Exceptionization: A Java VM Optimization for Non-Java Languages

BYUNG-SUN YANG, JAE-YUN KIM, and SOO-MOOK MOON, Seoul National University

Java virtual machine (JVM) has recently evolved into a general-purpose language runtime environment to execute popular programming languages such as JavaScript, Ruby, Python, and 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 article 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 were taken, then it would run like a Java exception handler, that is, a catch block. We also devised *de-exceptionization*, a mechanism to cope with the case when a pruned path is executed more often than expected.

Since exceptionization is a generic JVM optimization, independent of any specific language runtime, it would be generally applicable to other language runtimes on the JVM. Our experimental result shows that exceptionization accelerates the performance of several non-Java languages. For example, JavaScript-on-JVM runs faster by as much as 60% and by 6% on average, when experimented with the Octane benchmark suite on Oracle's latest Nashorn JavaScript engine and HotSpot 1.9 JVM. Furthermore, the performance of Ruby-on-JVM shows an improvement by as much as 60% and by 6% on average, while Python-on-JVM improves by as much as 6% and by 2% on average. We found that exceptionization is more effective to apply to the branch bytecode of the language runtime itself 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 language runtime of non-Java languages.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; **Dynamic compilers**; **Runtime environments**;

Additional Key Words and Phrases: Javascript, java virtual machine, just-in-time compiler, non-java languages

ACM Reference Format:

Byung-Sun Yang, Jae-Yun Kim, and Soo-Mook Moon. 2017. *Exceptionization*: A java VM optimization for non-java languages. ACM Trans. Archit. Code Optim. 14, 1, Article 5 (April 2017), 25 pages.

DOI: <http://dx.doi.org/10.1145/3046681>

This work was supported by the Ministry of Trade, Industry & Energy (MOTIE) through the Electronics and Telecommunications Research Institute (ETRI) (Project No. 10045344).

New paper, not an extension of a conference paper.

Authors' addresses: B.-S. Yang, J.-Y. Kim, and S.-M. Moon, Department of Electrical Engineering and Computer Science, Seoul National University, Seoul 08826, South Korea; emails: {byungsun.yang, jaeykim}@altair.snu.ac.kr, smoon@snu.ac.kr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1544-3566/2017/04-ART5 \$15.00

DOI: <http://dx.doi.org/10.1145/3046681>

1. INTRODUCTION

Since the late 1990s, the performance of the Java virtual machine (JVM) has been matured enough, allowing Java to run at less than twice slower than C/C++ on some benchmarks [Debian.org 2016]. This is mainly due to its just-in-time (JIT) compiler that translates the Java bytecode into machine code at runtime, such as the HotSpot JIT Compilers in Java Development Kits (JDKs).

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 [Urma 2014]. What made this possible are advents of the language runtimes in Java (e.g., Nashorn JavaScript engine [Ponge 2014], Jython engine [Wierzbickis 2015], and JRuby engine [jRuby.org 2016]), extensions of the JVM specification [Linkholm et al. 2014; Rose 2011], and augmentation of the script APIs [Grogan 2006] 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 also differs from that of regular Java in terms of memory usage and execution characteristics [Li et al. 2013]. Although there are some promising initial reports on the performance of the JVM-based language environments [Debian.org 2016], 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 article 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 main control flow. The exclusion of infrequent paths generates a simpler control flow graph, allowing better code generation with more precise data flow information and smaller JIT compilation overhead. If a pruned path were taken, then the JVM executes the path in a lazy way, similarly to the Java exception handling proposed in LaTTe [Lee et al. 2004]. 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 JVM of the OpenJDK 9 alpha release [Oracle 2016a]. 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 [2016b].

Three contributions can be found in this article. First, we analyzed the behaviour of branch bytecodes when JVM runs a non-Java language and found that it is very frequent and highly biased. Second, 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 [Grogan 2014; Gal et al. 2009; Castanos et al. 2012].

The rest of the article is organized as follows: Section 2 describes the multi-language support of JVM and the architectures of non-Java language runtimes in OpenJDK. Section 3 analyzes branch behaviours of several JVM-hosted languages with a comparison to Java, which is the key motivation for our research. We explain the main

idea and issues of exceptionization in Section 4, followed by the experimental evaluation in Section 5. Section 6 discusses related achievements that inspired our research. Conclusion and future work are given in Section 7.

2. 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.

