JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Practical Validation of Bytecode to Bytecode JIT Compiler Dynamic Deoptimization

Clément Béra[a] Eliot Miranda[b] Marcus Denker[a]
Stéphane Ducasse[a]

a. RMOD - INRIA Lille Nord Europe

b. Cadence Design Systems

Abstract

Speculative inlining in just-in-time compilers enables many performance optimizations. However, it also introduces significant complexity. The compiler optimizations themselves, as well as the deoptimization mechanism are complex and error prone. To stabilize our bytecode to bytecode just-in-time compiler, we designed a new approach to validate the correctness of dynamic deoptimization. The approach consists of the symbolic execution of an optimized and an unoptimized bytecode compiled method side by side, deoptimizing the abstract stack at each deoptimization point (where dynamic deoptimization is possible) and comparing the deoptimized and unoptimized abstract stack to detect bugs. The implementation of our approach generated tests for several hundred thousands of methods, which are now available to be run automatically after each commit.

1 Introduction

High-level object-oriented programming languages such as C# and Java are commonly implemented on top of a Virtual Machine (VM). The VM optimizes the code it runs at runtime using information about the current execution state that cannot be determined statically. This approach allows VMs such as Java's HotSpot VM [PVC01] or .NET's CLR [ECM01] to reach high performance.

One interesting aspect of such VMs is their deoptimization capabilities [HCU92]. In these VMs, optimizations such as inlining and dead branch elimination are based on assumptions derived from the types encountered in the program so far; assumptions which may be invalidated as the program continues to run. When one of these assumptions is invalidated, the VM dynamically deoptimizes the runtime stack, continues execution and subsequently reoptimizes under new assumptions.

Such high performance virtual machines are complex artifacts, difficult to stabilize. To guarantee the stability of a program, a developer usually writes large suites of tests. As dynamic deoptimization relies on the runtime history and optimized code which are unavailable statically, writing such test suites to validate dynamic deoptimization is difficult.

In this paper we present a static approach that automatically generates tests to validate dynamic deoptimization for a given set of bytecode compiled methods. Our approach consists in the symbolic execution of both optimized and unoptimized versions of the compiled method's bytecode, stopping the execution at each point where dynamic deoptimization is possible and comparing the deoptimized and unoptimized stack of abstract values to guarantee that each value has been correctly deoptimized. Therefore, we can turn every executable method into a test using this approach. The paper focuses on the dynamic deoptimization of speculative inlining, dead branch elimination and array bounds check removal [BGS00].

The paper makes the following contribution: we introduce a technique to generate automatically tests to validate dynamic deoptimization for a given set of methods using symbolic execution. The technique has been validated on our bytecode to bytecode runtime optimizer [1]. The approach has been used to support agile development and is currently used on 160 000 methods.

To make the paper easy to understand, the background and the solution sections are described using Java, then the validation is done on a Smalltalk runtime. All the paper could have been written using Smalltalk but fewer people are familiar with Smalltalk syntax, so we used Java to make the paper more accessible.

The first section describes how a just-in-time compiler optimizes and deoptimizes methods. Then we discuss why it is difficult to generate tests to validate dynamic deoptimization and detail our solution. Lastly, we present our implementation, which validates our approach by generating several hundred thousands tests.

2 Background

In this section we describe the context of our work. We will reuse the example shown to explain how we apply dynamic deoptimization in Section 4, explaining how it is optimized. The reader who is well aware of these optimizations may skip this section.

2.1 Java classes as examples

A JIT compiler optimizes a method at runtime based on its previous executions. It performs many different optimizations, however, in this paper, we focus on a subset of optimizations: method call inlining, dead branch elimination and array bounds check removal. To show how this subset of JIT compiler optimizations works, we define two Java classes. The class MyClass has three methods. Among them, the method named example is the one the JIT compiler is going to optimize. This method has a loop, a condition and two virtual calls as shown in Figure 1. In addition, the class Printable defines two methods called by the methods of MyClass. We assume the subclass MySubClass of MyClass has been loaded for the examples.

To perform optimizations, the JIT compiler uses runtime information gathered from first method executions. Let's run for example the main method shown at the end of Figure 1. In this example, the runtime type information for the method informs the JIT compiler that:

- The condition in the method example has always branched on true (the false branch has never been taken and no exception has ever been raised on null).

