintains a virtual method table for each loaded class. The table contains the start address of each method defined in the class or inherited from the super- class. Due to the single inheritance property of Java, if the start address of a method is placed at offset  $n$  in the virtual method table of a class, it can also be placed at offset  $n$  in the virtual method tables of all subclasses of the class. Consequently, the offset  $n$  is a translation-time constant. Since each object includes a pointer to the method table of its corresponding class as shown in Fig. 1, a virtual method invocation can be translated into an indirect function call after two loads, as follows:

```

invokevirtual <x.func>    // assume func takes two integer arguments
                        // and returns as integer.
                        // stack: ..., (object ref), (int), (int)
                        //      => ..., (int)

ld      [as{TOP-2}], at0
// pointer of the table is located at offset 0 in the object.
ld      [at0 + func_offset], at1
// pointer of func is located at func_offset in the table
call   at1
$TOP = TOP - 2          // The stack now has two less items.
$type[TOP] = 'I'      // foo_offset is a constant.

putfield <x.foo>          // stack: ..., (object ref), (integer) => ...
    st      is{TOP}, [as{TOP-1} + foo_offset]

```

$\$TOP = TOP - 2$

In the above example, the virtual method `x.func` is assumed to have two integer arguments and to return an integer value. At call `at1`, these two arguments and the implicit `this` argument are mapped to symbolic registers  $is\{TOP\}$ ,  $is\{TOP-1\}$ , and  $as\{TOP-2\}$ , respectively. Also, the return value is mapped to a symbolic register  $is\{TOP-2\}$  when the method returns.

It is desirable to allocate these symbolic registers following the SPARC calling conventions. In our example, the argument registers,  $as\{TOP-2\}$ ,  $is\{TOP-1\}$ , and  $is\{TOP\}$ , are preferably allocated into `%o0`, `%o1`, and `%o2`, respectively; otherwise, we should insert copies before the call instruction. Similarly, the return value register  $is\{TOP-2\}$  after the call should be allocated into `%o0`.

The calling conventions should also be followed at the callee side. At the beginning of `x.func`, the `this` argument and the two integer arguments are mapped to local symbolic registers  $al0$ ,  $il1$ , and  $il2$ , respectively. These registers must be allocated into `%i0`, `%i1`, and `%i2`, respectively. The return value symbolic register,  $is0$  at the end of the method, must be allocated to `%i0`. Section 4 describes how LaTTe can allocate registers following the calling conventions.

The LaTTe JVM also maintains an interface method table for each class which lists the start address of each method implementing an interface method. Each interface method is assigned a globally unique offset so that `invokeinterface` is also translated into an indirect function call after two loads. This is faster than searching the virtual method table although it incurs some space overhead. We have currently seen a maximum of 150 entries in an interface method table.

### Array Access Instructions

Arrays in Java are objects. The layout of a LaTTe array object starts with the same two fields as in Fig. 1, followed by the array length and the array data. The JVM is supposed to check array bounds for all array accesses. LaTTe inserts the bound check code based on a trap, as opposed to branches around calls to error routines, in order

to simplify control flow. The signal handler takes care of throwing the exception. The check of a NULL array reference is handled by SIGBUS as previously. The translation of `iaload`, for example, is as follows:

```
iaload          // stack: ..., (array ref), (index) => ..., array[index]
// array bound checking code
ld      [as{TOP-1} + offset_of_array_length_field], it0
subcc   is{TOP}, it0, %g0 // %g0 is a global register
                        // that contains zero.
tcc     0x10          // traps if is{TOP} < 0
                        //      or is{TOP} >= array_length
// load the array element
// can be removed if we arrange the first element at offset 0
add     as{TOP-1}, offset_of_array_data, it0
sll     is{TOP}, 2, it1 // since the sizeof(int) is 4 bytes
ld      [it0 + it1], is{TOP-1}
$TOP = TOP - 1
$type[is{TOP}] = 'I'
```

## A Translation Example

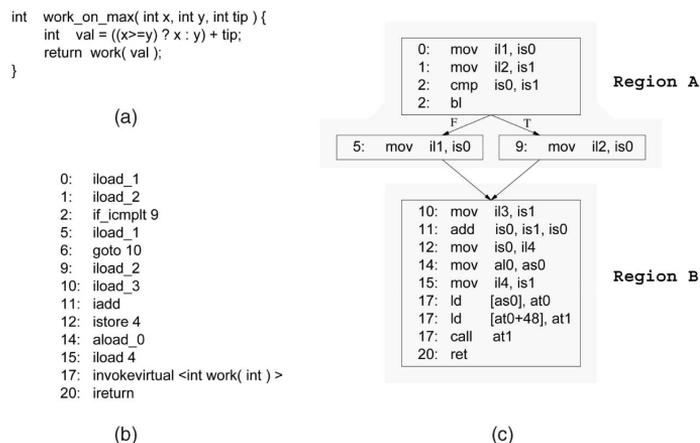


Figure 2.2: A translation example from bytecode into pseudo SPARC code. (a) Java source, (b) bytecode, and (c) CFG of pseudo SPARC code.

Figure 2.2 shows a simple translation example. The instance method *work\_on\_max()* in (a) simply takes the maximum of two values, adds a tip value, and calls another instance method *work()* (whose offset in the method table is 48). Starting from the first bytecode in (b) with TOP = 0, translation of each bytecode will generate the pseudocode in (c).

## 2.3 Fast Register Allocation

The translation rules described above indicate that it is simple to convert the bytecode into SPARC code with symbolic registers. We now describe our fast register allocator which effectively coalesces copies and conserves registers. The technique is based on the left-edge greedy interval coloring algorithm [5], extended to a larger region of code called the *tree region*.

### 2.3.1 Tree Regions

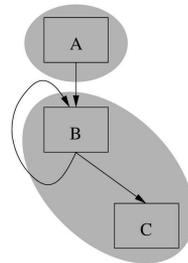


Figure 2.3: A CFG of basic blocks and tree regions.

The CFG of pseudocode is partitioned into tree regions which are single-entry, multiple-exit subgraphs shaped like trees. Tree regions start at the beginning of the program or at control join points and end at the end of the program or at other join points. For example, the CFG in Figure 2.3, composed of three basic blocks (A, B, C), has two tree regions depicted by shaded areas.

A tree region is a unit of optimizations in LaTTe, such as redundancy elimination, common subexpression elimination, constant propagation, loop invariant code motion, as well as register allocation. Tree regions can be enlarged by code duplication techniques such as loop unrolling to increase the opportunity for optimization in frequently executed parts of code. By working on tree regions, LaTTe trades off quality and speed of optimization.

After regions are constructed for a method, last uses of each symbolic register are computed. Stack symbolic registers are supposed to be dead once they are used, and the live range of temporary symbolic registers cannot span beyond the translated code sequence for a bytecode instruction. Consequently, last uses of these symbolic registers can be readily identified. For local symbolic registers, however, liveness is computed “approximately” via a single postorder traversal [6] of the regions such that every local symbolic register is assumed to be live on a backward edge of the CFG. This gives a conservative, yet fast, estimation of live variables in each region. Based on the liveness information, we can identify the last use of each local symbolic register. When a local symbolic register is dead on one path of a conditional branch while it is live on the other path, we mark the path where it is dead with the last use for the register.

The regions are then register-allocated one by one in a reverse postorder traversal of the regions, such that a region is allocated before its descendents are allocated, in a depth-first spanning tree of regions [6]. In each region during the traversal, the tree is traversed twice, first by postorder which is called the backward sweep, followed by preorder which is called the forward sweep. The backward sweep collects information on the preferred destination registers for instructions, which works as a local lookahead. The forward sweep performs real register allocation using that information. During each traversal, a map which is a set of (symbolic, real) register pairs is collected and propagated following the traversal direction. The map is called `p_map` in the backward sweep which describes preferred assignments for destination symbolic registers, and `h_map` in the forward sweep which describes the current register allocation result

of symbolic registers.

### 2.3.2 Backward Sweep and Forward Sweep

```
backward_sweep (instr)
  returns p_map // set of (symbolic,real) register pairs
{
  p = NULL;
  if (instr is the header of a next region) { // reached a region boundary
    if (there is an old h_map, h_old, saved at instr) // the region boundary has been visited
      p = h_old; // use the result of a prior forward sweep in other regions
    return p;
  }
  for (each successor s of instr) {
    p = merge_map(p, backward_sweep(s));
  }
  // Now, compute the p_map above instr
  if (instr has a target symbolic register x AND there is a preferred assignment (x,r) in p) {
    preferred_assignment(instr) = r; // if instr is a copy, preferred assignment is not used
    if (instr is a copy x=y ) { insert (y,r) into p; }
    delete (x,r) from p; // (x,r) is not propagated above instr since r is already defined at instr
  } else if (instr is a call) {
    p -= (returned value symbolic register, *);
    p += (argument symbolic register, out register) pairs;
  } else if (instr is a return) {
    p += (returned value symbolic register, return register);
  }
  return p;
}
```

Figure 2.4: The backward sweep algorithm.

The `backward_sweep()` algorithm in Figure 2.4 is called with the root of the region as an argument. The purpose of the backward sweep is computing the `p_map`, based on the required register assignment at the end of the region, or at method calls/returns according to the calling conventions. For example, if a symbolic register `il2` is to be allocated to a real register `r` at a method call due to the calling conventions, and we have an operation sequence "add `is1, is2, is1`; mov `is1, il2`" just before the call, then the destination register `is1` of the add is preferably allocated to `r` to avoid a copy. This preference can be known if the `p_map` propagated through the add includes  $(is1, r)$ .

Copies are important in computing the `p_map`. If the `p_map` includes  $(x, r)$  under a copy "mov `y, x`", then the `p_map` above the copy includes  $(y, r)$ . At a conditional branch, the `p_map` of both paths are *unioned*, yet if there are two different `p_map` for a

symbolic register, an arbitrary one is taken. If the destination register of an instruction is included in the `p_map`, its preferred assignment is set to its mapped real register in the `p_map`. Figure 2.4 shows this process in detail.

```

forward_sweep (instr, h, refcount, freereg)
{
  if (instr is the header of the next region) {
    if (instr does not have h_old assigned yet) {
      h_old(instr) = h;
      incremental_backward_sweep(instr); // if other paths reaching instr from the same region exist
    } else Reconcile_h_map(h, h_old(instr)); // reconcile by inserting copies
    return;
  }
  // parse the instruction as "z = x + y"
  real_rhs = "h[x] + h[y]";
  if (x is a last use) {
    if ((refcount[h[x]--] == 0) freereg += {h[x]};
    delete (x,h[x]) from h;
  }
  if (there is a second operand y != x AND y is a last use) {
    if ((refcount[h[y]--] == 0) freereg += {h[y]};
    delete (y,h[y]) from h;
  }
  if (instr has a destination register z) { // Now, determine the target register of instr
    if (instr is a copy z = x) real_lhs = real_rhs;
    else if (instr has a preferred assignment r for z AND r is in freereg) real_lhs = r;
    else if (freereg != NULL) real_lhs = first available one in freereg;
    else real_lhs = spill_a_real_register();

    h[z] = real_lhs; refcount[real_lhs]++;
    if (real_lhs != real_rhs) Generate("real_lhs = real_rhs");
  } else Generate("real_rhs");

  for (each successor s of instr) {
    kill_dead_variables(s, h, refcount, freereg);
    forward_sweep(s, h, refcount, freereg);
  }
}

```

Figure 2.5: The forward sweep algorithm.

After the preferred assignments for instructions are computed, the forward sweep is performed to allocate real registers. The `forward_sweep()` algorithm in Figure 2.5 is called at the root of the region with `h`, an `h_map` that is saved at the root. Other arguments include `refcount` that shows how many symbolic registers are mapped to each real register and `freereg` which indicates the set of real registers to which no symbolic registers map, as determined from `h`. For the starting region of a method, `h` is initialized by the map of parameters. For example, `h` for the method `x.func` in Section 3.2.4 is initialized by `(a0, %i0), (i1, %i1), (i2, %i2)`. As regions are

allocated in reverse postorder,  $h$  at the end of a region is propagated to the root of the next region and saved there.

The allocation is performed with a preorder traversal of the tree from the root. When an instruction  $z = x + y$  is encountered, the real code is generated as follows. First, the right-hand-side is generated as  $h[x] + h[y]$ . If the  $x$  use is the last use of  $x$ , the `refcount` of the real register  $h[x]$  is decremented by one, and  $h[x]$  is added to the `freereg` if the `refcount` becomes zero, and  $(x, h[x])$  is deleted from  $h$ , meaning that  $x$  is now dead. The same is done for  $y$ . For the target register  $z$ , if the instruction is a copy  $z = x$  and  $x$  was mapped to a real register  $r$ , then  $z$  is also allocated into  $r$ , meaning that the copy is coalesced. For noncopy instructions, if there is a preferred assignment for the instruction (a real register that  $z$  will eventually be mapped into) and if it is in `freereg`, we choose the register. Otherwise, we choose the first free register in `freereg`. If `freereg` is empty, we need to spill, which will be described shortly. Now, the pair  $(z, \text{the chosen real register})$  is inserted into  $h$ . After the `forward_sweep()` passes through a conditional branch, if some symbolic register  $x$  is dead on a path,  $(x, h[x])$  is deleted from  $h$ , and `refcount` and `freereg` are also updated.

Starting from the root of a region, all instructions are register-allocated as described above. When the root of the next region is encountered, we save the current `h_map` at that root so that the forward sweep at the next region can start with this as an initial `h_map`. Since the root is a join point, more than one forward sweep may reach the same root. If some `h_map` is already saved at the root when the current forward sweep reaches it, we need to reconcile the current `h_map` and the old one that has already been there by inserting some copies, as described below.

### 2.3.3 Reconciling `h_map` at Region Join Points

Let us call the old `h_map` and the new `h_map` `h_old` and `h_new`, respectively. Assume  $h\_old[x] = h\_old[y] = r$ . If  $h\_new[x] = h\_new[y] = r'$ , we need to

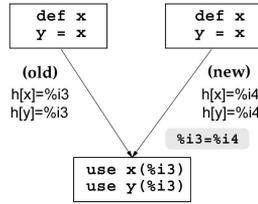


Figure 2.6: Reconciling register allocation.

insert a copy  $r = r'$  on the new incoming edge as shown in Figure 2.6. This conserves the old mapping, namely,  $h[x] = h[y] = r$ .

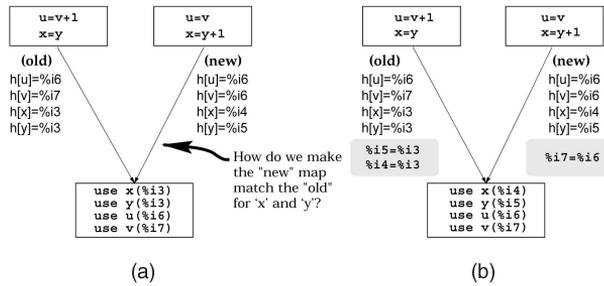


Figure 2.7: Reconciling register allocation. (a) Problem and (b) solution.

If  $h_{\text{new}}[x]$  is different from  $h_{\text{new}}[y]$ , however, there is a problem. Suppose in the new mapping,  $h_{\text{new}}[x] = r'$  but  $h_{\text{new}}[y] = r''$ . This can happen if there is  $x = y + 1$  on the new incoming edge which makes  $y$  unequal to  $x$  while there is  $x = y$  in the old incoming edge, making  $y$  equal to  $x$ . Figure 2.7 (a) depicts this situation. It also shows the opposite case, i.e.,  $u = v + 1$  is on the old incoming edge while there is  $u = v$  on the new incoming edge, which is easier to handle.

As shown in Figure 2.7 (b), it is still possible to reconcile the mapping by inserting copies in the old incoming edge. One issue is that if the region has already been allocated using  $h_{\text{old}}$  before  $h_{\text{new}}$  reaches the region, we might need to reallocate the region and probably its successor regions, which will be expensive. Fortunately, since we traverse regions in reverse postorder, this can occur only at a loop entry region; when a loop entry region is encountered following the back edge, it would have

already been allocated using `h_map` propagated through the loop entry edge.

In order to handle this, when a loop entry region is encountered for the first time, we force each pseudo register to be mapped to a separate real register by inserting copies (e.g., in Figure 2.7 (a), we insert a copy `mov %i3; %i4` at the old incoming edge and update  $h[y] = \%i4$ ). In this way, when the loop entry is encountered again through the back edge, we do not have to update the previous  $h$  of the region nor reallocate the region; we just add copies at the back edge if required.

Reconciliation overhead is, in practice, small due to the backward sweep. Let us assume that region A and C are predecessors of region B, and A is allocated first. The forward sweep at region A will save its `h_map` at the root of region B. Then, the backward sweep at region C will take the saved `h_map` as an initial value of its `p_map` and propagate across region C. So, the forward sweep at region C will generate an `h_map` more compatible with A's, which can reduce reconciliation.

Our algorithm also handles a case when there is more than one edge from region A to region B. In Figure 2.2 (c), for example, when the forward sweep at region A reaches the root of region B for the first time following the false path, we save the current `h_map` at the root. We know the true path from A also reaches the same root but has not yet been forward swept. At this point, we perform an incremental backward sweep for the true path to give preferred assignments based on the saved `h_map` from the false path. This will also reduce reconciliation when the forward sweep on the true path reaches the root of region B. Figure 2.5 includes the consideration for this case. The reconciling problem, in fact, is similar to replacing SSA nodes by a set of equivalent move operations [7] and we can use the same solution to minimize copies.