2.1. JVM Extensions for Multi-Languages

To host non-Java languages on JVM, the JVM specification [Linkholm et al. 2014] 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 [Grogan 2006]. 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 [Rose 2009, 2011]. 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 to support dynamic typing only with those instructions.

On the other hand, `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 at the bytecode level from Java SE 8 to create the invocation stub named `LambdaForm` [Duke and Rose 2013] on the fly that is executed per `invokedynamic` by JVM.

As a result, many JVM-based language implementations have emerged, such as JRuby for Ruby [jRuby.org 2016], Jython for Python [Wierzbickis 2015], Clojure [Hickey 2008], and Scala [Odersky et al. 2016].

2.2. Nashorn JavaScript Engine on HotSpot JVM

We adopted OpenJDK 9 alpha-release, the reference implementation of upcoming Java SE 9, as the base research environment [Oracle 2016a]. Figure 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 [Palczny et al. 2001]. 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 the help of the profile information from the interpreter and the C1-compiled methods.

Since Java SE 8 was released, OpenJDK has been accompanied by a JavaScript engine called Oracle Nashorn [Ponge 2014], compliant with the ECMA normalized JavaScript specification [ECMA 2011]. Nashorn deploys its own dynamic compilation system exploiting JSR 292 features from the underlying HotSpot VM and supports

Fig. 1. Nashorn on top of HotSpot in OpenJDK.

JSR 223 for interoperability with Java. It compiles JavaScript directly into bytecode with type specialization or generates bytecode selectively after interpretation with profiling.¹ Nashorn utilizes a generic abstract syntax tree (AST) representation and can be morphed into a front-end for other programming languages on HotSpot.

The JavaScript application in Figure 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 a type specialization technique called *optimistic typing* [Grogan 2014] to a generic JavaScript function, 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 Nashorn determines that 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

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

`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 the built-in Nashorn methods, `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 application and internally-used calls to runtime methods are translated with `invoke-dynamic`. 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 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 the 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 gradually propagates to HotSpot. The hot JavaScript functions identified by Nashorn also need some time to be also recognized as hot by HotSpot, leading to more complex JVM-level phase transitions, which can be named *gradual propagation*.

2.3. Other Non-Java Languages on HotSpot JVM

In this article, we will also discuss other JVM-hosted languages, such as Ruby and Python. JRuby 9.0.5.0 [jRuby.org 2016] 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 [Wierzbickis 2015] is a Python 2.7 implementation atop JVM. It deploys a simple JIT compiler with support of JSR 223. However, Jython does not utilize `invokedynamic`.

3. BRANCH BYTECODE BEHAVIOURS OF NON-JAVA LANGUAGES

Several researchers have discovered that runtime behavioral characteristics of non-Java languages on JVM differ from those of Java [Li et al. 2013; Sarimbekov et al. 2013]. 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 that 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 is shown in Figure 1. It usually combines the JIT compilers with the interpreters in an arbitrary manner and changes the execution sequence, modes of JavaScript functions, and bytecode methods at every run, leading to unpredictable performance fluctuation, not observable in traditional Java runs.

3.1. Branch Bytecode Statistics

We analyzed the branch bytecode behaviours of Java and three non-Java dynamic languages, including JavaScript, Ruby, and Python, to evaluate the control flow complexity in the JVM level by measuring branch bytecodes translated by the HotSpot C2 compiler, that is, branch instructions in hot methods. The Computer Language Benchmarks Game (CLBG) benchmark suite [Debian.org 2016] was chosen for the analysis,

Table I. C2 Compilation and Branch Bytecodes on CLBG

	Java	JavaScript	Ruby	Python
Execution Time	39.58s	155.41s (3.9×)	735.15s (18.6×)	1568.96s (39.6×)
# of C2 Compilations	38	1,370 (36.1×)	2,715 (71.5×)	1,935 (50.9×)
# of C2 Comp'd Branches	195	15,566 (79.8×)	27,572 (141.4×)	25,618 (131.4×)
C2 Comp. / min	57.61	528.94 (9.2×)	221.59 (3.9×)	74.00 (1.3×)
Branches / C2 Comp.	5.13	11.36 (2.2×)	10.16 (2.0×)	13.24 (2.6×)

Table II. C2 Compilation and Branch Bytecodes on Octane and SPECjvm2008

	JavaScript (Octane)	Java (SPECjvm2008)	Ratio (JavaScript/Java)
Execution Time	2min 51s	123min 23s	0.02
# of C2 Compilations	5844	4410	1.33
# of C2 Comp'd Branches	353,324	94,952	3.72
C2 Comp. / min	2050.5	35.7	57.44
Branches / C2 Comp.	60.46	21.53	2.81

Fig. 2. Bias distribution of branch bytecode execution frequency in C2-compiled methods (CLBG).