[1]We detail in the validation section why we use a bytecode to bytecode optimizer.

```
public class MyClass {
    public example(boolean bool, Printable[] array) {
      int i;
      for (i = 0; i < array.length; i++) {
          if (bool) {
              this.printlnElement(array[i]); }
          else {
              this.printElement(array[i]); } } }
protected printlnElement(Printable printable) {
          printable.printlnOnConsole(); }
   protected printElement(Printable printable) {
       printable.printOnConsole(); } }
public class MySubClass extends MyClass { /* ... */ }
public class Printable {
    public printlnOnConsole() {
       System.out.println(this); }
    public printOnConsole() {
       System.out.print(this); } }
```

Figure 1 – Java classes as examples

- The object corresponding to this is always an instance of MyClass (it could be an instance of one of the subclasses of MyClass, but it has never been the case).

```
public static void main (String[] args){
   MyClass guineaPig = new MyClass();
   Printable[] t4= Printable.generateExamplePrintable();
   for (i = 0; i < 99999; i++) {
      guineaPig.example(true, toPrint); } }
```

Figure 2 – Example of a main

2.2 Dead branch elimination

One optimization of the JIT compiler consists in removing branches that are never used. In the example, the runtime type information informs the JIT compiler that only the true branch has been taken in the condition. Therefore, the JIT compiler removes the false alternative for the condition on bool as shown on Figure 3. However, the optimization is *speculative*, the optimizing compiler assumed that bool is always true because it has always been true up until now. In the future, it may happen that the application runs a method that has never been called until now that calls the method example with false as a value for the boolean. Therefore, the JIT compiler adds a guard. The guard is a runtime check to validate the assumptions taken during the method optimizations. If the guard fails, the virtual machine needs to stop using the optimized method. This will be discussed in section 2.4.

As guards are present in very frequently used methods, they need to be very efficient to execute in machine code. In the figures, we use the message guard to represent a machine code guard. This representation is here to explain the behavior of the optimized method and is not valid Java code.

```
public optimizedExample0(boolean bool, Object[] array) {
    int i;
    for (i = 0; i < array.length; i++) {
        guard(bool, true);
        this.printlnElement(array[i]); } }
```

Figure 3 – The Java method after dead branch elimination

2.3 Speculative inlining

Another optimization JIT compilers perform is inlining. Inlining consists in replacing a method call by the body of its callee. In the example, we know from the runtime information that this is always an instance of MyClass. Therefore, the virtual call to the method name printlnElement always finds the method printlnElement implemented in MyClass. The JIT compiler inlines the method call printlnElement, and inlines again in this method call the method printlnOnConsole called on the printable object as shown in Figure 4. With dead branch elimination, the optimization is speculative. Even if this has never happened, the method may be called later in one of the subclass of MyClass which has redefined the method printlnElement, or an object in the Printable [ ] array may be null, invalidating the assumption taken for this optimization. So the optimizing compiler adds guards to cover uncommon cases.

```
public optimizedExample1(boolean bool, Object[] array) {
    int i;
    for (i = 0; i < array.length; i++) {
       guard(bool, true);
       guard(this, MyClass);
       guard(array[i], Printable);
       System.out.println(array[i]); } }
```

Figure 4 – The Java method after speculative inlining

2.4 Failing guards and dynamic deoptimization

The two previous optimizations added guards to the generated code. If one of the guard fails, the virtual machine has to run the method as if the method was not optimized. Therefore, it has to stop using the optimized version of the method and reuse the unoptimized version, as explained by Urs Hölzle *et al.,* [HCU92]. This is difficult as the runtime is not in the same state if it uses the optimized or the non optimized method.

Stack changes. The runtime stack is not in the same state when running optimized and regular methods: for instance the number of stack frames may be different. In our example, if one of the objects of the Printable [ ] array is null, the guard checking that the field value of the array has Printable as a class will fail. At this point, the last stack frame of the runtime stack holds the method optimizedExample, whereas the regular stack has two stack frames, one for example, and one for printlnElement because the latter was inlined. Therefore, the virtual machine needs to dynamically deoptimize the runtime stack to be able to restart the execution of the code with the non optimized methods as shown on Figure 5. Dynamic deoptimization includes recreating the proper stack frames as well as remapping correctly all the stack frame values such as the temporary variable values.