### 2.3.4 Register Spill

When no free registers are available at some instruction  $I$  during the forward sweep, we heuristically choose a real register  $r$  to spill. Let us assume that  $r$  is mapped only to pseudo registers  $x$  and  $y$  at that point ( $h[x] = h[y] = r$ ). We insert a

store instruction to a spill location `SPILL0` just before `I` and mark `x` and `y` last uses there. We then register allocate the inserted store, generating `st r; SPILL0` (since  $h[x] = r$ ). We now map the symbolic registers `x` and `y` to `SPILL0` (i.e.,  $h[x] = h[y] = SPILL0$ ) and `r` is moved back to `freereg` with its `refcount` zero. In this way, the forward sweep can continue at `I` with a new available register `r`. When a spilled register is used later by an instruction, say `add x; 2; w`, we replace the instruction by a new sequence of instructions, `[ld SPILL0, x; mov x, y; add x, 2, w;]` (the copy is needed since both `x` and `y` had the same value when spilled), and continue the register allocation. When the load and the copy are register allocated, `x` and `y` might be allocated to a different register this time, say `r'`. Both `x` and `y` are mapped to `r'`, and the `refcount` of `r'` is set appropriately.

At a region boundary, reconciling copies may occasionally include spill locations (e.g., `SPILL0 = r3, r3 = SPILL1`, or `SPILL0 = SPILL1`) as well as normal register copies. We handle them appropriately.

### 2.3.5 A Register Allocation Example

Figure 2.8 describes the register allocation process for the example in Figure 2.2. There were two regions in Figure 2.2 (c). The backward sweep and the forward sweep for the region A and the region B are described in Figure 2.8 (a) and (b), respectively.<sup>2</sup> The final register allocation result is shown in Figure 2.8 (c), where only the essential code is generated.

The difference and novelty of our register allocation algorithm compared to the original left-edge interval coloring algorithm [5] are as follows: Our algorithm uses aggressive copy elimination to avoid generating code for copy operations. It maps multiple symbolic registers to the same real register when they are equal, and uses

---

<sup>2</sup>The incremental backward sweep is not shown because it does not affect the allocation result in this example.

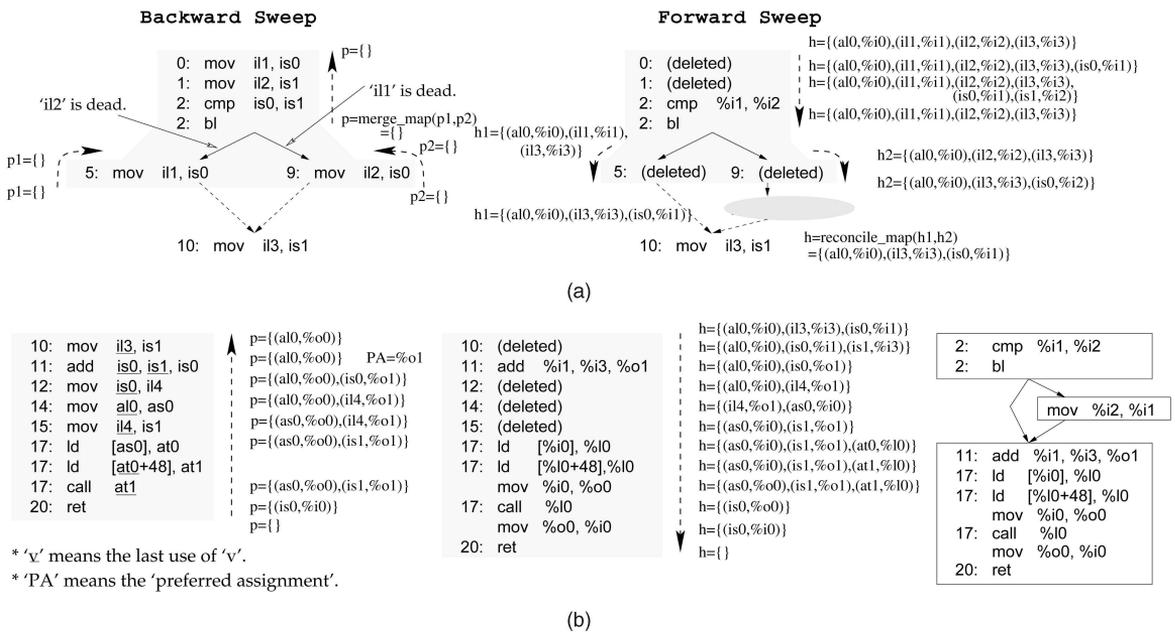


Figure 2.8: Register allocation process for the previous example. (a) Register allocation of Region A and (b) register allocation of Region B.

clever heuristics to match physical register assignments across tree region boundaries, in order to avoid introducing copy operations in such boundaries.

## 2.4 Comparison with Previous JIT Compilation Techniques

It is highly desirable to be able generate high-performance native code for a bytecode instruction, while keeping the translation process fast. The quantity: *(total compilation time for the bytecode) + (number of executions of the bytecode) \* (average execution time of the translated bytecode)* must be minimized, in order to reduce the contribution of a bytecode instruction to the total execution time. Hence, finding the right trade-off between translation time and execution time can be very important.

Modern adaptive JIT compilers selectively resort to traditional compiler optimizations which can consume a lot of time, but only for “hot-spot” methods, while interpreting or performing only moderate compiler optimizations on the less frequently executed parts of the program. Indeed, compile time pressure goes away when true hot-spots with very high re-use rates exist. However, continuously detecting the hot-spots accurately and with low overhead can itself be difficult; also, some programs do not have code fragments that are hot enough and worthy of a time-consuming optimization effort. Hence, a base compilation technique similar to LaTTe’s, that can already quickly generate high performance native code from the start (along with hardware and OS assistance for accurate profiling), could be helpful for all JIT compilers, including those following a profile-directed adaptive strategy for hot-spots.

In this section, we compare LaTTe’s JIT compilation technique with some of those earlier JIT compilation techniques that translate all executed methods, including Kaffe [8], VTune [9], and CACAO [10], focusing on quality and speed of register allocation. We then describe register allocation techniques employed by adaptive compilation techniques.

Kaffe is a public-domain JVM with a relatively simple JIT compiler. Kaffe detects

\*Assume a map of {(local0,%i0),(local1,%i1),(local2,%i2),(local3,%i3),(local4,%i4)} in VTune.

\*Assume a map of {(L0,%i0),(L1,%i1),(L2,%i2),(L3,%i3),(L4,%i4),(I0,%i0),  
(A0,%o0),(A1,%o1),(R0,%o0)} in CACAO.

	Mimic Stack	Code	Static Stack	Code	Code
0: iload_1	(%i1)		(L1)		
1: iload_2	(%i1,%i2)		(L1,L2)		
2: if_icmplt 9	()	cmp %i1, %i2 bl Label2	()	cmp %i1, %i2 bl Label2	cmp %i1, %i2 bl Label2
5: iload_1	(%i1)	Label1:	()	Label1:	Label1:
6: goto 10	(SPILL0)	st %i1, SPILL0 b Label3	(I0)	mov %i1, %i0 b Label3	b Label3
9: iload_2	(SPILL0)	Label2:	(I0)	Label2:	Label2:
		st %i2, SPILL0		mov %i2, %i0	mov %i2, %i1
10: iload_3	(SPILL0)	Label3:	(I0)	Label3:	Label3:
11: iadd	(SPILL0,%i3)	ld SPILL0, %i0 add %i0, %i3, %i1	(I0,L3)	add %i0, %i3, %i4	add %i1, %i3, %o1
12: istore 4	(%i1)	mov %i1, %i4	(L4)		
14: aload_0	()		()		
15: iload_4	(%i0)		(A0)	mov %i0, %o0	ld [%i0], %i0
17: invokevirtual <work>	(%i0,%i4)	mov %i4, %o1	(A0,A1)	mov %i4, %o1	ld [%i0+48], %i0
	(%o0)	ld [%o0], %i2	(R0)	ld [%o0], %i0	mov %i0, %o0
		ld [%i2+48], %i3		ld [%i0+48], %i0	call %i0
		call %i3		call %i0	mov %o0, %i0
20: ireturn	(%o0)	mov %o0, %i0 ret	(R0)	mov %o0, %i0 ret	ret
	(a)	(b)		(c)	(d)

Figure 2.9: Translation by VTune and CACAO. (a) Bytecode, (b) VTune, (c) CACAO, and (d) LaTTe.

basic blocks and performs single-pass code generation with register allocation (i.e., it generates no pseudocode). For all local variables and operand stack slots, there are corresponding entries in the C stack of the translated method. If a variable or a stack slot is used in a basic block, a register is used to load it from the C stack. At the end of a basic block, registers corresponding to locals or stack slots that have been defined in the basic block are spilled back to the C stack. Consequently, there are many loads/stores in the translated code.

Intel's VTune includes a JIT compiler for its x86 platform, yet the technique itself is applicable to RISC machines as well. All local variables are globally preallocated before the translation starts. Then, single-pass code generation is performed with local register allocation for stack slots and temporaries. A mimic stack is computed during the translation to trace the current operand stack which contains registers and the C stack addresses corresponding to local variables and temporaries. Lazy code generation with the mimic stack avoids many copies corresponding to xload, yet copies from the operand stack to local variables corresponding to xstore are generated. When the mimic stack is not empty at the end of a basic block, all stack entries are spilled to the C stack.<sup>3</sup> Figure 2.9 (b) shows the translation process by VTune for our previous example in Figure 2.9 (a). The VTune code can be compared with the LaTTe code shown in Figure 2.9 (d).

CACAO is a JIT compiler targeting the Alpha platform. Each local variable is also preallocated as in VTune, yet for operand stack slots which are live beyond a basic block, interface pseudo registers are allocated instead of spill locations in the C stack. CACAO first converts the bytecode into an intermediate form and analyzes the operand stack to build a static stack for each instruction which contains local variables and interface registers (i.e., not real registers). Delayed code generation using the static stack also avoids many copies corresponding to xload, yet CACAO can also

---

<sup>3</sup>Another version of VTune uses priority-based coloring, yet for most benchmarks, it gives worse results [9].

avoid some copies corresponding to `xstore` if its target local variable can be used as a destination for the computation result at the stack top (e.g., `[iload a; iload b; iadd; istore c;]` can be translated into “`add a; b; c`”). This is possible because CACAO performs more elaborate analysis on the intermediate code. Figure 2.9 (c) shows the translation process of CACAO.

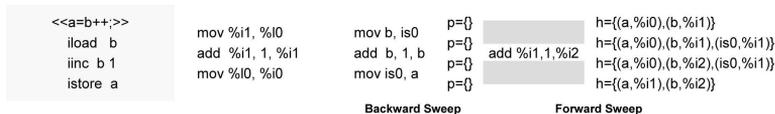


Figure 2.10: An inefficient translation example.

The approach of VTune/CACAO based on a simulated operand stack, has two types of inefficiencies compared to LaTTe. First, the fixed preallocation of local variables generates inefficient code. In LaTTe, if one local variable is copied into another variable (e.g., through the `xload-xstore` sequence), they can be allocated to the same register. This means that LaTTe can conserve registers better and can eliminate more copies than VTune/CACAO. LaTTe can also allocate different registers to different live ranges of a variable, if required. This is hard to achieve in VTune/CACAO because of the fixed preallocation, which might even cause some difficulty in code generation. For example, if there is an update of a variable while its previous value resides in the static/mimic stack due to a previous `xload`, the copy for the `xload` cannot be avoided. Figure 2.10 shows an example for a Java statement `a = b++` where a copy for `iload` cannot be avoided. A copy for `istore` cannot be avoided in VTune, and mostly in CACAO.<sup>4</sup>

Another inefficiency is that VTune/CACAO gives up coalescing at join points. When the mimic/static stack is not empty at a join point, all stack entries are mapped to the C stack/interface registers, always generating spills/copies, respectively. On the

<sup>4</sup>CACAO’s copy elimination for `xstore` is impossible if preallocation causes non-true data dependencies, e.g., for `[iadd; iload a; istore b; istore a;]`, we cannot remove the copy corresponding to “`istore a`” due to “`iload a`”.

other hand, LaTTe resolves join conflicts, coalescing the copy between the stack and the local variable at least for one path. A typical example is the Java condition statement<sup>5</sup> `a = (b > c) ? b : c`, in Figure 2.8 and 2.9. For this example, VTune and CACAO generated four and two more operations than LaTTe, respectively.

Many recent JVM JIT compilers employ more elaborate register allocation algorithms due to their adaptive compilation framework. The HotSpot JVM uses interpretation to detect hot spots and then uses a JIT compiler to compile and optimize such hot spots [11], [12]. The JIT compiler uses a global graph coloring allocator based on Briggs' and Chaitin's algorithm with some refinements for allocation speed and code quality. The Jalapeño JVM [13] and its enhanced open-source version called Jikes RVM [14] employ compile-only adaptive compilation. Each method is compiled by a quick compiler when it is first executed, and then is recompiled by an optimization compiler if it is computationally intensive. The optimizing compiler uses a linear-scan register allocation (LSRA) algorithm [15]. The major differences between LSRA and LaTTe are as follows: First, LaTTe coalesces copies aggressively during register allocation while LSRA does not and focuses on fast register allocation itself. Second, LaTTe employs backward sweep in order to reduce more copies, especially from those caused by calling conventions, yet LSRA does not have such a phase. Finally, the unit for register allocation is tree region in LaTTe, but it is a sequence of instructions in LSRA.

The IBM JIT compiler also uses interpreter-based adaptive compilation, yet its register allocation algorithm is simpler [16]. Frequently used local variables are allocated to physical registers first, and then the remaining registers are used for stack variables. When spilling is needed, the least recently used register is spilled to avoid any complex computation to search spill candidates.

---

<sup>5</sup>In Java standard class libraries, there are many source files that include a call to `Math:min` or `Math:max`. We found that the corresponding bytecode is not a static method call, rather it is an inlined sequence of bytecode for this conditional form of Java code. Therefore, the conditional form occurs rather frequently.

## 2.5 More Optimizations in the LaTTe JVM JIT Compiler

### 2.5.1 Optimizations in the JIT Compiler

The register mapping and allocation techniques comprise the basis of the LaTTe JIT compiler. It also includes other optimization techniques and is well-coordinated with other JVM components. In this section, we briefly overview the LaTTe JIT compiler and its other JVM components.

There are two versions of the LaTTe JIT compiler: a base version (`-Obase`) and an optimized version (`-Oopt`). The base version performs only the fast register allocation described previously without any other optimizations. The optimized version performs two additional optimizations: “traditional” optimizations and limited object-oriented (OO) optimizations. For traditional optimization, LaTTe performs common subexpression elimination (CSE), redundancy elimination (RE), loop invariant code motion (LICM), and inlining of static, private, and final methods. Many of these optimizations are performed on a unit of tree region.

LaTTe’s OO optimization is primarily for reducing the virtual call overhead of load-load-jump. LaTTe performs two such optimizations: customization [17] and dynamic inline patching [18], [19]. Customization creates a “*specialized*” version of a method based on the actual receiver type of a virtual call. With dynamic inline patching, both the inlined version and the load-load-jump sequence are generated, but the inlined version is executed until the target method is overridden.

### 2.5.2 On-demand Exception Handler Translation

Java uses *exceptions* to provide elegant error handling capabilities during program execution. Since an exception would be an “exceptional” event, LaTTe delays the translation of exception handlers in a method until the corresponding exception really occurs. The *exception manger* (EM) in the LaTTe JVM is responsible for locating/translating the exception handler. Invocation of the EM is initiated either from a call or from a trap

in the translated `try` block depending on the type of exceptions. The EM uses two tables: *exception table* (ET) loaded from the class file which is an array of (`from_pc`, `to_pc`, `handler_pc`, `catch_type`, `translated_code_ptr`), and *exception information table* (EIT), an array of (`native_pc`, `bytecode_pc`, `local_variable_map`). For each candidate of an exception-raising instruction (*e.g.*, calls, traps, loads, stores, and divisions), there is an entry in the EIT.

When an exception is thrown, the EM searches the EIT using the native address of the exception-raising instruction as a key to the corresponding bytecode address, which is used to search the ET for the chosen exception handler. In order to maintain the consistency of register allocation between `try` blocks and `catch` blocks for local variables, the `local_variable_map` field in the EIT keeps a map of (local variable, register number/spill address) that tells where each local variable is saved at that point. Based on the map, the EM copies the value of each local variable into a reserved location in the stack so that the translation of the chosen exception handler starts with an `h_map` where each local-variable symbolic register is mapped to the reserved stack location.

### 2.5.3 Lightweight Monitor for Synchronization

Java supports monitors, a language-level synchronization construct for multithreading. The LaTTe JVM includes an efficient user-level monitor implementation, called the *lightweight monitor* [21]. A 32-bit word dedicated to representing a lock is embedded in each object for efficient lock access (see Figure 2.1). The lock manipulation code is highly optimized and is inlined by LaTTe, so that only 9 SPARC instructions are spent for lock acquisition and 5 instructions for lock release in most probable cases.

### 2.5.4 Memory Management

Memory management is also crucial to JVM's performance. LaTTe allocates small objects using lazy worst fit [22], which usually allocates objects using pointer increments,

and uses worst fit to find a new free memory chunk if pointer-incrementing allocation does not work. LaTTe employs a partially conservative mark and sweep garbage collector, in the sense that the runtime stack is scanned conservatively for pointers while all objects located in the heap are handled in a type accurate manner. For the sweep phase, we use *selective sweeping* [23], which sorts all live objects by address and then frees each gap between live objects in constant time.