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 [Google 2016]) and Java (SPECjvm2008 [SPEC 2015]) 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.² Table I presents statistics of C2 compilation and branch bytecodes on CLBG for the four languages and Table II on Octane and SPECjvm2008.

The execution time, the number of C2 compilations, and the C2 compilation frequency in Table I can be an indirect indicator of the language runtime implementation complexity. Although the implementations for the three languages are equipped with JIT compilers, JavaScript has the most powerful and sophisticated implementation. The more sophisticated language runtime, the higher the frequency of C2 compilation.

Figure 2 and Figure 3 illustrate 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).³ Figure 4 also depicts the distribution histogram 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 2 and Figure 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 counts Java Library. JS Runtime/Library and JS

²We used binarytrees, fannkuchredux, fasta, mandelbrot, nbody, and spectralnorm among the micro-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.

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

Fig. 3. Bias distribution of branch bytecode execution frequency in C2-compiled methods (CLBG).

Fig. 4. Bias distribution of branch bytecode execution frequency in C2-compiled methods.

Compiler correspond to branches in the Nashorn engine. Finally, Dynamic Call counts branches in the invocation stubs of LambdaForm to implement invokedynamic.

3.2. 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 I and Table II 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 [Gal et al. 2009; Castanos et al. 2012; Kedlaya et al. 2013]. As illustrated in Figure 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 the JS Runtime/Library, and the LambdaForm stubs in Dynamic Call by the Nashorn engine.

Figure 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 4 shows the similar characteristic for the bigger benchmarks. We can conclude that JVM encounters many branches when running non-Java languages in spite of various optimizations 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 2 and Figure 3. Figure 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.

Fig. 5. C2 compilations over time (Octane).

Fig. 6. C2 compilations over time (SPECjvm2008).

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. Of the branches, 48% are biased above 5:9995 on Octane, while it is 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 I 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 II 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 JavaScript function.⁴

Figure 5 and Figure 6 depict C2 compilation counts over time for Octane and SPECjvm2008. The size of the time slot has been normalized to each other for lifetime comparison.

The life of Octane can be divided into three phases in Figure 5, which confirms the gradual propagation explained in Section 2. The first warm-up phase to 20s 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 40s to 85s, APP is frequently translated from

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

bytecode to machine code as 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 are being translated by the C2 compiler directly. After 85s, the system migrates into the saturation 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 6 shows that the Java run consists of the standard warm-up and saturation phases with additional two bursts. The first burst, from 1,250s to 2,250s, comes from the intentional benchmark execution to make JVM and the benchmark suite saturated before performance measurement. The other burst, from 6,000s to 7,000s, are needed to compile the SPEC report framework, including the Abstract Window Toolkit (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. 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 the context of our research, this complex JVM state transition introduces unstable branches that will be discussed later.

4. EXCEPTIONIZATION

4.1. Idea of *Exceptionization*

Java exception is a language construct to handle erroneous events, which do not happen normally, in a structured way [Gosling et al. 2014]. 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 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 [Kotzmann et al. 2008] or compiled on demand to run as native [Lee et al. 2004].

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 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, setting 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 three basic blocks with 13 instructions in (d), whereas it contains eight basic blocks with 19 instructions in (c). It is notable

```

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) Exceptionized machine code.

Fig. 7. Exceptionization on a sample code segment.

that the conditional 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) [Muchnick 1997]. EBBs widen the flow analysis scope for the compiler to generate better-quality code, effectively negating the complicated control flows of JavaScript. We can view exceptionization as another type of the control merge splitting [Chambers and Ungar 1990] or tail duplication [Hwu et al. 1993] with no actual basic block cloning. The actual basic block cloning will happen during de-exceptionization to be explained in Section 4.4.

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, both of which are beneficial to the non-Java language performance [Sarimbekov et al. 2013].

4.2. Selection of Exceptionization Targets

We chose to integrate exceptionization only with the HotSpot C2 compiler. The C2 compiler is supplied with sufficient branch profile information from the interpreter and