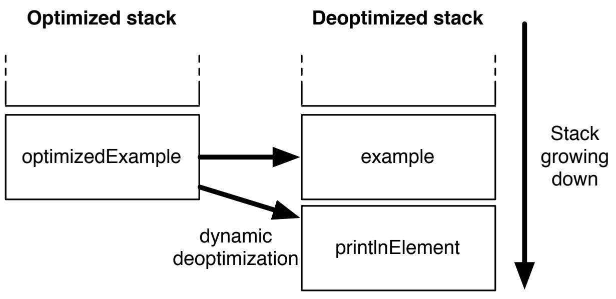


Figure 5 – Dynamic deoptimization example

Dynamic deoptimization and debugging. The virtual machine needs to be able to dynamically deoptimize the runtime stack for each guard. In addition, to support debugging features, the virtual machine may have to dynamically deoptimize the stack at other points. For example, if one puts a break point in the method println implemented in System.out, one expects the call stack shown in the debugger to have a stack frame for printlnOnConsole, printlnElement and example, as well as being able to see all the temporary variables in all the stack frames. Therefore, programming languages with advanced debugging features usually require to be able to deoptimize the stack at each call site.

On-stack-replacement points. The VM does not need to be able to deoptimize the runtime stack at any point in the code. As discussed in the previous paragraph, the VM needs to be able to deoptimize the runtime stack at a limited number of points, which includes guards, but also, depending on the language specifications, other points such as call sites. In this paper, we will use the term *on-stack-replacement point* to discuss about a point in code execution where dynamic deoptimization can be done. On-stack-replacement points limit JIT compiler optimizations, as it needs to be able to deoptimize the stack at these points.

2.5 Loop invariant code motion

A JIT compiler moves as much code as possible from inside a loop to outside of the loop as shown in Figure 6. In the example, it moved two guards away from the loop body. This allows the portion of code moved to be executed once instead of at each iteration of the loop. The JIT has to be careful when moving code around to still be able to deoptimize the stack at on-stack-replacement points. However, between two on-stack-replacement points, the JIT compiler can optimize the code as much as possible.

```
public optimizedExample2(boolean bool, Object[] array) {
   int i;
   guard(bool, true);
   guard(this, MyClass);
     for (i = 0; i < array.length; i++) {
        guard(array[i], Printable);
        System.out.println(array[i]); } }
```

Figure 6 – The Java method after loop invariant code motion

2.6 Array bounds check removal

Lastly, the optimizing compiler can find out that the variable i is always an integer within the bounds of array thanks to an algorithm such as the one described in the work of Rastislav Bodík *et al.,* [BGS00].

Usually, when accessing the field of an array, the virtual machine needs to check that the program is accessing a field within the bound of the object, else it raises an OutOfBounds exception. In our example, the integer i is always within the bounds of the array when executing array[i] due to the loop. Therefore, the bounds check is useless. In our pseudo-code, we define *[ ]* being the operation to access the field of an object without checking that the integer argument is within the bounds of the array. The optimized version of the example looks like the one shown in Figure 7.

```
public optimizedExample3(boolean bool, Object[ ] array) {
   int i;
   guard(bool, true);
   guard(this, MyClass);
      for (i = 0; i < array.length; i++) {
         guard(array*[i]*, Printable);
         System.out.println(array*[i]*); } }
```

Figure 7 – The Java method after bounds check removal

3 Challenge: testing dynamic deoptimization

When the JIT compiler optimizes the code of a method, it generates guards that force the VM to dynamically deoptimize the stack at runtime if an assumption taken during compilation is not valid anymore. In addition, the virtual machine needs to dynamically deoptimize the stack at other points than guards if the programming language specifies advanced debugging features. Dynamic deoptimization is therefore an important part of the JIT compiler infrastructure. According to the Self team, who were the first to implement dynamic deoptimization, this part was one of the most difficult part to implement of the virtual machine [HCU92].

3.1 Test infrastructure recommended

Building a test infrastructure is useful for most software projects. An optimizing compiler is a complex software and it benefits substantially from automatic testing.

Hard to stabilize a JIT compiler. Any optimizing compiler, executed ahead of time or at runtime, is hard to stabilize. In the past decade, several studies found bugs even in widely used production-quality C compilers [ER08, YCER11]. A category of bugs is composed of machine code incorrectly generated by the optimizing compiler from a valid input without raising an error. These kinds of bugs are compiled silently but generate runtime errors. Therefore they are very dangerous for production applications as described in the work of Yang *et al.,* [YCER11].