## 2.6 Experimental Results

In this section, we perform an evaluation of LaTTe’s JIT compilation technique. In order to evaluate whether LaTTe’s sophisticated register mapping and allocation really pays off, we compare the performance of LaTTe’s JIT compiler with that of Kaffe’s, by implementing both JIT compilers on the same LaTTe JVM. Then, we evaluate how LaTTe allocates registers.

### 2.6.1 Experimental Environment

Our benchmarks are composed of seven SPECjvm98 benchmarks [24], 12 Java Grande benchmarks [25], and 14 nontrivial Java programs we found from the public domain (listed in Table 1 with the translated bytecode size). They are a good mix of integer and floating-point programs.

Our test machine is a SUN Ultra5 270 MHz with 256 MB of memory, running Solaris 2.6, tested in a single-user mode. We ran each benchmark five times and took the minimum running time. In fact, there was little variance in those five running times.

### 2.6.2 Evaluation of LaTTe’s JIT Compilation Techniques

We modified the LaTTe JVM to use Kaffe’s JIT compiler as an execution engine, and compared its performance with that of the base version of the LaTTe JIT compiler. Since neither JIT compilers perform any serious optimizations other than the code

Table 2.1: LaTTe JVM running time (seconds) with LaTTe JIT and Kaffe JIT.

Benchmark	Translated Bytes	LaTTe ( <b>Kaffe</b> )		LaTTe ( <b>Base</b> )		Ratio	
		TR	TOT	TR	TOT	TR (base/Kaffe)	TOT (Kaffe/base)
<b>SPECjvm98</b>							
..201_compress	24315	0.28	144.06	0.81	69.71	2.89	2.07
..202_jess	45230	0.38	94.57	1.27	41.68	3.34	2.27
..209_db	26414	0.29	138.20	0.88	61.50	3.03	2.25
..213_javac	91963	0.63	150.77	2.38	52.79	3.78	2.86
..222_mpegaudio	38768	0.66	197.68	1.48	79.99	2.24	2.47
..227_mtrt	33998	0.34	106.97	1.07	50.71	3.15	2.11
..228_jack	51208	0.48	88.82	1.68	48.80	3.50	1.82
GeoMean (SPEC)						<b>3.10</b>	<b>2.24</b>
<b>Java Grande</b>							
Series	8162	0.14	80.11	0.37	57.35	2.64	1.40
LUFact	9393	0.15	13.74	0.40	7.69	2.67	1.79
SOR	8139	0.14	46.62	0.37	28.51	2.64	1.64
HeapSort	8087	0.14	13.68	0.37	8.99	2.64	1.52
Crypt	9264	0.15	24.36	0.39	17.73	2.60	1.37
FFT	8583	0.14	107.80	0.39	63.82	2.79	1.69
SparseMatMult	8148	0.14	78.55	0.37	69.80	2.64	1.13
Search	10505	0.15	121.37	0.43	63.17	2.87	1.92
Euler	22613	0.23	431.32	0.78	192.36	3.39	2.24
MolDyn	24105	0.36	431.85	2.03	44.60	5.64	9.68
MonteCarlo	14487	0.16	123.21	0.52	47.92	3.25	2.57
RayTracer	11020	0.15	230.33	0.41	75.32	2.73	3.06
GeoMean (GRANDE)						<b>2.96</b>	<b>2.04</b>
<b>Others</b>							
richard_g	7673	0.14	38.27	0.36	17.34	2.57	2.21
richard_gf	7760	0.14	38.71	0.36	16.16	2.57	2.40
richard_gns	7768	0.14	42.26	0.36	21.84	2.57	1.93
richard_dna	8026	0.15	90.74	0.36	33.01	2.40	2.75
richard_dav	8186	0.15	138.04	0.36	79.63	2.40	1.73
richard_daf	8186	0.14	133.76	0.36	47.34	2.57	2.83
richard_dai	8314	0.15	233.30	0.37	78.16	2.47	2.98
richard_dall	17189	0.17	662.93	0.52	290.82	3.06	2.28
spell	4505	0.12	27.28	0.23	21.18	1.92	1.29
javacc	125698	0.96	160.98	3.78	42.53	3.94	3.79
deltablue	9429	0.15	7.92	0.40	2.88	2.67	2.75
jjtree	62594	0.39	5.09	1.61	4.04	4.13	1.26
jlex	25785	0.22	3.16	0.75	1.76	3.41	1.80
hashjava	24437	0.23	2.18	0.74	2.51	3.22	0.87
GeoMean (Others)						<b>2.79</b>	<b>2.06</b>
GeoMean (all)						<b>2.92</b>	<b>2.09</b>

translation with register allocation, this experiment can evaluate the effectiveness of LaTTe’s sophisticated register mapping and allocation, compared against a naive one that maps local variables and stack slots to memory.

Table 2.1 shows the total running time (TOT)<sup>6</sup> of each JIT configuration with the translation overhead (TR); TR is part of TOT. The table shows that the TOT with LaTTe’s JIT is about half of the TOT with Kaffe’s JIT. As for the translation overhead, LaTTe’s TR is three times larger than Kaffe’s TR on average, yet both TRs take a tiny portion of the TOTs.

We also checked the relationship between the translation overhead and the translated bytecode size. Figure 2.11 depicts for each benchmark the TR of both JIT compilers and the translated bytecode size shown in Table 2.1. We can see that LaTTe’s TR grows much faster than Kaffe’s since LaTTe requires more compilation passes

<sup>6</sup>TOT means the total elapsed time, which is not comparable with a SPECjvm98 metric.

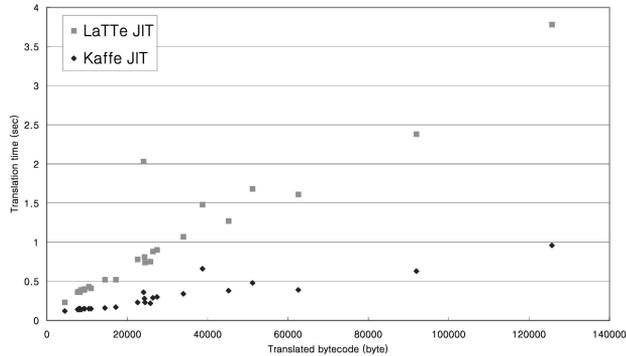


Figure 2.11: Translation overheads and translated bytes of LaTTe JIT and Kaffe JIT.

with elaborate analysis. However, LaTTe’s TR still increases almost linearly<sup>7</sup> to the translated amount of bytecode, as Kaffe’s TR does.

The results in this section indicate that LaTTe’s sophisticated register mapping and allocation really pay off without causing a big translation overhead. In order to complete the evaluation, however, it would be desirable to compare with register-mapping JIT compilers such as CACAO or VTune. Unfortunately, it would be extremely difficult to implement and tune them completely on the same framework and to make a fair comparison.

### 2.6.3 Speed and Quality of LaTTe’s Register Mapping and Allocation

Although LaTTe’s JIT compilation overhead is higher than that of Kaffe’s, Table 2.1 indicates that LaTTe’s translation overhead is reasonable since it takes a tiny portion of the total running time; for all benchmarks except for `javacc`, TR consistently takes only one or two seconds when TOT takes several tens of seconds. Since register mapping and allocation is the main contributor to TR in the base LaTTe (our experiments show that it takes 67 percent of TR, on average), this means LaTTe’s register mapping

<sup>7</sup>In fact, all phases in the LaTTe JIT compilation are linear in the bytecode size except for the register allocation phase. The backward sweep and the forward sweep are linear, but the reconciliation at join points is quadratic, however, we found in practice that the reconciliation time is negligible in most cases.

and allocation is reasonably fast.

In order to examine how LaTTe allocates registers, we measured for each method the “peak” number of real registers (including spill locations) used during its register allocation. This is measured by tracing the number of live real registers mapped to some symbolic registers in the `h_map` during the forward sweep. Comparing this number with worst-case register requirements or minimum requirements when pre-allocating local variables will be helpful in evaluating LaTTe.

Table 2.2: Register mapping and allocation quality of the base LaTTe for top five frequent methods.

Top Five Frequent	1st method				2nd method				3rd method				4th method				5th method				
	M	L	S	T	M	L	S	T	M	L	S	T	M	L	S	T	M	L	S	T	
_200_check	< 8	7	3	1	< 16	13	5	0	< 4	2	2	1	< 12	10	5	0	< 6	3	5	0	
_201_compress	< 11	10	6	0	< 10	7	5	0	< 10	5	6	1	< 4	1	5	0	< 8	5	4	1	
_202_jess	< 18	16	5	0	< 7	4	3	1	< 4	3	4	0	< 6	4	3	1	< 3	2	2	1	
_209_db	< 12	11	2	1	< 14	9	3	3	< 6	2	5	0	< 6	4	5	0	< 10	4	3	3	
_213_javac	< 7	4	3	0	< 8	7	3	1	< 5	1	5	0	< 11	8	3	1	< 8	4	5	0	
_222_mpegaudio	< 15	13	4	1	# 20	31	5	1	< 10	7	5	0	< 9	6	4	0	*	8	0	4	6
_227_mtrt	< 13	9	4	1	< 6	4	3	1	< 7	5	4	0	< 1	1	1	0	< 3	2	2	1	
_228_jack	< 4	1	4	0	< 16	13	5	0	< 18	14	6	1	< 8	7	3	1	< 11	9	5	0	
Series	< 17	15	8	0	< 6	6	6	0	< 13	4	12	0	< 8	7	3	1	< 11	8	3	3	
LUFact	< 18	13	9	0	< 15	10	6	1	< 15	9	8	1	< 18	15	10	0	# 12	12	3	1	
SOR	< 22	18	8	1	< 8	4	6	0	< 5	3	5	0	< 11	6	6	0	< 8	7	3	1	
HeapSort	< 8	5	4	1	< 9	5	4	1	< 8	4	6	0	< 7	3	4	1	< 6	3	3	1	
Crypt	< 18	15	4	1	< 7	3	4	1	< 7	4	3	1	< 8	7	3	1	< 11	8	3	3	
FFT	< 39	35	6	0	< 15	11	4	1	< 8	4	6	0	< 11	7	6	0	< 5	3	5	0	
Sparse	< 16	9	8	1	< 8	4	6	0	< 7	2	5	1	< 5	3	5	0	< 1	1	1	0	
Search	< 7	6	3	0	< 20	20	5	0	< 10	5	5	1	< 8	5	4	1	< 8	5	5	0	
Euler	< 17	9	10	0	< 26	19	8	1	< 12	5	9	1	< 23	14	11	0	< 23	14	11	0	
MD	* 55	52	6	0	< 8	1	8	0	< 7	3	5	0	< 9	5	6	0	< 11	9	6	0	
MC	< 8	4	6	0	< 10	2	9	0	< 12	9	6	0	< 7	2	7	0	< 19	15	8	0	
RayTracer	< 8	2	6	0	< 15	9	7	0	< 7	3	5	0	< 11	7	4	0	< 7	3	6	0	
richards.g	< 4	2	0	2	< 5	3	3	0	< 6	5	3	1	< 5	2	3	1	< 7	3	5	0	
richards.gf	< 4	2	0	2	< 5	3	3	0	< 6	5	3	1	< 5	2	3	1	< 7	3	5	0	
richards.gns	< 2	1	0	1	< 3	2	4	0	< 6	5	3	1	< 5	3	3	0	< 5	2	3	1	
richards.dna	< 2	1	1	0	< 3	2	3	0	< 2	1	0	1	# 6	7	3	1	< 2	1	1	0	
richards.dav	< 2	1	1	0	< 3	2	3	0	< 2	1	0	1	# 6	7	3	0	< 2	1	1	0	
richards.daf	< 2	1	1	0	< 3	2	3	0	< 2	1	0	1	# 6	7	1	2	< 2	1	1	0	
richards.dai	< 2	1	1	0	< 3	2	3	0	< 2	1	0	1	# 6	7	3	0	< 2	1	1	0	
richards.all	< 4	2	0	2	< 4	2	0	2	< 2	1	0	1	< 3	2	4	0	< 2	1	1	0	
spell	< 12	11	2	1	< 16	13	5	0	< 8	7	3	1	< 6	3	5	0	< 7	4	3	3	
javacc	< 8	7	2	0	< 11	8	3	1	< 8	7	3	1	< 6	2	5	0	# 8	6	0	4	
delta	< 6	2	5	0	< 3	3	2	0	< 3	1	1	1	< 3	1	1	1	< 3	2	2	0	
jjtre	< 16	13	5	0	< 6	5	6	0	< 7	4	5	0	< 4	3	3	0	< 11	8	5	0	
jlex	< 10	10	3	0	< 6	2	5	0	# 7	4	0	4	< 8	6	2	0	< 8	3	3	3	
hash	< 8	7	3	1	< 10	7	6	0	< 11	8	3	1	< 5	1	5	0	< 12	11	2	1	

For the base LaTTe, Table 2.2 shows the peak number (denoted by M) for the top five methods with the highest bytecode execution counts in each benchmark (which comprises 57 percent of the total bytecode execution counts on average), along with the number of local variables (denoted by L). The table also includes the number of stack entries used (denoted by S) and the number of temporary registers used (denoted

by  $T$ ), at some point in the method where  $S + T$  is maximum. The sum  $L + S + T$  thus is the worst-case register requirement. If local variables are preallocated as in CACAO,  $L + T$  is the minimum number of real registers required (for VTune, some non-overlapping local variables can be allocated to the same register via limited live range analysis).

We can find from the table that  $M$  is smaller than  $L + S + T$  in many cases (marked by  $j$ ). For some methods,  $M$  is even smaller than  $L + T$  (marked by  $\#$ ) or even than  $L$  itself (`_222_mpegaudio` and four `richards` benchmarks). This is possible because LaTTe can coalesce copies between local variables generated by the `xload-xstore` bytecode sequences, and can allocate the same register into non-overlapping local variables through its conservative live variable analysis. This flexibility is due to LaTTe’s aggressive register mapping with pseudocode generation as well as LaTTe’s efficient register allocation, which obviates preallocating local variables as in CACAO or VTune.

In this table, we can also find there are only two methods that spill (marked by  $*$ ). Since spills are related to the register pressure of the translated code as well as to the quality of register allocation, we also need to check those cases where register pressure would be higher.

We examined the top five methods with the largest number of local variables as shown in Table 2.3. Although the register pressure is much higher, we see spills only in five methods.<sup>8</sup> (In this table,  $M$  is still smaller than  $L + S + T$  and smaller than  $L + T$  or  $L$  in even more methods).

We have also measured the same data for the optimized version of LaTTe for the same methods in Table 2.2 and Table 2.3 where the register pressure is higher due to inlining and other optimizations. In particular,  $L$  tends to be increased due to inlining. Also, there are many cases when  $S$  is reduced while  $T$  is increased. This is due to CSE

---

<sup>8</sup>The first method in many benchmarks ( $M = 35$ ,  $L = 37$ ,  $S = 7$ ,  $T = 1$ ) that causes the spill is the same one in the JDK class library called `toDouble()` which converts double numbers into strings.

Table 2.3: Register allocation quality of the base LaTTe for top five largest-locals methods.

Top Five Largest-Local	1st method				2nd method				3rd method				4th method				5th method								
	M	L	S	T	M	L	S	T	M	L	S	T	M	L	S	T	M	L	S	T					
..200.check	*35	37	7	1	22	14	8	0	<	16	13	5	0	<	13	13	13	0	<	16	13	3	1		
..201.compress	*35	37	7	1	22	14	8	0	<	16	13	5	0	<	13	13	13	0	<	16	13	3	1		
..202.jess	*35	37	7	1	*	21	16	10	1	<	18	16	5	0	#	14	15	5	0	22	14	8	0		
..209.db	*35	37	7	1	22	14	8	0	<	16	13	5	0	<	13	13	13	0	<	16	13	3	1		
..213.javac	*35	37	7	1	<	23	22	8	0	<	26	22	10	1	<	24	22	4	1	<	23	21	8	0	
..222.mpegaudio	22	14	8	0	<	16	14	3	0	<	16	13	5	0	<	13	13	13	0	<	16	13	3	1	
..227.mtrt	*35	37	7	1	<	27	23	7	0	<	23	21	7	0	<	21	19	8	1	<	21	16	8	1	
..228.jack	*35	37	7	1	22	14	8	0	#	12	14	7	1	<	18	14	6	1	<	16	13	5	0		
Series	*35	37	7	1	<	17	15	8	0	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	
LUFact	*35	37	7	1	<	18	15	10	0	<	17	14	12	0	<	16	13	5	0	<	18	13	9	0	
SOR	*35	37	7	1	<	22	18	8	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	
HeapSort	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
Crypt	*35	37	7	1	<	18	15	4	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	
FFT	*35	37	7	1	<	39	35	6	0	<	16	13	5	0	<	15	11	4	1	<	13	10	9	0	
Sparse	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	16	9	8	1	
Search	*35	37	7	1	<	20	20	5	0	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	
Euler	*35	37	7	1	<	38	33	13	0	<	32	21	18	0	<	26	19	8	1	<	20	16	12	1	
MD	*55	52	6	0	*	35	37	7	1	*	31	30	4	1	<	19	19	3	0	<	16	13	5	0	
MC	*35	37	7	1	<	19	15	4	1	<	19	15	8	0	#	14	15	5	0	<	16	13	5	0	
RayTracer	*35	37	7	1	<	24	21	7	0	<	20	19	6	0	<	25	17	12	0	#	14	16	10	0	
richards_g	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
richards_gf	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
richards_gns	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
richards_dna	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
richards_dav	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
richards_daf	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
richards_dai	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
richards_all	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3	
spell	<	16	13	5	0	<	12	11	2	1	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
javacc	#	14	21	7	0	*	22	20	6	1	#	21	19	3	3	<	18	16	4	1	<	19	16	3	1
delta	*35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	13	10	5	1	
jjtre	#	8	20	3	1	#	15	18	3	1	#	6	15	3	1	#	12	14	5	0	<	16	13	5	0
jlex	18	14	4	0	<	16	13	5	0	<	14	12	3	1	<	14	11	5	0	<	13	10	9	0	
hash	#	27	25	3	3	<	22	21	5	1	<	17	15	4	1	<	16	14	3	0	<	20	13	17	1

which replaces many stack variables by temporary variables. We found that even with this higher register pressure, LaTTe rarely spills registers.