```

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);
    }
}
}

```

Fig. 8. Pseudo code for *exceptionization*.

the C1-compiled methods so the branch behaviour speculation used by exceptionization becomes practically precise. Moreover, aggressive optimizations in the C2 compilation can exploit better the widened EBBs due to exceptionization.

Figure 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 intermediate representation 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 while other branches are being de-optimized or if the compiler determines not to exceptionize for some reason, such as an explicit turn-off runtime option.

4.3. Branch Exception Handling

In our implementation, the branch exception handling utilizes the uncommon trap and the deoptimization of HotSpot [Paleczny et al. 2001] 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 a 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 speculation fails and the compiled code becomes invalid, then the trap handler eventually triggers recompilation of the method to correct the failed speculation. Currently, HotSpot generates 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) with one or more interpreter stack frames (interpreter frames) and

Fig. 9. Branch exception handling.

continues the method execution in the interpretation mode.⁵ Thus, uncommon traps, as well as On-Stack Replacement (OSR) transitions of methods are always accompanied by deoptimizations.

Figure 9 explains our branch exception handling. The (*or other differentiating type*) *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 timestamp per exceptionized branch. The handler may trigger another action called de-exceptionization instead of interpretation if necessary.

4.4. De-Exceptionization

Interpretation is the default method to run an exceptionized code that is the pruned target of an exceptionized branch. So 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 the 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 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.

4.4.1. How to De-Exceptionize a Branch. Since the HotSpot JIT compilers use a particular 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 starts to be used for the following method invocations. This approach is seamlessly pluggable to the original HotSpot, so our current implementation uses this approach as the default option.

One possible drawback of this approach is that the benefits discussed in Figure 7 and the resources invested for exceptionization are all wasted after de-exceptionization. So we implemented another approach where 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

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

Fig. 10. De-exceptionization with partial method compilation.

```

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;
}

```

Fig. 11. Pseudo code to trigger de-exceptionization.

control transfer to the partially compiled branch target from the branch, as depicted in Figure 10.

Method inlining may create 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 a 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 7(d) that correspond to Block F to and to Block J in Figure 7(c) may be translated twice with the effect of tail duplication [Hwu et al. 1993] if the branch is de-exceptionized.

If a method contains too many exceptionized branches and most 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.

4.4.2. When to Trigger De-Exceptionization. By default, HotSpot invalidates the method immediately and then triggers a 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 a more patient policy for de-exceptionization in order to keep using the exceptionized code before the invalidation, which was found more effective by experiments presented in Section 5.

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 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.

5. EXPERIMENTAL RESULTS

Exceptionization has been implemented 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 JavaScript⁶ because of the optimizations on the JavaScript library, the optimistic typing, and the execution model change [Grogan 2014; Lagergren 2015]. The implementation was compiled by gcc 4.8.4 and 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.

We measured the performance impact of exceptionization on JavaScript, Ruby, Python, and Java. The JavaScript performance was evaluated with the Octane benchmark suite version 2.0 from Google [2016] and the Kraken benchmark suite version 1.1 from Mozilla.org [2010] with Nashorn as the language runtime. JRuby 9.0.5.0, a Ruby implementation on JVM [jRuby.org 2016], was used for Ruby with the publicly available performance benchmark suite [Cangiano 2016]. The macro benchmarks in the official benchmark suites for Python.org [2016] were experimented with Jython 2.7, a Python implementation [Wierzbickis 2015]. SPECjvm2008 [SPEC 2015] and the DaCapo benchmark suite [DaCapo 2009] were chosen for the Java performance measurement.

We extracted an arithmetic mean from the performance numbers generated through 100 separate JVM runs of each benchmark suite with no benchmark modification, which differs from the Oracles approach via the JMH (Java Microbenchmark Harness) framework requiring the dissection of a benchmark suite [Lagergren 2013]. 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, whereas it was 1% for Java.

The relatively large performance fluctuation of the dynamic languages is mainly because of the multi-thread and two-layered architecture, as shown in Figure 1, which changes the execution order and JIT compilation sequence whenever the JVM system launches. Nonetheless, we verified that the following experimental performance results remain consistent on multiple experiment trials, meaning that the performance effects from exceptionization are independent of such performance fluctuations.

5.1. 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.⁷ Except_0005, the empirically chosen exceptionization environment uses the value of 0.0005 as the exceptionization threshold. Except_0, with the threshold value of 0, is also tested for comparison. It only exceptionizes branches that have not-yet-taken targets. Both Except_0005 and Except_0 recompile the whole method for de-exceptionization with the de-exceptionization threshold of 4.

5.1.1. JavaScript Performance. Figure 12 shows the relative performance of each configuration for Octane8 and Figure 13 that for Kraken compared to Base. We adopted performance numbers generated by each suite as the performance indicators. Score of

⁶We checked the performance with Octane for JavaScript and SPECjvm2008 for Java.

⁷The 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.

⁸The base OpenJDK9 raises an error on zlib, so we removed it from the benchmark suite.

Fig. 12. Relative JavaScript performance on Octane.

Fig. 13. Relative JavaScript performance on Kraken.

Octane and Total of Kraken are the overall performance numbers calculated by the suites.⁹ Their relative ratio practically corresponds to the geometric mean of relative performances of their member benchmarks.

Except_0005 improves the Octane performance by 6% and the Kraken performance by 5%. Except_0 achieves 8% improvement for Octane and 3% for Kraken. In order to get a statistically rigorous performance comparison [Georges et al. 2007], we calculated 95% confidence intervals of Octane Score mean value and Kraken Total mean value. Base shows the Octane Score in $(2088.53, \pm 29.39)$ and the Kraken Total in $(12344.98, \pm 35.87)$. Except_0 shows $(2252.41, \pm 26.42)$ and $(11982.34, \pm 57.93)$ for each, and Except_0005 shows $(2220.41, \pm 38.02)$ and $(11658.56, \pm 48.86)$ for each. In spite of the relatively high performance fluctuations explained previously, the confidence intervals

⁹The higher Octane Score and the lower Kraken Total indicate better performance.

Fig. 14. Relative Ruby performance.

of exceptionization do not overlap with those of Base. We hereby concluded that the performance improvements are statistically sound.

Except_0 demonstrates the consistent performance improvement over Base. It accelerates almost all benchmarks in Octane and Kraken except Box2D, beatdetection, and cryptosha256iterative.

Except_0005 significantly accelerates several benchmarks, including EarleyBoyer, Splay, SplayLatency (Octane), gaussian-blur, stanford, and crypto-pbkdf2 (Kraken). DeltaBlue, NavierSokes, Mandreel (Octane), beat-detection, oscillator, and desaturate (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 that in Except_0. However, the mechanism behaves differently when branching to the exceptionized path happening 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 the more optimized code for the method may be used longer. The evaluation of different de-exceptionization policies will be discussed in Section 5.4.