These results indicate that even with LaTTe's aggressive mapping of registers and copy coalescing, the register pressure of the translated code would rarely be too high, which makes LaTTe's fast, region-based register allocation with local lookahead effective enough to avoid spills.<sup>9</sup>

In conclusion, LaTTe generates efficient code via aggressive register mapping and efficient register allocation. On the other hand, it is unlikely for a JIT compiler that can generate code as efficient as LaTTe's to be much faster than LaTTe since LaTTe's JIT compilation overhead is already small enough. Therefore, we believe LaTTe made a reasonable trade-off between speed and quality of JIT compilation.

---

<sup>9</sup>Actually, even some of those spills are due to the SPARC calling conventions, not due to high register pressure.

## Chapter 3

# Java Virtual Machine Optimization for Dynamic Languages

Over the past two decades, the performance of the Java virtual machine (JVM) has been matured enough, allowing Java to run less than twice slower than C/C++ on some benchmarks [1]. This is mainly due to its just-in-time (JIT) compiler that translates the Java bytecode to machine code at runtime, such as the HotSpot JIT compilers in JDK 9.

High-performance along with its rich set of APIs has recently encouraged the JVM to host many non-Java languages such as JavaScript, Ruby, Python, and Scala [2]. This is made possible by implementing the language runtime in Java (e.g., Nashorn JavaScript engine [3], Jython engine [4], or JRuby engine [5]) and by extending the JVM specification [6, 7] or augmenting the script APIs [8] to handle non-Java language features like dynamic typing or first-class function.

Efficient execution of the JVM-hosted languages is challenging since there are complicated interactions between the language runtimes, the JVM and its extensions. The runtime behaviour is also different from that of regular Java in terms of memory usage and execution characteristics [9]. Although there are some promising initial reports on the performance of the JVM-based language environments [1], we believe

that there is still more room for performance improvement. Specifically, we found that the JVM encounters much more branch bytecodes than when executing Java, suffering from more complex control flows. We also found that many of these branches are highly biased such that the infrequent paths of these branches are rarely taken.

This paper proposes a simple and novel technique for the JVM JIT compiler optimization named exceptionization to exploit these branch behaviours in a speculative way. Exceptionization transforms a highly-biased branch bytecode into an implicit exception-throwing instruction during JIT compilation, which allows the JIT compiler to prune the infrequent target of the branch from the method control flow. The exclusion of infrequent paths generates a simpler control flow graph, allowing better code generation with more precise data flow information and reduced JIT compilation overhead. If a pruned path was taken, the JVM executes the path in a lazy way similar to the Java exception handling proposed in LaTTe [10]. We also devised de-exceptionization to cope with the case when a pruned path is actually executed more often than expected, by recompiling the whole method or the pruned path.

We implemented exceptionization in HotSpot<sup>TM</sup>JVM of the OpenJDK 9 alpha release [11]. We experimented with JavaScript, Ruby, and Python as the JVM-hosted language to evaluate the performance benefit of exceptionization, because there exist publicly available language runtimes on top of the OpenJDK package including Nashorn, an official JavaScript engine from Oracle [12].

Three contributions can be found in this paper. First, we analyzed the behaviour of branch bytecode when JVM runs a non-Java language and found that it is very frequent and highly biased. Secondly, we presented exceptionization to exploit this branch behaviour to improve the overall performance. Finally, we empirically showed that efficient execution of the language runtime itself by the JVM is as important as efficient translation and optimization of the non-Java language by the language runtime, although so far the latter has been the major issue for this research area [13, 14, 15].

The rest of Chapter 3 is organized as follows: Section 3.1 describes the multi-

language support of JVM and the architectures of non-Java language runtimes in OpenJDK. Section 3.2 analyzes branch behaviours of several JVM-hosted languages with comparison to Java, which is the key motivation of our research. We explain the main idea and issues of exceptionization in Section 3.3, followed by the experimental evaluation in Section 3.4. Section 3.5 discusses related achievements that inspired our research.

## **3.1 Non-Java Languages on JVM**

This section describes the JVM supports for non-Java languages and shows how non-Java languages including JavaScript are executed on top of JVM.

### **3.1.1 JVM Extensions for Multi-Languages**

To host non-Java languages on JVM, the JVM specification [6] has recently been augmented with two extensions, JSR 223: scripting for the Java platform and JSR 292: supporting dynamically typed languages on the Java platform.

JSR 223, incorporated into Java SE 6, defines the interoperability APIs between Java and script-style languages such as JavaScript, Ruby, and Python [8]. Java and other languages can share JVM as the common runtime environment, encouraging JVM-based language implementations.

Absorbing JSR 292, Java SE 7 provided a new bytecode instruction, `invokedynamic`, to allow faster function invocation with dynamically-typed objects used in modern programming languages [16, 7]. Since JVM has been designed originally for Java, a statically-typed language, existing call bytecode instructions such as `invokevirtual`, `invokestatic`, and `invokeinterface` assume a class hierarchy chain to find the actual callee for a method invocation. Thus, a dynamic language runtime on JVM should construct an implicit class hierarchy with all possible target objects of any dynamic calls, in order to support dynamic typing only with those instructions.

invokedynamic has no restriction on the class hierarchy for a dynamic call and allows the actual callee to be cached right after the invocation, which becomes an invaluable asset to implement JIT compilation efficiently for dynamically-typed languages on JVM. The instruction has been implemented in the bytecode level from Java SE 8, so the invocation stub named LambdaForm [17] is created on the fly and executed per invokedynamic by JVM.

As a result, many JVM-based language implementations have emerged such as JRuby for Ruby [5], Jython for Python [4], Clojure [18], and Scala [19].

### **3.1.2 Nashorn JavaScript Engine on HotSpot JVM**

We adopted OpenJDK 9 alpha-release, the reference implementation to upcoming Java SE 9, as the base research environment [11]. Figure 3.1 illustrates the architecture of the OpenJDK package supporting JavaScript on JVM.

HotSpot™ in the OpenJDK package is a highly-engineered product-level JVM implementation from Oracle, armed with a three-tiered dynamic compilation system to maximize performance [20]. The first-tier execution engine is an assembly interpreter with profiling. If a method is found sufficiently warm, then the second-tier linear-scan JIT compiler (C1 compiler) translates the bytecode method into machine code. The third-tier JIT compiler (C2 compiler) generates high-quality code for hot methods through aggressive optimizations including speculative type specialization, method inlining, escape analysis, graph-coloring register allocation, and instruction scheduling with help of the profile information from the interpreter and the C1-compiled methods.

Since Java SE 8, OpenJDK is accompanied by a JavaScript engine called Oracle Nashorn™ [3], compliant with the ECMA normalized JavaScript specification [21]. Nashorn™ deploys its own dynamic compilation system exploiting JSR 292 features from the underlying HotSpot VM and supports JSR 223 for interoperability with Java. It compiles JavaScript directly into bytecode with type specialization or generates

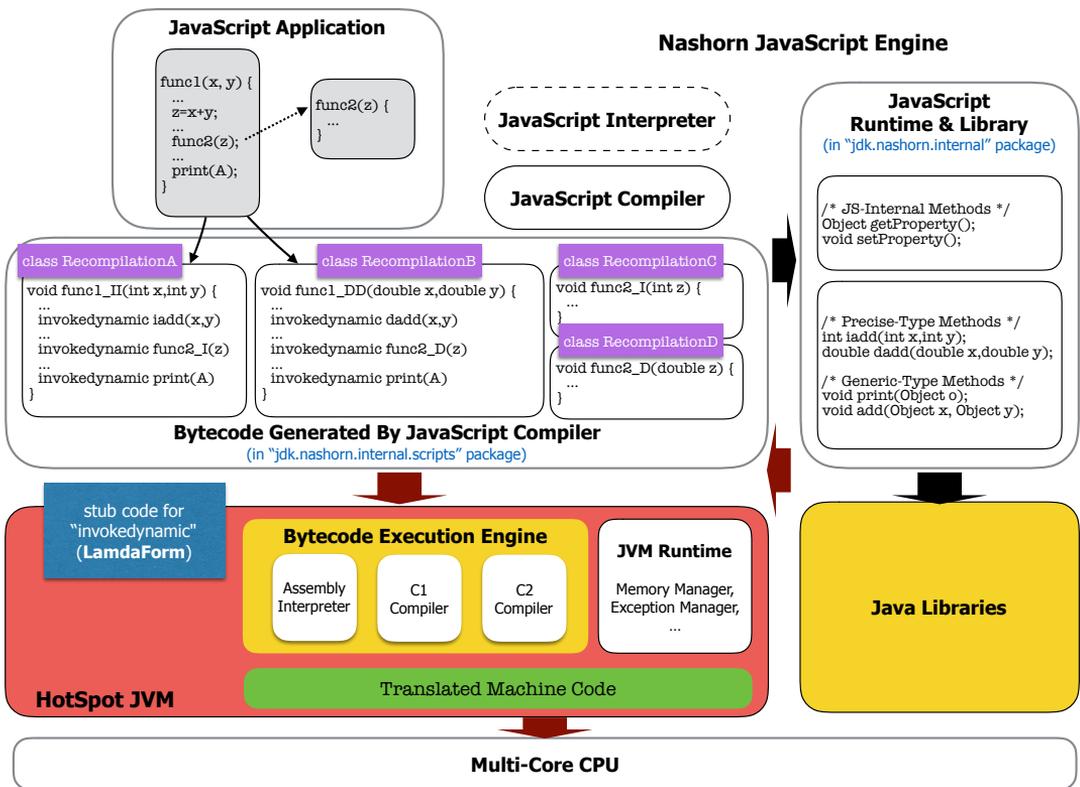


Figure 3.1: Nashorn on top of HotSpot in OpenJDK.

bytecode selectively after interpretation with profiling.<sup>1</sup> Nashorn utilizes a generic AST representation and can be morphed into a front-end for other programming languages on HotSpot.

The JavaScript application in Figure 3.1 contains two functions: `func1` and `func2`. `func1` performs an addition of two variables, invokes `func2` with the result, and prints the value of `A`. The functions and variables are dynamically-typed, so that executing them on the statically-typed JVM requires a kind of translation or type specialization by Nashorn.

Nashorn applies the optimistic typing technique [13] to a generic JavaScript function through the variable type speculation, by generating multiple bytecode methods with possible type combinations per function. Thus, `func1` is translated into multiple methods including `func1_II` with integer variables and `func1_DD` with double variables. If the optimistic typing is not applicable or a conservative translation is necessary, then any dynamically-typed object will be translated into `java.lang.Object` like the case of the JavaScript function call to `print()`. Each generated method resides in an implicit class of the `jdk.nashorn.internal.scripts` package.

The JavaScript runtime along with the supplementary library provides the primitive runtime functionalities, and operates as a glue layer between JavaScript functions and the underlying Java APIs. For example, it provides addition methods such as `iadd()`, `dadd()`, and `add()` to execute `x+y` in JavaScript. `print()` in the JavaScript library will invoke an appropriate Java `print()` method according to the parameter type. The JavaScript compiler utilizes internal method calls to `getProperty()` or `setProperty()` for the access to an object field (property) of JavaScript, so that a generated bytecode method contains many method calls.

The implicit classes for JavaScript functions, the JavaScript runtime, and the library require no solid class hierarchy among them, because function calls in the JavaScript

---

<sup>1</sup>Although the bleeding-edge Nashorn in OpenJDK 9 has experimentally begun to use interpretation in order to reduce the warm-up time, Nashorn compiles JavaScript code without interpretation by default.

application and internally-used calls to runtime methods are translated with invokedynamic. HotSpot should take care of the translated JavaScript code, the Nashorn execution engine, the LambdaForm stub for invokedynamic, and Java libraries simultaneously to run a JavaScript application. And there exist two layers of JIT compilers in Figure 3.1: one from JavaScript to JVM bytecode and the other from bytecode to machine code.

In its warm-up phase, Nashorn may interpret or compile the given JavaScript code, so that the JavaScript execution engine part grows warm to be C1-compiled and then starts to be hot enough for the C2 compilation. As Nashorn moves to its saturation phase, the JavaScript code gets translated into bytecode by the JavaScript JIT compiler while the JavaScript runtime and library modules in Nashorn get frequently invoked to execute the code, so that the different parts of the system become identified as hot.

Therefore, the phase transition from warm-up to saturation in Nashorn is gradually propagated to HotSpot and the hot JavaScript functions in the application identified by Nashorn need some time to be also recognized as hot by HotSpot<sup>TM</sup>, leading to more complex JVM-level phase transitions, which can be named as gradual propagation.

### **3.1.3 Other Non-Java Languages on HotSpot JVM**

In this paper, we will also discuss other JVM-hosted languages such as Ruby and Python. JRuby 9.0.5.0 [5] is the selected implementation of Ruby 2.2 atop JVM. It is equipped with a JIT compiler that generates bytecode through a level of conservative static optimizations, and supports both JSR 223 and JSR 292. Jython 2.7 [4] is a Python 2.7 implementation atop JVM. It deploys a simple JIT compiler with support of JSR 223. invokedynamic is not utilized by Jython.

## 3.2 Branch Bytecode Behaviours Of Non-Java Languages

Several researches discovered that runtime behavioral characteristics of non-Java languages on JVM differ from those of Java [9, 22]. There are two factors for such differences: language grammar and implementation.

A non-Java language has its own grammar in a new programming style, such as functional programming and script programming, introducing exotic runtime patterns. For example, JavaScript is a dynamically-typed script language which leads to the heavy usage of generic functions with multi-type or generic-type parameters.

The multi-threaded and two-layered language implementation on top of the multi-core CPU architecture as shown in Figure 3.1 usually combines the JIT compilers with the interpreters in an arbitrary manner and changes the execution sequence and modes of JavaScript functions and bytecode methods at every run, leading to the unpredictable performance fluctuation, not observable in traditional Java runs.

### 3.2.1 Branch Bytecode Statistics

We analyzed the branch bytecode behaviours of Java and three non-Java dynamic languages including JavaScript, Ruby, and Python, in order to evaluate the control flow complexity in JVM level by measuring branch bytecodes translated by the HotSpot C2 compiler, i.e. branch instructions in hot methods. The CLBG benchmark suite [1] was chosen for the analysis, because it provides the sets of identical micro-benchmarks in multiple languages for language performance comparison. Two standard benchmark suites for JavaScript (Octane 2.0 [23]) and Java (SPECjvm2008 [24]), were additionally used to capture the real-world program behaviours.

In addition to the execution time of each benchmark suite, we instrumented the number of C2 compilations, the number of compiled branch bytecodes, and the execution frequency (executions per second) of branch bytecodes in C2-compiled methods.

<sup>2</sup> Table 3.1 presents statistics of C2 compilation and branch bytecodes on CLBG for

---

<sup>2</sup>We used `binarytrees`, `fannkuchredux`, `fasta`, `mandelbrot`, `nbody`, and `spectralnorm` among the micro-

Table 3.1: C2 compilation and branch bytecodes on CLBG

	Java	JavaScript	Ruby	Python
Execution Time	39.58s	155.41s (3.9x)	735.15s (18.6x)	1568.96s (39.6x)
# of C2 Compilations	38	1370 (36.1x)	2715 (71.5x)	1935 (50.9x)
# of C2 Comp'ed Branches	195	15566 (79.8x)	27572 (141.4x)	25618 (131.4x)
C2 Comp. / min	57.61	528.94 (9.2x)	221.59 (3.9x)	74.00 (1.3x)
Branches / C2 Comp.	5.13	11.36 (2.2x)	10.16 (2.0x)	13.24 (2.6x)

Table 3.2: C2 compilation and branch bytecodes on Octane and SPECjvm2008

	JavaScript (Octane)	Java (SPECjvm2008)	Ratio (JavaScript/Java)
Execution Time	2m 51s	123m 23s	0.02
# of C2 Compilations	5844	4410	1.33
# of C2 Comp'ed Branches	353324	94952	3.72
C2 Comp. / min	2050.5	35.7	57.44
Branches / C2 Comp.	60.46	21.53	2.81

the four languages and Table 3.2 on Octane and SPECjvm2008.

The execution time, the number of C2 compilations, and the C2 compilation frequency in Table 3.1 allow a brief evaluation of the language runtime implementations. JavaScript seems to have the most powerful and sophisticated implementation while Python has still room to improve. The better language runtime leads to the bigger C2 compilation frequency.

Figure 3.2 and Figure 3.3 illustrates the distribution histogram of branch bytecode execution frequency with regard to branch bias, the execution probability of the infrequent target of a branch in the range of  $[0.0, 0.5]$  for Java, JavaScript, Ruby, and Python on the micro-benchmarks (CLBG).<sup>3</sup> Figure 3.4 also depicts the distribution histogram benchmarks in CLBG for the four languages on JVM. The bytecode branch execution frequency is the mean value of those of the six benchmarks, while others are the summations of the benchmarks.