5.1.2. 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 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 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.

Fig. 15. Relative Python performance.

Fig. 16. Relative Java performance on SPECjvm2008.

5.2. Performance Impact on Java

Since exceptionization modifies JVM itself, its performance impact on Java was tested with SPECjvm2008.¹⁰ Figure 16 indicates that exceptionization has no negative performance impact on SPECjvm2008 by and large, which was predicted in Section by the observation that Java applications have relatively fewer branches to exceptionize than those of JavaScript. 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 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.

The highly accelerated benchmarks through exceptionization such as EarleyBoyer, Splay, gaussian-blur (JavaScript), parse_log (Ruby), and fop, jython (Java) are common in containing more complex control flows. EarleyBoyer, parse_log, fop, and jython have parse logics with many branches. Splay also uses many branches to implement the splay tree. For such benchmarks, exceptionization may provide a better opportunity for performance improvement by exploiting the branch behavioral characteristics.

Furthermore, since Nashorn and JRuby deploy better JIT compilers than Jython as discussed in Section 3, we speculate that the more sophisticated a language runtime,

¹⁰ Among the official benchmarks in SPECjvm2008, compiler.compiler and compiler.sunflow do not run on Java SE 8 and later versions, because the benchmark uses APIs that became invalid due to the language specification change.

Fig. 17. Relative Java performance on DaCapo.

Fig. 18. JavaScript performance with different thresholds relative to Except_0005.

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 that contain many branches.

5.3. Performance Variation by Exceptionization Threshold

Figure 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 with the observation of branch bias distribution in Figure 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.

5.4. Performance Impact from De-Exceptionization

Figure 19 evaluates the performance impact of various de-exceptionization policies based on Except_0005 with Octane. De-except 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.

Fig. 19. Relative JavaScript performance with different de-exceptionizations.

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 in recompilation overhead and slightly poorer-quality code. In any case, it is such a suitable approach for Java with stable branches that a similar policy is effective in the base HotSpot JVM.

PartComp shows performance improvement for several benchmarks, including NavierStokes and Mandreel, over the default Except_0005, although the overall performance gain is not yet satisfactory. So, more deliberate de-exceptionization techniques coupled with the partial method compilation may provide an 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 III.¹¹ As expected, Except_0005 tends to generate more branch exceptions than Except_0. De-except and Invalid reduce branch exceptions significantly due to de-exceptionization when compared to Interpret. Since Invalid is the most rigid to branch exceptions, it raises the least number of exceptions.

PartComp generates the moderate number of branch exceptions, because jumps to the partially compiled code as shown in Figure 10 are counted as a kind of branch exception. Too many branch exceptions can be a performance burden,¹² while too few exceptions may blow away the possible performance gain from exceptionization. From the result in Figure 19, finding heuristics to control the raised branch exception number will be one of our future research topics.

The large performance loss of Interpret and a large number of raised exceptions are still surprising with knowledge that a not-yet-taken path tends not to be taken in the future. Table IV 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

¹¹Except_0 with Invalid is virtually identical to Base. It cuts off a not-yet-taken path and invalidates the resulting compiled method when the path is taken.

¹²The branch exception handling takes 22 μ s in our experimental environment.

Table III. 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
Mandrel	88	42	6255	1083826	56	22	4973	630910
MandrelLatency	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

Table IV. *Exceptionizable* Branches in JavaScript and Java Encountered During Compilation

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

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 remain so in SPECjvm2008. This confirms the fourth observation in Section 3 that JavaScript shows more unstable behaviours than Java. The gradual propagation may bring about 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 IV also indicates that there is not a small number of unstable and exceptionized branches eligible to de-exceptionization in JavaScript, which accounts for the many branch exceptions of PartComp and Interpret in Table III.

5.5. 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 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 the exceptionized branches come from the language runtime. The comparison between JavaScript and Java reveals that exceptionizations simplifying control flows of the JavaScript runtime lead to the performance improvement.

As described previously, the optimistic typing in OpenJDK 9 significantly reduces branches via type specialization, which explains the relatively small portion of APP.

Fig. 20. Origin distribution of exceptionized branches in JavaScript and Java.

Mature language runtimes, equipped with such aggressive specializations, can benefit from exceptionization through the 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.

6. RELATED WORK

The SELF-91 compiler introduced deferred compilation [Chambers and Ungar 1991] to generate code for the uncommon branch targets lazily. If an uncommon branch target is taken, then 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 type analysis to predict type checking branches and virtual method invocations for lazy code generation. The SELF-91 compiler also introduces a code replication technique called extended message splitting to split control flow merges for more precise type analysis [Chambers and Ungar 1990].

HotSpot [Paleczny et al. 2001] 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 by invalidating speculatively optimized code. Additionally, the HotSpot C2 compiler generates both fast and slow paths to translate virtual calls and type checks in bytecode with profiled type information in a similar way to the message splitting in the SELF-91 compiler.

Exceptionization depends solely on the branch profile information provided by JVM, so it may collaborate with the deferred compilation scheme orthogonally. It has a path splitting effect by removing an incoming path to a control merge by delaying the code replication until de-exceptionization if a branch is highly biased.

Section 5 shows that exceptionization improves the performance more in the presence of deferred compilation and path splitting by HotSpot. HotSpot prunes a not-yet-taken branch target with an assumption that the path will not be reached, which works well for Java, but 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.

The partial method compilation [Whaley 2001] and the region-based compilation [Suganuma et al. 2006] 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 [Gal et al. 2009]. 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 trace compilation.

The technical debate regarding which approach is better between the method-based JIT and the partiallycompiling trace-based JIT has not yet been settled. We consider exceptionization as a hybrid approach to extract the benefit of the trace-based JIT in the method-based JIT environment by pruning infrequent paths of branches during method compilation.

Jikes RVM [Fink and Qian 2003] adopted an extended version of the HotSpot uncommon trap that is streamlined with the OSR 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 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 [Lee et al. 2004]. It ignores any exception handlers in a method while JIT is 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 [Castanos et al. 2012]. 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 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 [Würthinger et al. 2013]. 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 1.

Exceptionization could be integrated into the framework. However, we wanted to evaluate the effect of exceptionization in a product-level JVM, with no dependence on guest language implementation.

7. CONCLUSION AND FUTURE WORK

Since a non-Java language on JVM requires a corresponding language runtime as the intermediate layer imposing more bytecode to execute, JVM should confront more complex control flows to support alien language semantics and to enable sophisticated language optimizations. We found that the JVM executing non-Java languages encounters twice more bytecode branches than Java, most of which are highly biased, from the experimental analysis on both micro-benchmarks and macro-benchmarks of multiple languages.

This article proposed exceptionization, a JVM-level optimization to boost the performance of non-Java languages on JVM. Exceptionization transforms a highly biased branch to an implicit exception-throwing instruction by pruning the infrequent target from control flow by the JVM JIT compiler. The JIT compiler generates better-quality code thanks to the more accurate data flow analysis on the simplified control flow. We described how to handle a branch exception raised when the pruned target is taken, which relies on the interpretation. De-exceptionization is streamlined with the branch exception handling in order to eliminate possible interpretation 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 and DaCapo shows that exceptionization is transparent to Java performance. Other non-Java languages on JVM, including Ruby and Python, are also benefited by exceptionization.

Although there exist preceding works to deal with common cases and uncommon cases separately [Chambers and Ungar 1990, 1991] that have been incorporated into HotSpot, exceptionization can extract more opportunities to improve performance by exploiting the branch behaviors, which is useful as a JVM optimization for non-Java languages. We discovered that a language runtime (engine) creates highly biased branches so massively that the major performance improvement of exceptionization originates from the better handling of the language runtime by the JVM JIT compiler. This indicates that the careful handling of a language runtime is as important as integrating various optimization techniques into the language runtime for the overall performance.

Some highly biased branches were found to be so unstable, meaning that their actual behaviours differed from predictions based on the profile information, that they ruined the benefit from exceptionization during non-Java executions. Therefore, meticulous de-exceptionization mechanisms become important for JavaScript performance. As a future work, we plan to upgrade the de-exceptionization mechanism based on the partial compilation along with a more sophisticated exceptionization target selection algorithm to deal with the gradual propagation of non-Java language runtimes more efficiently. We will also extend the concept of exceptionization to other bytecode instructions, such as calls and type-checks, which tend to have highly biased implicit control transfers.

REFERENCES

Antonio Cangiano. 2016. Ruby Benchmark Suite. Retrieved May 13, 2016 from <https://github.com/acangiano/ruby-benchmark-suite>.

- Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. 2012. 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 (OOPSLA'12)*. ACM Press, New York, NY, 195–212. DOI: <http://dx.doi.org/10.1145/2384616.2384631>
- Craig Chambers and David Ungar. 1990. 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 (PLDI'90)*. ACM Press, New York, NY, 150–164. DOI: <http://dx.doi.org/10.1145/93542.93562>