<sup>3</sup>Since both the interpreter and the C1-compiled methods profile branch results, the C2 compiler can calculate a bias for each branch bytecode.

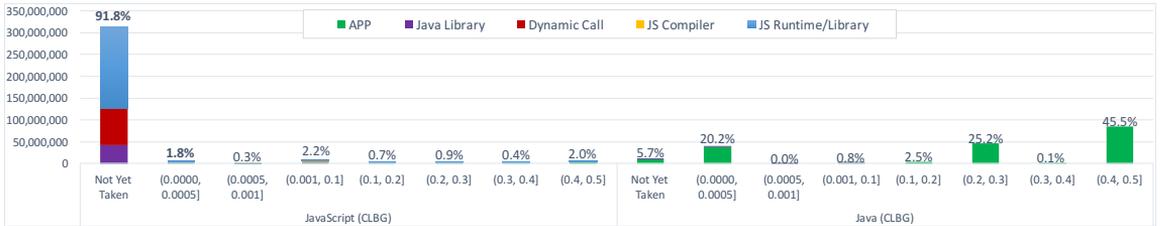


Figure 3.2: Bias distribution of branch bytecode execution frequency in C2-compiled methods (CLBG).

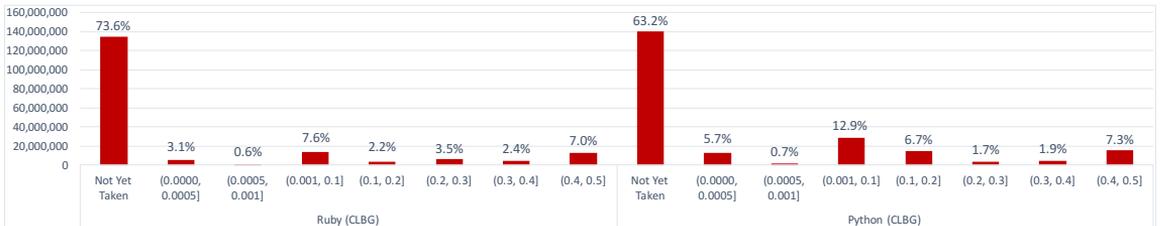


Figure 3.3: Bias distribution of branch bytecode execution frequency in C2-compiled methods (CLBG).

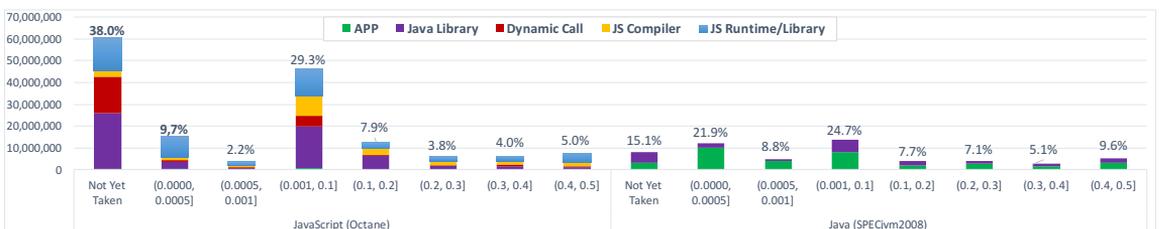


Figure 3.4: Bias distribution of branch bytecode execution frequency in C2-compiled methods.

on the macro-benchmarks for JavaScript and Java (Octane and SPECjvm2008). The number above each bar means the relative ratio of each bias range.

Figure 3.2 and Figure 3.4 present the origins of branch bytecodes with five categories. APP means that a branch appears in the Java or JavaScript application code, and a branch in the standard Java libraries is counted as Java Library. JS Runtime/Library and JS Compiler correspond to branches in the Nashorn engine. Dynamic Call counts branches in the invocation stubs of LambdaForm to implement invokedynamic.

### 3.2.2 Behavioral Characteristics of Branch Bytecodes

The measurement reveals several behavioral characteristics of branches in non-Java dynamic languages as follows.

- A dynamic language generates much more complex control flows than Java. : Table 3.1 and Table 3.2 reveal that HotSpot encounters at least twice more branch bytecodes on average during the C2 compilation for dynamic languages than for Java. Similarly, the execution frequencies of branch bytecodes in dynamic languages are higher than in Java. Therefore, it is obvious that the dynamic language execution on JVM involves more complex control flows than Java.
- The language runtime itself is the main source of the complex control flow. : The dynamic typing of non-Java languages has been identified as the key reason of complex control flows with many branches [14, 15, 25]. As illustrated in Figure 3.1, dynamically-typed JavaScript objects and variables are manipulated via the optimistic typing in JS Compiler, the JavaScript internal methods and the generic-type methods in JS Runtime/Library, and the LambdaForm stubs in Dynamic Call by the Nashorn engine.

Figure 3.2 supports this knowledge, because JS Compiler, JS Runtime/Library, and Dynamic Call take 82% of executed branch bytecodes, contributing to the

excessive branch execution of JavaScript. Figure 3.4 shows the similar characteristic for the bigger benchmarks. We can conclude that JVM keeps to encounter many branches when running non-Java languages in spite of various optimizations performed by modern language runtimes that may eliminate branches in APP, because the language runtime implementations introduce their own complex control flows.

- A dynamic language contains more highly-biased branches than Java. : For CLBG, 96% of branch bytecodes are biased above 1:9 in JavaScript, 85% in Ruby, and 83% in Python while just 27% in Java as shown in Figure 3.2 and Figure 3.3. Figure 3.4 also shows that the larger portion (79% vs. 71%) of branches are biased above 1:9 in JavaScript in case of the bigger benchmarks.

Furthermore, the bias range,  $[0.0,0.1]$  presents the significant difference between Java and non-Java languages. About 92% of branches take only one target for CLBG and 38% for Octane in JavaScript, while just 6% and 15% of branches are single-sided for CLBG and SPECjvm2008 in Java. 48% of branches are biased above 5:9995 on Octane while just 37% on SPECjvm2008.

Coupled with the first observation, this implies that the JVM encounters many highly-biased branches to execute non-Java languages. If these highly-biased branches are exploited wisely, then the system performance may be improved.

- A non-Java language, specifically JavaScript, shows more unstable behaviours than Java. : Table 3.1 presents that executing an identical workload would require more C2 compilations if the workload is written in non-Java languages, due to the presence of the language runtimes. Similarly, Table 3.2 indicates that the C2 compilation frequency of Octane is 57 times higher than that of SPECjvm2008. More bytecode methods need to be compiled for JavaScript execution because of the sophisticated Nashorn engine as well as the application code. Additionally, the optimistic typing generates multiple methods for a

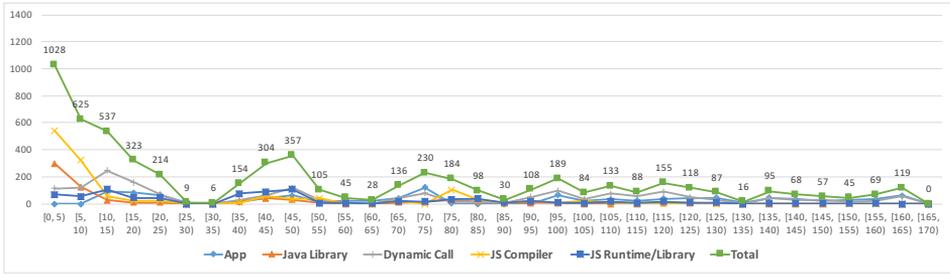


Figure 3.5: C2 compilations over time (Octane).

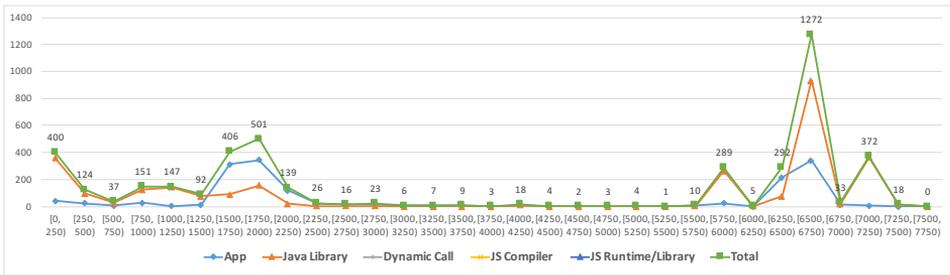


Figure 3.6: C2 compilations over time (SPECjvm2008).

JavaScript function.<sup>4</sup>

Figure 3.5 and Figure 3.6 depict C2 compilation counts over time for Octane and SPECjvm2008. The size of time slot has been normalized to each other for lifetime comparison.

The life of Octane can be divided into three phases in Figure 3.5, which confirms to the gradual propagation explained in Section 3.1. The first warm-up phase to 20 seconds prepares the basic environment for JavaScript code execution. JS Compiler takes the major portion of hot spots because the JavaScript code should be compiled into bytecode for execution. In the second phase from 40 seconds to 85 seconds, APP is translated from bytecode to machine code frequently as

<sup>4</sup>The workload of SPECjvm2008 may be so heavier than that of Octane as to take more execution time, lowering average compilation frequency. However, we claim that it does not fully explain such large difference.

well as JS Compiler and JS Runtime/Library. This phase can be considered as the actual warm-up phase of the JavaScript application, because the Nashorn engine becomes specialized for the application and the hot JavaScript methods get translated by the C2 compiler directly. After 85 seconds, the system migrates into the saturated phase, even though the C2 compilation continues to be not uncommon at all due to Dynamic Call and APP, which are necessary to execute hot JavaScript functions.

Figure 3.6 shows that the Java run consists of the standard warm-up and saturated phases with additional two bursts. The first burst from 1250 seconds to 2250 seconds comes from the intentional benchmark execution to make JVM and the benchmark suite saturated before performance measurement. The other burst from 6000 seconds to 7000 seconds are needed to compile the SPEC report framework including AWT and the graphics system.

Consequently, the gradual propagation lets JavaScript change its hot method sets more frequently than Java. The frequent compilation of JS Compiler and JS Runtime/Library in the second phase means that the JavaScript runtime states have not been stable by that time, so any compiler assumptions in the first phase may become invalid. We use the term, *stable*, to specify whether an instruction shows the same as its profiled behaviour after JIT compilation or not. If a branch behaves consistently with its profiled behaviour, then it is called *stable*. This notion of stability is the basic assumption of many profile-driven optimizations in the JIT compiler. In context of our research, this complex JVM state transition introduces unstable branches that will be discussed later.

## 3.3 Exceptionization

### 3.3.1 Idea of *Exceptionization*

Java exception is a language construct to handle erroneous events, which do not happen normally, in a structured way [26]. A method invocation or an operation may raise an exception if something goes wrong. On exception, JVM searches the appropriate exception handler provided as a catch block and transfers control to it.

Java exception handlers in a method are usually ignored during JIT compilation, because exceptions are rarely thrown so that their exception handlers will not be reached in most cases. If an exception is thrown in a compiled method, then the corresponding catch block may be interpreted through deoptimization [27] or compiled on demand to run as native [10].

Since a non-Java language generates quite a few highly-biased branches, many branch targets will not be taken much at all, as we found previously. Exceptionization transforms a highly-biased branch into an implicit exception-throwing instruction. A branch exception is thrown if a highly-biased branch is about to jump to the infrequent target, and the infrequent path gets treated as the handler to the branch exception.

Figure 3.7 illustrates exceptionization on a code segment that is the simplified version of a hot method in Nashorn. The method in (a) sets  $x$  and  $y$  by checking information of two input Nodes:  $n1$  and  $n2$ . If both values of  $x$  and  $y$  are 1, then the method returns true. Otherwise, it returns false. In most cases, the method, `checkInfo()`, is found to return true so as to set both  $x$  and  $y$  to 1. Bytecode for the method is given in (b) as a control flow graph with yellow blocks indicating the frequent path.

Normally, the C2 compiler translates the method into the machine code in (c) with the same control flow as (b). However, exceptionization lets the compiler generate the optimized code in (d), in streamline with the aggressive register allocation and other optimizations. The frequent path contains 3 basic blocks with 13 instructions in (d) while 8 basic blocks with 19 instructions in (c). It is notable that the conditional

```

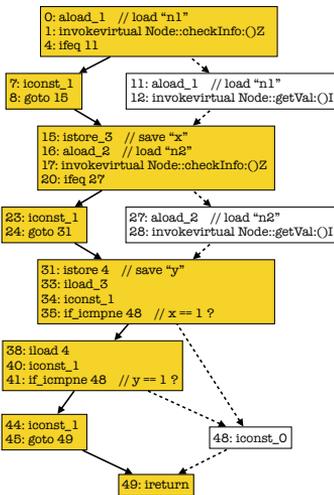
public boolean checkNode(final Node n1, final Node n2) {
    // mostly, checkInfo() returns "true".
    int x = (n1.checkInfo()) ? 1 : n1.getVal();
    int y = (n2.checkInfo()) ? 1 : n2.getVal();

    return (x==1 && y==1);
}

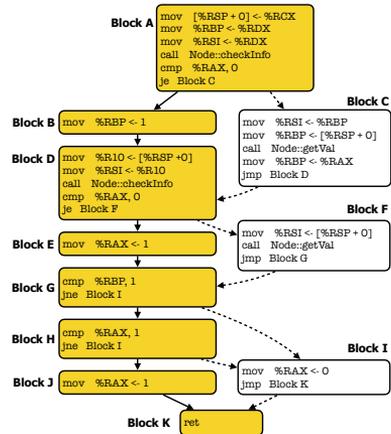
```

(a) Java code.

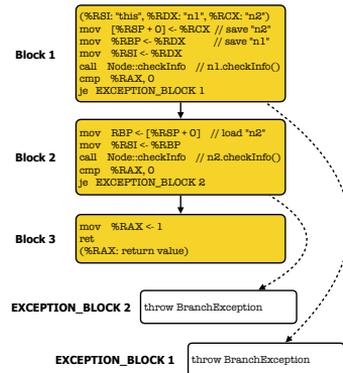
\* Yellow blocks compose the frequent path.



(b) Control flow graph of bytecode.



(c) Machine code with no exceptionization.



(d) Exceptionalized machine code.

Figure 3.7: Exceptionization on a sample code segment.

branches from return ( $x==1\&\&y==1$ ) are fully eliminated by exceptionization in (d) because the compiler knows that the two variables are 1 on the frequent path returning true by removing the two join points in front of Block D and Block G in (c). Generally, exceptionization removes joins in the control flow of a method, producing larger extended basic blocks (EBBs) [28]. EBBs widen the flow analysis scope for the compiler to generate better-quality code, with effectively negating the complicated control flows of JavaScript. We can view exceptionization as another type of the control merge splitting [29] or tail duplication [30] with no actual basic block cloning. The actual basic block cloning will happen during de-exceptionization to be explained.

Additionally, the simplified control flow reduces the compilation resource usage including the JIT compilation time and the code cache translation. The JIT compiler may re-invest saved resources for deeper inlining and more compilations, which are both beneficial to the non-Java language performance [22].

### 3.3.2 Selection of *Exceptionization* Targets

We chose to integrate exceptionization only with the HotSpot C2 compiler. The C2 compiler is supplied with so sufficient branch profile information from the interpreter and the C1-compiled methods, that the branch behaviour speculation used by exceptionization becomes practically precise. Moreover, aggressive optimizations in the C2 compilation can maximize the benefits from exceptionization.

Figure 3.8 presents the pseudo code for exceptionization. For a branch instruction, the C2 compiler calculates the branch bias based on the gathered branch profile in the IR generation phase. If the bias is below a preset `EXCEPTIONIZATION_THRESHOLD` and the branch is eligible to exceptionization, then the compiler will append a branch-exceptioning instruction to the control flow graph with no successor. A branch is not eligible to exceptionization during de-exceptionization, or in case the compiler determines not to exceptionize due to other reasons such as an explicit turn-off runtime option.

In the IR generation phase of the C2 compiler,

```
cfg = (control flow graph of IR)
bytecode_instr = get_next_instruction(worklist);
...
if (bytecode_instr is a branch) {
    (fallthrough_target, taken_target) = get_targets(bytecode_instr);
    prob = get_taken_target_probability(method_profile_data, bytecode_instr);
    min_prob = MIN(prob, 1.0-prob);
    if (min_prob < EXCEPTIONIZATION_THRESHOLD and
        eligible_to_exceptionization(bytecode_instr)) {
        // exceptionization
        if (min_prob == prob) {
            // prune taken_target
            insert_next_instruction(worklist, fallthrough_target);
            append_branch_exception_throw_instruction(cfg, taken_target);
        } else {
            // prune fallthrough_target
            insert_next_instruction(worklist, taken_target);
            append_branch_exception_throw_instruction(cfg, fallthrough_target);
        }
    } else {
        // normal translation / no exceptionization
        insert_next_instruction(worklist, fallthrough_target);
        insert_next_instruction(worklist, taken_target);
    }
}
```

Figure 3.8: Pseudo code for *exceptionization*.

### 3.3.3 Branch Exception Handling

In our implementation, the branch exception handling utilizes the uncommon trap and the deoptimization of HotSpot [20] as the underlying building blocks, as HotSpot implemented the Java exception handling with those mechanisms.

HotSpot adopts the uncommon trap as the bail-out mechanism to fall back on interpretation when a speculative assumption made by the JIT compiler turns out to be wrong during execution. Instead of generating generic code to satisfy all cases, the compiler performs speculative optimizations on a method with the profile information or a priori rules. The compiler inserts a guard check for the speculation with a conditional jump to the trap handler. If a failed speculation leads to the invalidity of the compiled code, then the trap handler eventually triggers recompilation of the method to correct the failed speculation. Currently, HotSpot generates the uncommon traps for method calls on unloaded classes, null-check and array bound-check transformations, or never-executed paths.

Deoptimization in HotSpot replaces the activation stack frame of a compiled method (compiled frame) into one or more interpreter stack frames (interpreter frames) and continues the method execution in the interpretation mode.<sup>5</sup> Thus, uncommon traps as well as OSR (On-Stack Replacement) transitions of methods are always accompanied by deoptimizations.

Figure 3.9 explains our branch exception handling. The throw `BranchException` instruction is actually a trap to the branch exception handler, which is a modified version of the Java exception handler in HotSpot to perform `BranchException`-specific actions such as the exceptionization scoreboard manipulation and the de-exceptionization management. The handler saves the method context and deoptimizes to the interpreter with constructed interpreter frames. The exceptionization scoreboard is a hash table to maintain the exceptionization count and the exception time stamp per exceptionized branch. The handler may trigger another action called de-exceptionization instead of

---

<sup>5</sup>A single compiled frame can be mapped to multiple interpreter frames because of method inlining.

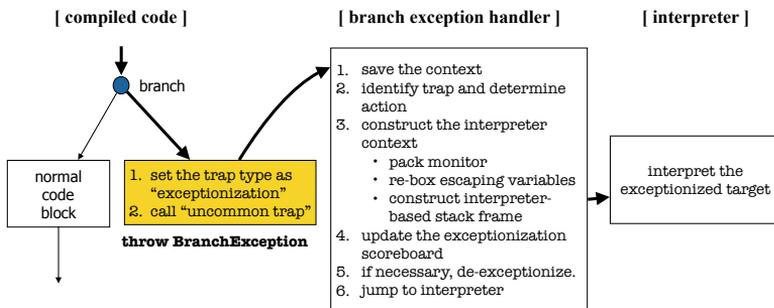


Figure 3.9: Branch exception handling.

interpretation if necessary.

### 3.3.4 De-exceptionization

Interpretation is the default method to run an exceptionized code, the pruned target of an exceptionized branch, but the frequent execution of an exceptionized code will degrade the performance. Therefore, the branch exception handler should monitor how frequently an exceptionized code is taken by looking up the exceptionization scoreboard. If it finds the frequency high, then it will de-exceptionize the exceptionized branch to eliminate interpretation overhead. Since the JIT compiler runs as a separate JVM thread, the code continues to be interpreted until the translated code is available. The gradual propagation introduces unstable branches so as to increase the importance of an efficient de-exceptionization mechanism for non-Java languages.

There are two issues in the de-exceptionization implementation: how to de-exceptionize a branch and when to trigger de-exceptionization.

#### How to *de-exceptionize* a branch

Since the HotSpot JIT compilers use a method as the compilation unit, the whole method may be recompiled with both targets of the exceptionized branch included in the control flow for de-exceptionization. After recompilation, the translated code

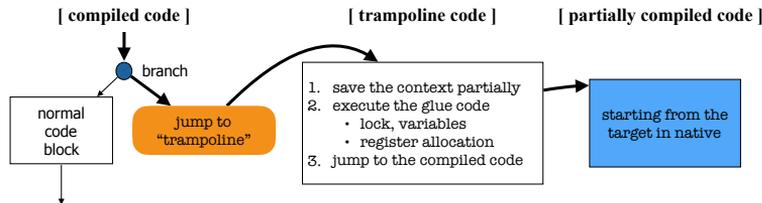


Figure 3.10: *De-exceptionization* with partial method compilation.

starts to be used for the following method invocations. This approach is so seamlessly pluggable to the original HotSpot that our current implementation uses this approach as the default option.

One possible drawback of this approach is that the benefits discussed in Figure 3.8 and the resources invested for exceptionization are all wasted out after de-exceptionization. So, we implemented another approach that only the portion of a method reachable from the exceptionized code is compiled instead of the whole method. When the partially compiled code is ready, the trap handler generates a trampoline code to enable direct control transfer to the partially compiled branch target from the branch as depicted in Figure 3.10.

Method inlining creates one compiled method corresponding to multiple bytecode methods. If a branch exception is thrown in an inlined method, then the portions of outer methods reachable from the inner method return may be fully compiled for de-exceptionization. However, we currently de-exceptionize only the inlined method with interpreting outer methods, because such full-stack compilation will introduce heavy code duplication and implementation complexity. Thus, a trampoline code is required to deoptimize the stack to interpret the outer methods.

The partial compilation approach keeps the original exceptionized code as well as the new code, leading to two specialized versions for the exceptionized branch. It should be noted that the blocks below Block 2 in Figure 3.7 (d) that correspond to Block F to Block J in Figure 3.7 (c) may be translated twice with the effect of tail

```

In the branch exception handler,
branch_instr = get_exception_throwing_instruction();
(exception_count, time_stamp) = update_exceptionization_score(
    exceptionization_scoreboard, branch_instr, cur_time_stamp);
exception_frequency(branch_instr) = 1 / (cur_time_stamp - time_stamp);

if (exception_count > DEEXCEPTIONIZATION_THRESHOLD &&
    exception_frequency > EXCEPTION_FREQUENCY_THRESHOLD) {
    call de-exceptionization for the method containing branch_instr;
}

```

Figure 3.11: Pseudo code to trigger *de-exceptionization*.

duplication [30] if the branch is de-exceptionized.

If a method contains too many exceptionized branches and many of them are de-exceptionized, then the whole method should be recompiled without exceptionization discarding all translated codes for the method, to control code duplications.

### **When to trigger *de-exceptionization***

By default, HotSpot invalidates the method immediately and then triggers recompilation of the method via another profiling interpretation, when the uncommon trap for a never-executed path happens in a compiled method. Instead, we adopted the more patient policy for de-exceptionization in order to keep using the exceptionized code more before the invalidation, which was found more effective by experiments presented in Section 3.4.

The branch exception handler updates the exceptionization scoreboard to keep the exception count and the time stamp for a branch with retrieving the previous information if a branch exception is thrown, as described in Figure 3.11. After calculating the exception count and the frequency with time stamps to measure the burstiness of a branch exception, the handler can trigger de-exceptionization if a branch exception is

thrown many times and frequently.

### 3.4 Experimental Results

We implemented exceptionization in the OpenJDK9+b99 package with HotSpot 1.9 JVM. While OpenJDK9+b99 shows almost the same performance for Java as OpenJDK8, the latest official package, it is 1.08 times faster for JavaScript,<sup>6</sup> because of the optimizations on the JavaScript library, the optimistic typing, and the execution model change [13, 31].

We experimented with an Intel i7-4790K 4-core machine with 32GB RAM and 4GHz clock speed, running Ubuntu 14.04 with Kernel 4.1.12. The implementation was compiled by gcc 4.8.4.

We measured the performance impact of exceptionization on JavaScript, Ruby, and Python, and Java. The JavaScript performance was evaluated with the Octane benchmark suite version 2.0 from Google [23] and the Kraken benchmark suite version 1.1 from Mozilla [32] with Nashorn as the language runtime. JRuby 9.0.5.0, a Ruby implementation on JVM [5] was used for Ruby with the publicly available performance benchmark suite [33]. The macro benchmarks in the official benchmark suites for Python [34] were experimented with Jython 2.7, a Python implementation [4]. SPECjvm2008 [24] and the DaCapo benchmark suite [35] were chosen for the Java performance measurement.

Our experiments show that the runtime performance of a non-Java dynamic language on JVM fluctuates significantly compared to Java, due to the multi-threaded and two-layered architecture as explained in Section 2. In order to amortize the fluctuating performance numbers, we extracted an arithmetic mean from the performance numbers generated through 100 separate JVM runs of each benchmark suite with no benchmark modification, which is different from Oracle’s approach via the JMH (Java

---

<sup>6</sup>We checked the performance with Octane for JavaScript and SPECjvm2008 for Java.

Microbenchmark Harness) framework requiring the dissection of a benchmark suite [36].<sup>7</sup>

### 3.4.1 Performance Impact on Non-Java Languages

We experimented the performance benefit from exceptionization with three configurations. Base is the vanilla OpenJDK 9 environment in which an application is launched in the default JVM configuration that is set up with no additional runtime option to “java”.<sup>8</sup> Except\_0005, the empirically chosen exceptionization environment, uses the value of 0.0005 as the exceptionization threshold. And Except\_0 with the threshold value of 0 is also tested for comparison. It only exceptionizes branches which have not-yet-taken targets. Both Except\_0005 and Except\_0 recompile the whole method for de-exceptionization.

#### JavaScript Performance

Figure 3.12 shows the relative performance of each configuration for Octane<sup>9</sup> and Figure 3.13 for Kraken compared to Base.<sup>10</sup> Except\_0005 improves the Octane performance by 6% and the Kraken performance by 5%. Except\_0 achieves the 8% improvement for Octane and 3% for Kraken.

Except\_0 demonstrates the consistent performance improvement over Base. It accelerates almost all benchmarks in Octane and Kraken except Box2D, beatdetection

---

<sup>7</sup>Commonly for the considered dynamic languages, we found that the generated performance numbers form an approximate Gaussian distribution and the size of 100 is a tradeoff between experiment efficiency and statistical soundness. The standard deviation of the numbers was about 6%, 5.5%, and 7% for JavaScript, Ruby, and Python, respectively.

<sup>8</sup>The ergonomic system in HotSpot automatically configures the JVM to use an 8GB max heap with the G1 garbage collector in the compressed 32-bit pointer mode in our environment.

<sup>9</sup>The base OpenJDK9 raises an error on zlib. So we removed it from the benchmark suite.

<sup>10</sup>We adopted the performance number generated by each suite as the performance indicator. “Score” of Octane and “Total” of Kraken are the overall performance numbers calculated by the suites. We checked their ratio practically corresponds to the geometric mean of each relative performance.

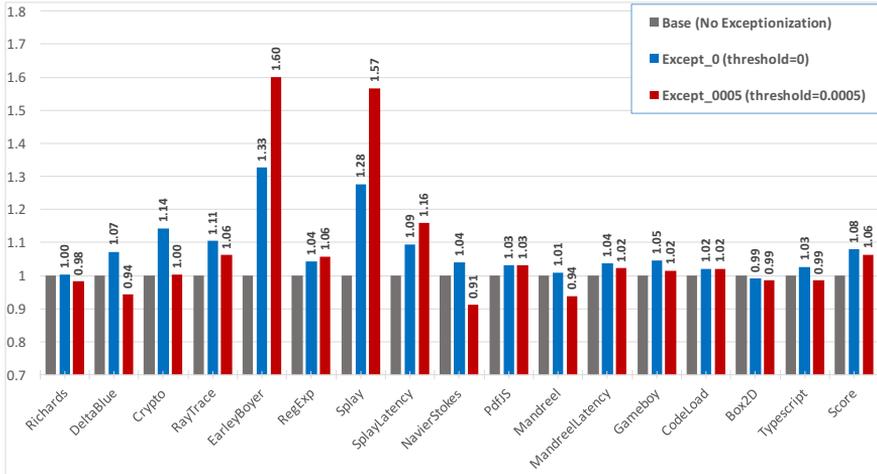


Figure 3.12: Relative performance on Octane (higher is better).

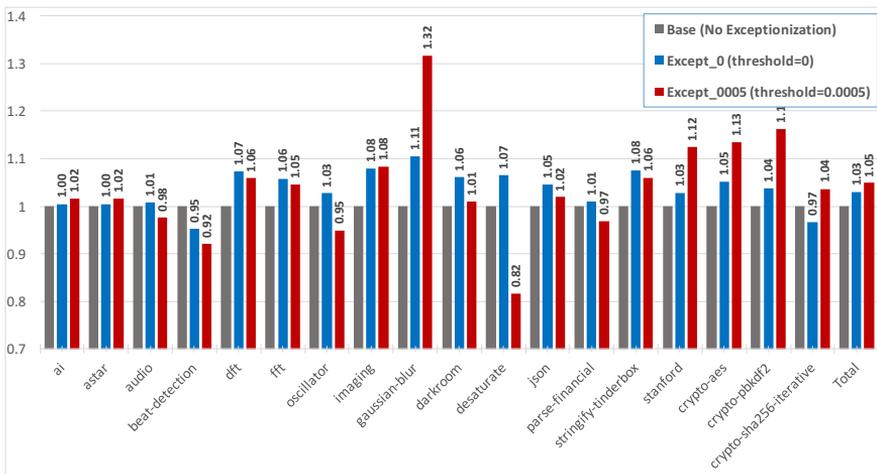


Figure 3.13: Relative performance on Kraken (higher is better).

and cryptosha256iterative.

Except\_0005 accelerates several benchmark including EarleyBoyer, Splay, Splay-Latency (Octane), gaussian-blur, stanford, and crypto-pbkdf2 (Kraken) significantly. And DeltaBlue, NavierSokes, Mandreel (Octane). beat-detection, oscillator, and de-saturate (Kraken) suffer from performance loss in Except\_0005, while most of them evade such loss in Except\_0. The exceptionized branch targets with the bias range of (0, 0.0005) in these benchmarks must have been taken frequently and the interpretation and de-exceptionization of them have damaged the performance. This implies that the cleverer target choice of exceptionization and de-exceptionization will be necessary for further improvement on the negatively impacted benchmarks.

Originally, Base cuts off not-yet-taken branch targets with the uncommon trap, meaning that the target selection policy becomes identical to Except\_0. However, the mechanism goes different when branching to the exceptionized path happens later, which provides a performance gain to Except\_0. While Base invalidates the method immediately and tries to recompile the whole method, Except\_0 tends to keep the original exceptionized method to trigger the recompilation for de-exceptionization as late as possible, so that the more optimized code for the method may be used longer. The evaluation of different de-exceptionization policies will be discussed in Section 3.4.4.

## **Ruby and Python Performance**

To check the general effectiveness of exceptionization, we also conducted performance experiments with JRuby 9.0.5.0 and Jython 2.7. Figure 3.14 shows that exceptionization improves the Ruby performance by 60% at maximum and by 6% on average. Python also gets faster by as much as 6% by exceptionization as shown in Figure 3.15. Thus, we are confident that exceptionization is a general JVM optimization technique to accelerate non-Java language implementations on JVM. As a whole, we can see that Except\_0005 shows the better performance than other configurations on the benchmarks for dynamic languages except Octane.

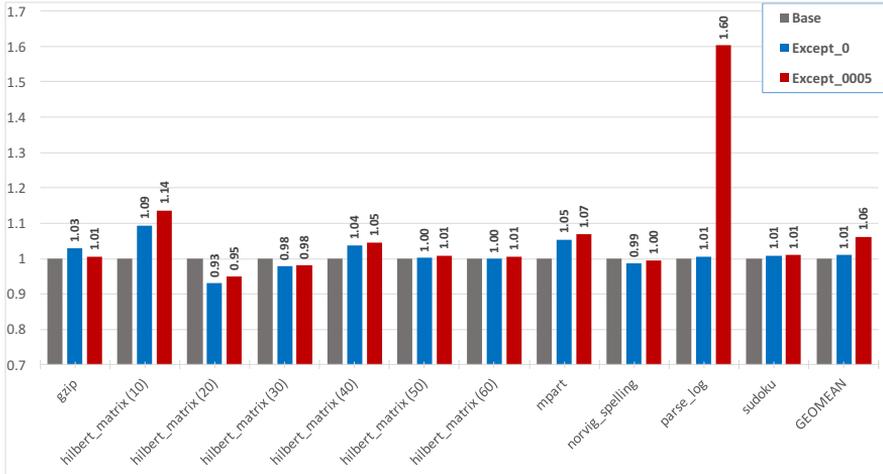


Figure 3.14: Relative performance on Ruby (higher is better).

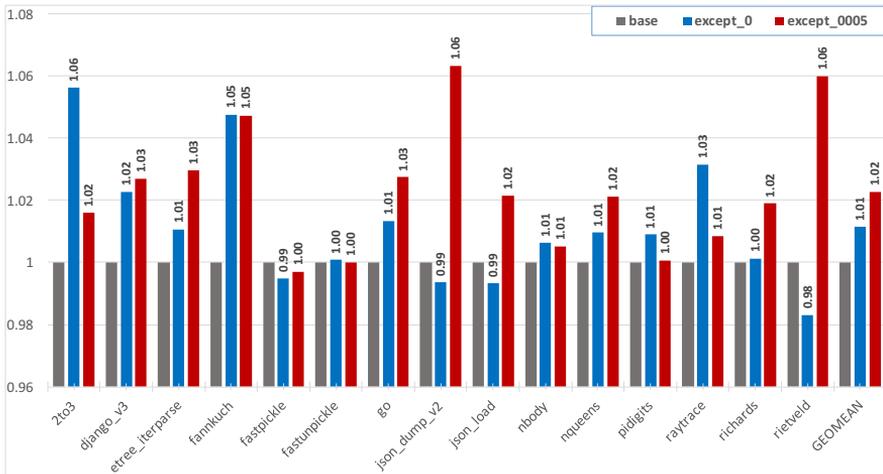


Figure 3.15: Relative performance on Python (higher is better).