- Craig Chambers and David Ungar. 1991. Making pure object-oriented languages practical. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*. ACM Press, New York, NY, 1–15. DOI: <http://dx.doi.org/10.1145/117954.117955>
- DaCapo. 2009. The DaCapo Benchmark Suite. Retrieved October 13, 2016 from <http://dacapobench.org>.
- Debian.org. 2016. The Computer Language Benchmarks Game. Retrieved April 23, 2016 from <http://benchmarksgame.alioth.debian.org>.
- J. Duke and John Rose. 2013. OpenJDK Wiki: Method handles and invokedynamic. Retrieved May 9, 2016 from <https://wiki.openjdk.java.net/display/HotSpot/Method+handles+and+invokedynamic>.
- ECMA. 2011. *ECMAScript Language Specification, Standard ECMA-262* (5.1 ed.). Ecma International, CH-1204 Geneva. Retrieved from <http://www.ecma-international.org/ecma-262/5.1/>.
- Stephen J. Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'03)*. IEEE Computer Society, Washington, DC, 241–252.
- Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI'09)*. ACM Press, New York, NY, 465–478. DOI: <http://dx.doi.org/10.1145/1542476.1542528>
- Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the 2007 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM Press, New York, NY, 57–76. DOI: <http://dx.doi.org/https://doi.org/10.1145/1297027.1297033>
- Google. 2016. Octane: The JavaScript Benchmark Suite for the modern web. Retrieved April 23, 2016 from <https://developers.google.com/octane/>.
- James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Brach, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley, Boston, MA.
- Mike Grogan. 2006. JSR 223: Scripting for the Java Platform. Retrieved April 23, 2016 from <https://jcp.org/en/jsr/detail?id=223>
- Mike Grogan. 2014. JEP 196: Nashorn Optimistic Typing. Retrieved April 23, 2016 from <http://openjdk.java.net/jeps/196>.
- Rich Hickey. 2008. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS'08)*. ACM Press, New York, NY, Article 1. DOI: <http://dx.doi.org/10.1145/1408681.1408682>
- Wen-Mei Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John E. Holm, and Daniel M. Lavery. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomput.* 7, 1–2 (May 1993), 229–248. DOI: <http://dx.doi.org/10.1007/BF01205185>
- jRuby.org. 2016. JRuby: The Ruby Programming Language on the JVM. Retrieved April 23, 2016 from <http://www.jruby.org/>.
- Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. 2013. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS'13)*. ACM Press, New York, NY, 37–48. DOI: <http://dx.doi.org/10.1145/2508168.2508177>
- Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russel, and David Cox. 2008. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Arch. Code Optimiz.* 5, 1, Article 7 (May 2008). DOI: <http://dx.doi.org/10.1145/1369396.1370017>
- Marcus Lagergren. 2013. OpenJDK Wiki: Monitoring Nashorn Performance. Retrieved April 23, 2016 from <https://wiki.openjdk.java.net/display/Nashorn/Monitoring+Nashorn+Performance>.

- Marcus Lagergren. 2015. Nashorn for java 9: Quickly bootstrapping nashorn with a new execution model. In *JVM Language Summit 2015 (JVMLS'15)*. Retrieved from <http://www.slideshare.net/lagergren/a-new-execution-model-for-nashorn-in-java-9>.
- SeungIl Lee, Byung-Sun Yang, and Soo-Mook Moon. 2004. Efficient java exception handling in just-in-time compilation. *Softw.: Pract. Exper.* 34, 15 (Dec. 2004), 1463–1480. DOI : <http://dx.doi.org/10.1002/spe.622>
- Wing Hang Li, David R. White, and Jeremy Singer. 2013. 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 (PPPJ'13)*. ACM Press, New York, NY, USA, 101–112. DOI : <http://dx.doi.org/10.1145/2500828.2500838>
- Tim Linkholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley, Boston, MA.
- Mozilla.org. 2010. Kraken JavaScript Benchmark (version 1.1). Retrieved April 23, 2016 from <http://krakenbenchmark.mozilla.org/>.
- Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, New York, NY. 175–175 pages.
- Martin Odersky, Lex Spoon, and Bill Venners. 2016. *Programming in Scala* (3rd ed.). Artima Inc, Walnut Creek, CA.
- Oracle. 2016a. OpenJDK: JDK 9. Retrieved April 23, 2016 from <http://openjdk.java.net/projects/jdk9/>.
- Oracle. 2016b. OpenJDK: Project Nashorn. Retrieved April 23, 2016 from <http://openjdk.java.net/projects/nashorn/>.
- Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot™ server compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology (JVM'01)*. USENIX Association, Berkeley, CA, Article 2.
- Julien Ponge. 2014. Oracle nashorn: A next-generation javascript engine for the JVM. *Java Mag.* (Jan. 2014), 59–65.
- Python.org. 2016. Python Benchmarks by Python Software Foundation. Retrieved May 13, 2016 from <https://hg.python.org/benchmarks/>.
- John R. Rose. 2009. Bytecodes meet combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages (VMIL'09)*. ACM Press, New York, NY. DOI : <http://dx.doi.org/10.1145/1711506.1711508>
- John R. Rose. 2011. JSR 292: Supporting Dynamically Typed Languages on the Java Platform. Retrieved April 23, 2016 from <https://jcp.org/en/jsr/detail?id=292>
- Albek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. 2013. Characteristics of dynamic JVM languages. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL'13)*. ACM Press, New York, NY, 11–20. DOI : <http://dx.doi.org/10.1145/2542142.2542144>
- SPEC. 2015. SPECjvm2008. Retrieved April 23, 2016 from <https://www.spec.org/jvm2008/>.
- Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2006. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 1 (Jan. 2006), 134–174. DOI : <http://dx.doi.org/10.1145/1111596.1111600>
- Raoul-Gabriel Urma. 2014. Alternative languages for the java virtual machine. *Java Mag.* (July 2014), 5–11.
- John Whaley. 2001. Partial method compilation using dynamic profile information. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*. ACM Press, New York, NY, 166–179. DOI : <http://dx.doi.org/10.1145/504282.504295>
- Frank Wierzbickis. 2015. Jython: Python for the Java Platform. Retrieved April 23, 2016 from <http://www.jython.org/>.
- Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM Press, New York, NY, 187–204. DOI : <http://dx.doi.org/10.1145/2509578.2509581>

Received June 2016; revised December 2016; accepted January 2017