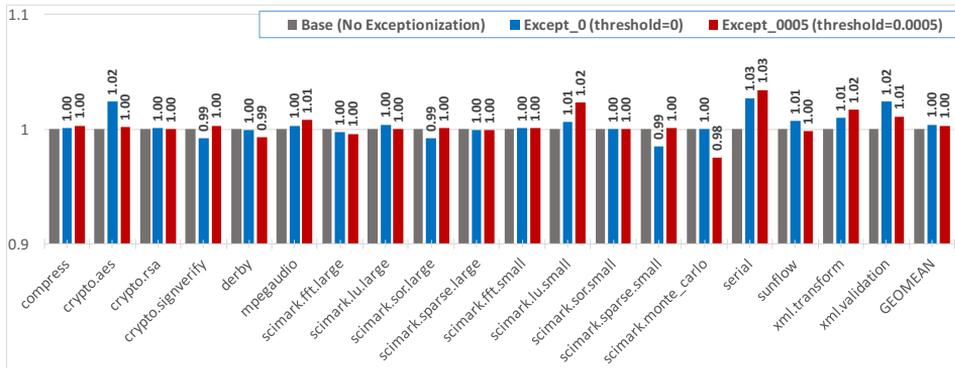


Figure 3.16: Relative performance on SPECjvm2008 (higher is better).

Since Nashorn and JRuby deploy better JIT compilers than Jython as discussed in Section 3.2, we speculate that the more a language runtime is sophisticated, the more performance gain can be expected from exceptionization. As will be explained soon, exceptionization helps the JVM JIT compiler to generate better-quality machine code for the complex language runtimes.

### 3.4.2 Performance Impact on Java

Since exceptionization modifies JVM itself, its performance impact on Java was tested with SPECjvm2008.<sup>11</sup> Figure 3.16 indicates that exceptionization has no negative performance impact on SPECjvm2008 by and large, which has been predicted by the observation that Java applications have relatively fewer branches to exceptionize than those of JavaScript, in Section 3.2. We also measured the relative performance with DaCapo, a server-side benchmark suite, which showed just 1% performance improvement by Except\_0005 and the same performance by Except\_0 as presented in Figure 3.17. Thus, exceptionization can be regarded as a safe optimization specific to the non-Java language execution on JVM with no negative impact on Java.

<sup>11</sup>Among the official benchmarks in SPECjvm2008, compiler.compiler and compiler.sunflow do not run on Java SE 8 and the later, because the benchmark uses APIs that became invalid due to the language specification change.

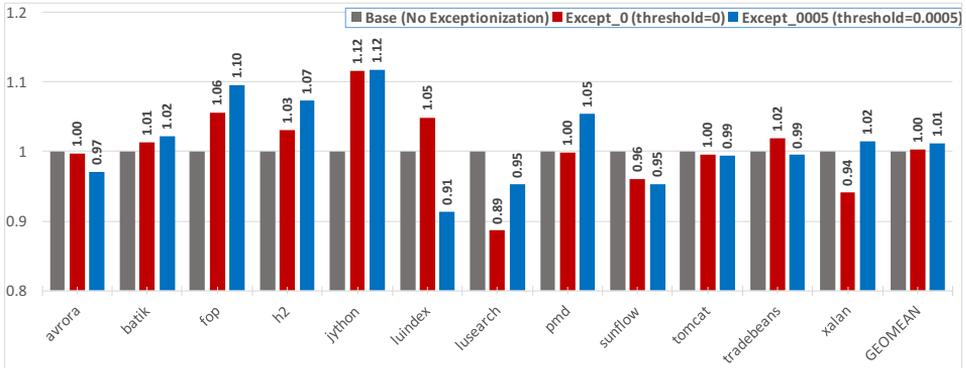


Figure 3.17: Relative performance on DaCapo (higher is better).

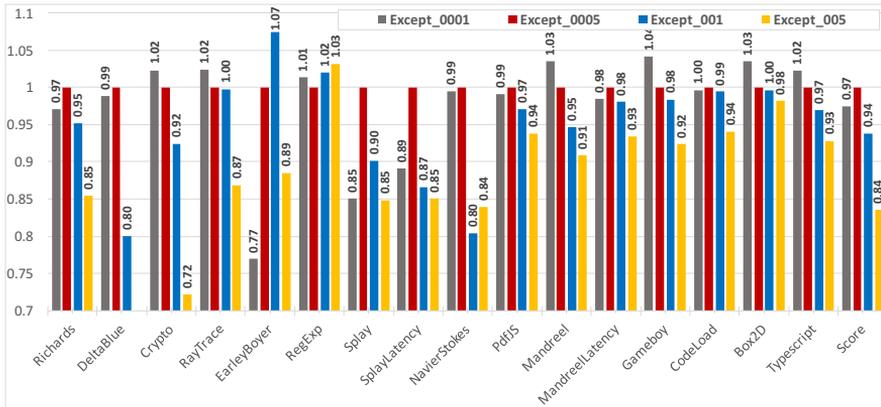


Figure 3.18: Relative performance with different thresholds on Octane.

### 3.4.3 Performance Variation by *Exceptionization* Threshold

Figure 3.18 compares several threshold configurations, Except\_0001 for 0.0001, Except\_0005 for 0.0005, Except\_001 for 0.001, and Except\_005 for 0.005. The branch bias range of  $[0, 0.001]$  looks like the key area to determine a threshold for exceptionization according to the result, which agrees to the observation of branch bias distribution in Figure 3.4. There is no linear relationship between threshold and performance. The threshold value of 0.0005 was found to be empirically the best for performance.

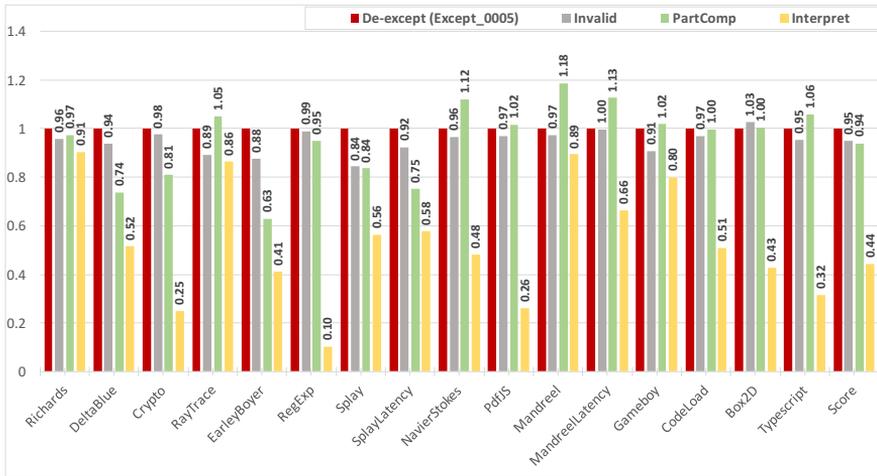


Figure 3.19: Relative performance with different de-exceptionizations (higher is better).

### 3.4.4 Performance Impact from *De-exceptionization*

Figure 3.19 evaluates the performance impact of various de-exceptionization policies based on Except\_0005 with Octane. De-exception is the default de-exceptionization configuration with the whole method compilation. Invalid simulates HotSpot’s approach for the uncommon traps on not-yet-taken paths by invalidating the compiled method if a branch exception is raised. PartComp compiles the method partially from the exceptionized target instead of the whole method compilation. Interpret just interprets an exceptionized code without de-exceptionization.

Invalid suffers from 5% performance loss, because it is too strict and impatient to an exceptioning branch to keep the initial exceptionized code for later re-use, resulting to recompilation overhead and slightly poorer-quality code. Anyway, it is such a suitable approach for Java with stable branches that the similar policy is effective in the base HotSpot JVM.

PartComp shows performance improvement for several benchmarks including Navier-Stokes and Mandreel over the default Except\_0005, though the overall performance gain is not satisfactory yet. So, more deliberate de-exceptionization techniques cou-

Table 3.3: Raised branch exceptions during Octane execution

	Except_0005				Except_0			
	De-exception	Invalid	PartComp	Interpret	De-exception	Invalid	PartComp	Interpret
Richards	642	253	67749	842850	418	138	69100	454543
DeltaBlue	44	13	24254	385242	37	12	27132	104185
Crypto	78	30	18061	272184	61	20	13347	241276
RayTrace	68	26	7077	82864	37	13	5511	16418
EarleyBoyer	97	30	13794	537240	73	18	12408	333393
RegExp	114	36	32340	8624972	77	21	27217	7067358
Splay	48	22	8580	321046	41	14	4446	121999
SplayLatency	1	0	82	238	1	0	41	155
NavierStokes	36	13	1422	4977	14	3	950	4393
PdfJS	359	135	69372	2924149	265	89	62854	1191719
Mandreel	88	42	6255	1083826	56	22	4973	630910
MandreelLatency	0	0	1	22	0	0	0	12
Gameboy	142	67	16567	579926	69	28	17548	341434
CodeLoad	173	48	49386	2577918	115	34	40714	2447219
Box2D	49	17	13987	2222275	40	13	14195	1886383
Typescript	209	81	41182	17702003	122	39	41293	15261368
Score	2	2	7	248	3	2	11	211
Total Number	2150	815	370116	38161980	1429	466	341740	30102976

pled with the partial method compilation may provide opportunity to improve the performance more.

We counted the branch exceptions raised during the Octane run for Except\_0005 and Except\_0 with the four de-exceptionization configurations as shown in Table 3.3. As expected, Except\_0005 tends to generate more branch exceptions than Except\_0. De-exception and Invalid reduce branch exceptions significantly due to de-exceptionization, when compared to Interpret. Since Invalid is more rigid to branch exceptions, it raises the least number of exceptions.

PartComp generates moderate number of branch exceptions. Too many branch exceptions can be a performance burden,<sup>12</sup> while too few exceptions may blow away the possible performance gain from exceptionization. Linking with the result in Fig 3.19,

<sup>12</sup>The branch exception handling time takes 22 microseconds in our experimental environment.

Table 3.4: *Exceptionizable* branches in JavaScript and Java encountered during compilation

Bias	Octane		SPECjvm2008	
	Exceptionized	Normal	Exceptionized	Normal
0 (Not-Yet-Taken)	112926	57573	44876	0
(0, 0.0005)	3125	37895	3359	1855

finding heuristics to control the raised branch exception number will be one of the future research topics.

The large performance loss of Interpret and the large number of raised exceptions are still surprising with knowledge that a not-yet-taken path tends not to be taken in future. Table 3.4 presents the numbers of exceptionizable branches, whose biases are below the threshold, encountered by the C2 compiler for Octane and SPECjvm2008. Some of them are exceptionized and the others are just compiled as normal due to de-exceptionization. The exceptionizable branches may have been counted repeatedly because de-exceptionization may compile a method multiple times.

Contrary to Octane, SPECjvm2008 shows no normal branch with the zero bias. If a branch has a not-yet-taken target during C2 compilation, then the target will keep never-taken in SPECjvm2008. This confirms the fourth observation in Section 3.2, that JavaScript shows more unstable behaviours than Java. The gradual propagation may bring about the branch instability. As the application phase changes, different parts of the Nashorn engine will be invoked, leading to unstable branch behaviours with different control flows. This property explains the poor performance of Interpret well.

Table 3.4 also indicates that there are not a small number of unstable and exceptionized branches eligible to de-exceptionization in JavaScript, which accounts for many branch exceptions of PartComp and Interpret in Table 3.3.

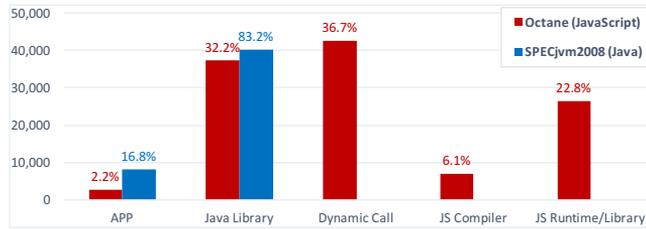


Figure 3.20: Origin distribution of exceptionized branches in JavaScript and Java.

### 3.4.5 Origin of Exceptionized Branches in JavaScript

To clarify where exceptionization of the excessive branches happens in JavaScript, we categorized the origins of exceptionized branches as shown in Figure 3.20. The ratio of a specific category is given as the label above each bar. Since Dynamic Call, JS Compiler, and JS Runtime/Library correspond to the Nashorn engine, we can see that about two thirds of exceptionized branches are found to come from the language runtime. The comparison between JavaScript to Java reveals that exceptionization’s simplifying control flows of the JavaScript runtime leads to the performance improvement.

As described previously, the optimistic typing in OpenJDK 9 significantly reduces branches via type specialization, which explains the relative small portion of APP. Mature language runtimes are equipped with such aggressive specializations that exceptionization improves the overall performance through better JIT compilation of the language runtimes.

On the contrary, naïve language runtime implementations may not apply such specialization. For example, Nashorn in OpenJDK 8 applies no specialization by default, and most generic operations are translated to calls to methods with `java.lang.Object` parameters in the JavaScript library, containing a series of type checks for precise-type operations. The portion of JS Runtime/Library gets increased as well as APP in such cases.

Therefore, we can conclude that the efficient JIT compilation of language runtimes such as exceptionization is crucial for the performance regardless of the maturity of

the runtimes.

### 3.5 Related Work

The SELF-91 compiler introduced deferred compilation [37] to generate code for the uncommon branch targets lazily. If an uncommon branch target is taken, the system encounters the embedded uncommon branch extension that invokes a compiler to generate code on demand. The deferred compilation generally aims to save the time required for type analysis and compiled code space.

The deferred compilation utilizes the type analysis to predict type checking branches and virtual method invocations for the lazy code generation. On the while, exceptionization focuses on handling branch bytecodes in context of supporting non-Java languages on JVM. Exceptionization depends solely on the branch profile information provided by JVM, so that it may collaborate with the deferred compilation scheme orthogonally. The subtype checking and method invocation mechanisms in HotSpot already work in a way similar to the deferred compilation through the separate translation of common flows and uncommon flows.

The SELF-91 compiler also introduces a code replication technique called extended message splitting to split control flow merges for more precise type analysis [29]. Exceptionization achieves the similar effect by removing an incoming path to a control merge with delaying the code replication until de-exceptionization if a branch is highly biased.

The partial method compilation [38] and the region-based compilation [39] selected a compilation unit or region via heuristics on profile information. Instead of the full compilation, they compile the unit partially to reduce the compilation overhead with additional runtime performance improvement thanks to better control flow, which may be beneficial to dynamic compilation systems.

Similarly, a trace-based JIT compiler was researched on top of the SpiderMonkey

JavaScript VM [14]. Instead of compiling a method, the VM collects deeply-inlined trace information and generates machine code for selected traces with profile-based optimizations. Thus, the path pruning in exceptionization can be implicitly handled during the trace compilation.

Since it is still an interesting issue, which approach is the better between the method-based JIT and the partially-compiling trace-based JIT, we consider exceptionization as a hybrid approach to apply the benefit of the trace-based JIT to the method-based JIT environment by pruning infrequent paths of branches during the method compilation.

HotSpot [20] utilized the uncommon trap to facilitate its own deferred compilation mechanism. With data aggregated from profiling, flow analysis, and a priori rules, HotSpot performs speculative optimizations to generate machine code tailored for common cases by inserting uncommon traps for the uncommon cases. The uncommon trap triggers HotSpot to deoptimize into the interpreter with invalidating speculatively optimized code.

HotSpot prunes a not-yet-taken branch target with assumption that the path will not be reached, which works well for Java, not for non-Java languages such as JavaScript, due to the different branch characteristics as we discussed previously. In order to exploit the relatively unstable and highly-biased branches of non-Java languages, exceptionization proposes a new policy to prune an infrequent branch target and to handle raised branch exceptions with augmentation through de-exceptionization.

Jikes RVM [40] adopted an extended version of the HotSpot uncommon trap that is streamlined with the OSR (On-Stack Replacement) transition. It deployed an efficient OSR transition mechanism to transfer control from the trapping method to the fresh code generated by the deferred compilation. Each OSR point in a method can be mapped to an uncommon trap for the deferred compilation and it can also be the entry point of the fresh code. The OSR transition is achieved through the on-demand generation of a compromising code between the two corresponding contexts. The partial

method recompilation approach for de-exceptionization shares the common idea with the Jikes RVM OSR transition because it also generates a specialized trampoline code for the branch to de-exceptionize.

LaTTe JVM introduced an on-demand exception handler translation [10]. It ignores any exception handlers in a method while JIT compiling the method. If an exception is thrown, then the JVM translates the appropriate exception handler on demand and jumps to the generated code. Exceptionization follows this approach by modeling a highly-biased branch as an implicit exception-throwing instruction.

The Fiorano JIT compiler team shared their experience in extending a statically typed language JIT compiler for Python [15]. The generic operations in a dynamically typed language often involve library calls, heap side-effects, and complex control flows, which are very hard to analyze. Thus, the main optimization opportunities in a dynamically typed language come from the type specialization with profile information, a kind of strength reduction on the generic operations. The team advocates the guard-based specialization that creates a specialized code guarded by a runtime condition.

We regard exceptionization as a JVM-level support to facilitate guard-based specializations applied by the language runtime on top of JVM. Nashorn performs such specialization aggressively via the optimistic typing and exceptionization can accelerate both the specialized application code and the language runtime simultaneously.

Truffle with Graal formed a generic multi-language virtual machine written in Java from Oracle Lab [41]. A guest language may be implemented with the combination of the front-end parser, the AST interpreter, and the JIT compilation system. The language front-end parser generates an AST representation that can be executed via either interpretation or JIT compilation. Therefore, the system architecture to support non-Java languages becomes more concise and better structured than the HotSpot architecture in Figure 3.1.

Exceptionization could be integrated in the framework. However, we wanted to

evaluate the effect of exceptionization in a product-level JVM, with no dependence on a guest language implementation.

## Chapter 4

### Summary and Conclusion

In this paper, we described the design and implementation of LaTTe, a Java JIT compiler with fast and efficient register mapping and allocation. Our aggressive register mapping with a separate pass for real register allocation that coalesces copies with a local lookahead is an elaborate engineering solution that trades off the quality of generated code and the speed of JIT compilation. This trade-off was confirmed empirically by measuring translation overhead and performance impact. The overall performance of the LaTTe JVM is shown competitive without any complex adaptive compilation mechanism, which proposes an ideal JVM implementation option for embedded devices with tight resource constraints.

This paper also introduced *exceptionization*, a JVM-level optimization to boost up the performance of non-Java languages on JVM, with the Nashorn JavaScript engine over HotSpot JVM in the start-of-art OpenJDK 9 package.

A non-Java language requires the corresponding language runtime as the intermediate layer imposing more bytecode for JVM to execute. JVM should confront more complex control flows to support alien language semantics and to enable high-level language optimizations. Consequently, we found that the JVM executing JavaScript encounters 2.8 times more bytecode branches than when executing Java, most of which are highly biased.

*Exceptionization* transforms a highly-biased branch to an implicit `BranchException`-throw instruction by pruning its less-taken target from control flow by the JVM JIT compiler. The JIT compiler manages to generate better-quality code thanks to the more precise data flow analysis on the simplified control flow. We described how to handle a `BranchException` raised by taking the pruned target, which relies on the interpretation. *De-exceptionization* is streamlined with the branch exception handling in order to eliminate execution overhead from frequent branch exceptions.

Our experiments showed that *Exceptionization* increases JavaScript performance by 6% for Octane and 5% for Kraken in OpenJDK 9, which is almost equivalent to the performance improvement from the major JavaScript engine upgrade from OpenJDK 8 to OpenJDK 9. Testing with SPECjvm2008 shows that *exceptionization* is transparent to Java performance.

Not a small number of highly-biased branches were found to become so *unstable*, which means that their actual behaviors conflict with predictions based on the profile information, as to spoil the benefit from *exceptionization*, specifically during JavaScript execution. Therefore, meticulous *de-exceptionization* mechanisms are important for the JavaScript performance.

We discovered that the JavaScript engine itself creates *exceptionizable* branches so massively that the major performance improvement of *exceptionization* comes from the better handling of Nashorn by the JVM JIT compiler. This indicates that the careful handling of a language runtime itself is important for the overall performance as well as integrating various optimization techniques into the language runtime for the efficient non-Java execution on JVM.

*Exceptionization* is easily applicable to host other non-Java languages because *exceptionization* is a JVM-level optimization. So, we experimented *exceptionization* with other language runtimes for Ruby and Python and found that it also increases the performance consistently.

# Bibliography

- [1] Debian.org, “The computer language benchmarks game,” 2016. [Online]. Available: <http://benchmarksgame.alioth.debian.org>
- [2] R.-G. Urma, “Alternative languages for the java virtual machine,” *Java Magazine*, pp. 5–11, Jul. 2014.
- [3] J. Ponge, “Oracle nashorn: A next-generation javascript engine for the jvm,” *Java Magazine*, pp. 59–65, Jan. 2014.
- [4] F. Wierzbickis, “Jython: Python for the java platform,” May 2015. [Online]. Available: <http://www.jython.org/>
- [5] jRuby.org, “Jruby: The ruby programming language on the jvm,” Apr. 2016. [Online]. Available: <http://www.jython.org/>
- [6] T. Linkholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*. Boston, MA, USA: Addison-Wesley, May 2014. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se8/html/index.html>
- [7] J. R. Rose, “Jsr 292: Supporting dynamically typed languages on the java platform,” Jul. 2011. [Online]. Available: <https://jcp.org/en/jsr/detail?id=292>
- [8] M. Grogan, “Jsr 223: Scripting for the java platform,” Dec. 2006. [Online]. Available: <https://jcp.org/en/jsr/detail?id=223>

- [9] W. H. Li, D. R. White, and J. Singer, “Jvm-hosted languages: They talk the talk, but do they walk the walk?” in *Proceedings of 2013 International Conference on Principles and Practices of Programming on the Java Platform: virtual machines, languages, and tools*, ser. PPPJ ’13. New York, NY, USA: ACM Press, 2013, pp. 101–112.
- [10] S. Lee, B.-S. Yang, and S.-M. Moon, “Efficient java exception handling in just-in-time compilation,” *Software: Practice and Experience*, vol. 34, no. 15, pp. 1463–1480, Dec. 2004.
- [11] Oracle, “Openjdk: Jdk 9,” 2016. [Online]. Available: <http://openjdk.java.net/projects/jdk9/>
- [12] —, “Openjdk: Project nashorn,” 2016. [Online]. Available: <http://openjdk.java.net/projects/nashorn/>
- [13] M. Grogan, “Jep 196: Nashorn optimistic typing,” 2014. [Online]. Available: <http://openjdk.java.net/jeps/196>
- [14] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, “Trace-based just-in-time type specialization for dynamic languages,” in *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, ser. PLDI ’09. New York, NY, USA: ACM Press, 2009, pp. 465–478.
- [15] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu, “On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages,” in *Proceedings of the 2012 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA ’12. New York, NY, USA: ACM Press, 2012, pp. 195–212.

- [16] J. R. Rose, “Bytecodes meet combinators: invokedynamic on the jvm,” in *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL ’09. New York, NY, USA: ACM Press, 2009.
- [17] J. Duke and J. Rose, “Openjdk wiki: Method handles and invokedynamic,” Aug. 2013. [Online]. Available: <https://wiki.openjdk.java.net/display/HotSpot/Method+handles+and+invokedynamic>
- [18] R. Hickey, “The clojure programming language,” in *Proceedings of the 2008 Symposium on Dynamic Languages*, ser. DLS ’08. New York, NY, USA: ACM Press, 2008.
- [19] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala, Third Edition*. Walnut Creek, CA, USA: Artima Inc, Apr. 2016.
- [20] M. Paleczny, C. Vick, and C. Click, “The java hotspot<sup>TM</sup> server compiler,” in *Proceedings of the 2001 Symposium on Java<sup>TM</sup> Virtual Machine Research and Technology*, ser. JVM ’01. Berkeley, CA: USENIX Association, 2001.
- [21] ECMA, *ECMAScript Language Specification, Standard ECMA-262*, 5th ed. CH-1204 Geneva: Ecma International, Jun. 2011. [Online]. Available: <http://www.ecma-international.org/ecma-262/5.1/>
- [22] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, , and W. Binder, “Characteristics of dynamic jvm languages,” in *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL ’13. New York, NY, USA: ACM Press, 2013, pp. 11–20.
- [23] Google, “Octane: The javascript benchmark suite for the modern web,” 2016. [Online]. Available: <https://developers.google.com/octane/>
- [24] SPEC, “Specjvm2008,” Nov. 2015. [Online]. Available: <https://www.spec.org/jvm2008/>

- [25] M. N. Kedlaya, J. Roesch, B. Robotmili, M. Reshadi, and B. Hardekopf, “Improved type specialization for dynamic scripting languages,” in *Proceedings of the 9th Symposium on Dynamic Languages*, ser. DLS ’13. New York, NY, USA: ACM Press, 2013, pp. 37–48.
- [26] J. Gosling, B. Joy, G. L. S. Jr., G. Brach, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*. Boston, MA, USA: Addison-Wesley, May 2014. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se8/html/index.html>
- [27] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russel, , and D. Cox, “Design of the java hotspot<sup>TM</sup> client compiler for java 6,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 1, May 2008.
- [28] S. S. Muchnick, *Advanced Compiler Design and Implementation*. New York, NY: Morgan Kaufmann, 1997.
- [29] C. Chambers and D. Ungar, “Iterative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’90. New York, NY, USA: ACM Press, 1990, pp. 150–164.
- [30] W.-M. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. E. Holm, and D. M. Lavery, “The superblock: An effective technique for vliw and superscalar compilation,” *The Journal of Supercomputing - Special Issue on Instruction-Level Parallelism*, vol. 7, no. 1-2, pp. 229–248, May 1993.
- [31] M. Lagergren, “Nashorn for java 9: Quickly bootstrapping nashorn with a new execution model,” in *JVM Language Summit 2015*, ser. JVMLS ’15, Aug. 2015. [Online]. Available: <http://www.slideshare.net/lagergren/a-new-execution-model-for-nashorn-in-java-9>

- [32] Mozilla.org, “Kraken javascript benchmark (version 1.1),” Sep. 2010. [Online]. Available: <http://krakenbenchmark.mozilla.org/>
- [33] A. Cangiano, “Ruby benchmark suite,” 2016. [Online]. Available: <https://github.com/acangiano/ruby-benchmark-suite>
- [34] Python.org, “Python benchmarks by python software foundation,” 2016. [Online]. Available: <https://hg.python.org/benchmarks/>
- [35] DaCapo, “The dacapo benchmark suite,” Dec. 2009. [Online]. Available: <http://dacapobench.org>
- [36] M. Lagergren, “Openjdk wiki: Monitoring nashorn performance,” 2013. [Online]. Available: <https://wiki.openjdk.java.net/display/Nashorn/Monitoring+Nashorn+Performance>
- [37] C. Chambers and D. Ungar, “Making pure object-oriented languages practical,” in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’91. New York, NY, USA: ACM Press, 1991, pp. 1–15.
- [38] J. Whaley, “Partial method compilation using dynamic profile information,” in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’01. New York, NY, USA: ACM Press, 2001, pp. 166–179.
- [39] T. Suganuma, T. Yasue, and T. Nakatani, “A region-based compilation technique for dynamic compilers,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 1, pp. 134–174, Jan. 2006.
- [40] S. J. Fink and F. Qian, “Design, implementation and evaluation of adaptive re-compilation with on-stack replacement,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime*

*Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 241–252.

- [41] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, “One vm to rule them all,” in *Proceedings of the 2013 ACM International Symposium on New ideas, New paradigms, and Reflections on Programming & Software*. New York, NY, USA: ACM Press, 2013, pp. 187–204.

## 초 록

자바 가상 머신 (JVM)은 자바로 작성된 프로그램을 수행하기 위한 환경으로 제안되어, 특정 머신에 독립적인 가상의 실행 환경을 제공한다. 32비트 스택 머신으로서 동작하는 자바 가상 머신은, 그것이 정확하게 이식된 어떠한 시스템 상에서도 자바 컴파일러를 통하여 생성된 바이트코드 명령어들을 수행할 수 있다. 자바 가상 머신의 이러한 장치 독립성은 자바 프로그래밍 언어와 자바 가상 머신 자체의 큰 성공을 가져와서, 현재 클라우드 서버에서부터 핸드폰과 스마트카드를 아우르는 다양한 시스템들에서 자바 실행 환경으로서 사용되고 있다.

바이트코드 명령어가 특정 시스템 상에서 제대로 동작하기 위해서는, 자바 가상 머신의 런타임 환경에 의해 정확하게 인터프리트되어야 하는데, 이러한 인터프리트 오버헤드는 자바 프로그램 성능 저하의 중요한 원인이 된다. 반면에 전통적인 C/C++ 프로그램의 경우에는 특정 시스템에 적합한 형태로 컴파일되어 동작하는 방식을 사용하기 때문에 이러한 성능 저하가 발생하지 않는다. 이러한 문제를 해결하기 위해 최근에 일반적으로 채택되는 성능 향상 모듈로서, 자바 적시 (Just-in-Time, JIT) 컴파일러 기능이 등장하였는데, 이것은 자바 바이트코드를 프로그램 실행 중에 필요에 따라 실제 머신 코드로 동적으로 변환하는 방식을 사용한다.

이러한 자바 적시 컴파일 기법에 있어서의 가장 큰 문제는 자바 가상 머신의 스택 엔트리와 지역 변수들을 효과적면서도 빠르게 실제 머신 상의 레지스터로 대응시키는 방법에 대한 것이다. 왜냐하면 레지스터 기반의 계산 작업은 메모리를 사용하는 것에 비해 훨씬 빠른 반면, 이러한 적시 컴파일에 소요되는 시간은 프로그램의 전체 수행 시간의 일부를 차지하게 되기 때문이다. 이 논문에서는 효율적으로 레지스터

대응이 이루어진 RISC 코드를 빠른 속도로 생성할 수 있는 오픈 소스 자바 적시 컴파일러인 LaTTe를 소개한다. LaTTe는 먼저 모두 지역 변수들과 스택 엔트리를 가상 레지스터에 대응을 시킨 후, 이것들을 실제 머신 레지스터에 할당하는 작업을 수행한다. 이 과정에서 지역 변수들과 스택 엔트리 사이의 push와 pop에 대응하는 복사 명령어들을 적극적으로 결합, 제거하는 작업을 진행한다. 이러한 효율적인 레지스터 할당 기법에 더하여, LaTTe는 common subexpression elimination, 동적 메쏘드 인라인, specialization과 같은 다양한 전통적 최적화 및 객체 지향 최적화 기법을 포함하고 있다. 우리는 또한 자바 예외 상황 처리와 monitor 처리를 위한 새로운 기법인 “주문형 예외 상황 처리”와 “경량 monitor”를 LaTTe에 포함시켜, 전체 자바 가상 머신의 성능을 더욱 끌어올렸다.

실험 결과 상에서 LaTTe의 정교한 레지스터 대응과 할당 기법은 실질적인 성능 향상을 제공하여, 모든 지역 변수와 스택 엔트리를 메모리에 기계적으로 대응시키는 간단한 적시 컴파일러 대비 두배 빨라진 성능을 보인다. 또한 LaTTe는 바이트코드를 위한 레지스터 대응 및 할당 기법에 있어서 생성 코드의 품질과 작업의 속도 상의 합리적인 타협점을 찾고 있음을 확인할 수 있다. 우리는 이러한 실험 결과가 병렬 및 분산형 자바 컴퓨팅 환경에서도 다음과 같이 도움이 될 것으로 기대한다. 1) 단일 쓰레드 자바 성능을 증대, 2) coherence와 쓰레드 동기화를 위해 전체 시스템이 적절하게 정렬해야 하는 메모리 접근 명령어의 수가 크게 감소.

추가적으로, 최근에 자바 가상 머신은 자바스크립트, 파이썬, 루비와 같은 주류 동적 스크립트 언어들을 수행할 수 있는 범용 언어 실행 환경으로 진화하였다. 이러한 언어들은 동적 타입과 first-class 함수와 같은 자바에 없는 복잡한 기능들을 포함하고 있기 때문에, 이것들의 지원을 위해서는 추가적인 언어 런타임 엔진과 바이트코드 확장이 자바 가상 머신 상에서 제공이 되어야 한다. 현재 강력한 적시 컴파일러를 채용한 고성능 자바 가상 머신들이 존재하고 있지만, 동적 언어들을 자바 가상 머신 상에서 효율적으로 실행하는 것은 여전히 어려운 문제이다.

이 논문에서는 exceptionization이라고 하는 자바 가상 머신의 적시 컴파일러를 위한 간단하고도 새로운 최적화 기법을 소개하여, 자바 가상 머신 기반 언어 런타임의 성능을 향상시키고자 한다. 우리는 자바를 수행하는 경우에 비해, 비자바 언어를

실행하는 경우에는 자바 가상 머신이 최소 2배 더 많은 분기 명령어를 실행해야 하는 것을 관찰하였다. 그리고 이러한 분기 명령어들 중의 많은 수가 한쪽 타겟으로 크게 치우치는 경향이 있는 것도 확인하였다. Exceptionization은 이렇게 크게 치우친 분기 명령어를 마치 자바 예외 상황인 exception을 생성하는 명령어인 것처럼 취급하게 한다. 이를 통하여 자바 가상 머신의 적시 컴파일러는 해당 분기 명령어의 적게 방문되는 타겟을 전체 제어 흐름 그래프에서 떼어낸 후에, 생성된 자주 수행되는 제어 흐름을 보다 정교한 최적화 기법을 적용하여 공격적으로 컴파일을 하여, 고품질의 머신 코드를 생성하게 된다. 만약 떼어낸 타겟이 추후에 실행이 된다면, 자바 가상 머신은 그것을 마치 자바 예외 상황 처리자인 catch block처럼 처리하여 수행시킨다. 우리는 또한 떼어낸 타겟이 예상보다 더 자주 실행되는 경우를 처리하기 위한 기법으로 de-exceptionization을 개발하였다.

Exceptionization은 범용의 자바 가상 머신 최적화 기법으로서, 특정 언어 런타임에 독립적으로 동작하기 때문에, 자바 가상 머신 상의 타언어 수행에 일반적으로 적용이 가능하다. 실험 결과 상에서는 exceptionization이 몇개의 비자바 언어들의 성능을 가속시키는 것을 확인할 수 있다. HotSpot 1.9 JVM과 Nashorn 자바 스크립트 엔진에 적용하여 Octane 벤치마크를 수행하는 경우, 자바스크립트 성능이 최대 60%, 평균 6% 정도의 향상이 있음을 확인하였다. 추가적으로 루비의 경우에도 최대 60%, 평균 6%의 성능향상을, 파이썬의 경우에는 6%의 성능 향상이 발생하였다. 우리는 exceptionization이 어플리케이션 코드나 자바 클래스 라이브러리에 포함된 분기 명령어들보다, 언어 런타임 자체의 분기 명령어들에 보다 효과적으로 적용되고 있음을 확인하였다. 이것은 exceptionization의 성능 향상 효과가 비자바 언어 런타임을 보다 효율적으로 적시 컴파일을 함으로써 발생하는 것을 의미한다.

**주요어:** 자바, 자바스크립트, 자바 가상 머신, 적시 컴파일러, 동적 언어, 컴파일러 최적화

**학번:** 99420-841