Supporting agile development. While implementing dynamic deoptimization, one wants to be able to test it as any other program. Following agile methodologies such as unit testing or test-first should be supported. As a runtime optimizer is a complex domain, one wants to be able to change the design of the optimizer or an important part of its code base without invalidating the test suite. Therefore it is important to test the optimized code versus its non optimized version but it is not possible to write a proof for each of the increment and design variations.

The compiler validation approach presented in this paper was implemented while our team was enhancing a baseline JIT compiler with adaptive optimizations. To support our agile development process, we built a test suite for day to day development as well as a large test suite on our whole system. The first suite is used to validate the code at each commit whereas the latter is used to raise our confidence level in the stability of the product before production. A large test suite running on all the methods of our system could not have been implemented without the approach described in this paper.

3.2 Existing solutions

Static analysis. Compiler developers are able to catch more bugs by validating their compilers statically. Two static analysis techniques are widely used: compiler verification [Ler06] and translation validation [STL11, PSS98, Nec00, TGM11]. Compiler verification uses a theorem prover or a proof assistant to verify that the compiler is correct for all valid sources. Compiler verification requires the verification of the compiler implementation, which is difficult for large compilers. So far, it can scale up to moderately sized compilers [Ler06]. Large compilers can be formalized to some extent [ZNMZ12], but they have not yet been proven correct and it remains a daunting task to do so. On the other hand, translation validation uses a theorem prover to check that the output of the compiler is semantically equivalent to the source program. Translation validation was used to test GCC and LLVM, which are both large compilers and widely used [Nec00, TGM11, STL11].

Other JIT compiler validation techniques. However, as far as we know, existing techniques do not validate dynamic deoptimization but only compiler optimizations. This is partly because most static validation approaches apply on ahead of time compilers, which usually do not implement dynamic deoptimization. It is very difficult to compare our approach against techniques used in other JIT compilers because, as far as we know, other JIT compiler development teams published very few articles about their testing infrastructure and we found no papers on tests for dynamic deoptimization. Therefore, it is very difficult to be aware of their testing infrastructure without being part of their development team.

A baseline x86 JIT compiler has been formally validated [Myr10], but baseline JIT compilers do not use dynamic deoptimization. The Maxine VM team took advantage of having a VM written in a high-level language to build an efficient test infrastructure [DOD12], but as far as we know their publications do not discuss tests on dynamic deoptimization. Some work has also been done to validate and test other aspects than dynamic deoptimization in a JIT compiler, such as weak mutation testing [DOD12] or reduction of the number of false alarms raised by a JIT compiler validator [HLP^{+}13].

3.3 Constraints

Bytecode to bytecode runtime optimizer. The approach presented in this paper was designed for a bytecode to bytecode runtime optimizer. Although we discuss in Section 6 how it could apply in a more generic context, the approach, in its current state, applies only for runtime bytecode to bytecode optimizers. We worked on a stack-based bytecode, but we believe the approach could also work on a register-based bytecode set.

Runtime compiler callable statically. Our approach requires the optimizing runtime compiler to feature an API to interface with an external program or to be embedded as a dynamic library: it needs an interface to be able to optimize a method given as input some precomputed runtime information.

Stable stack frame manipulation API. In this paper we do not discuss about routines, typically written in assembly code, used to edit the stack frames. Most VMs have internally a representation for stack frames with APIs to read and edit them. This kind of representation is typically used for the debugger, and can in some cases also be used for exceptions and continuations. This internal representation can be tested separately. For our approach, we assumed this part of the system was already stable - that was the case in the VM we used for our validation - and we do not discuss this part of the validation process.

3.4 Our approach

For a given method and its optimized version, we access statically the runtime stack of the methods at each on-stack-replacement point thanks to symbolic execution and compare the two stacks to validate dynamic deoptimization. On the contrary to other solutions, we validate dynamic deoptimization of our JIT compiler by comparing the runtime stack of the regular method call graph and the optimized method associated stack. We do not compare source code, machine code or analyze the compiler implementation.

4 Solution: comparing abstract stacks

We propose to test dynamic deoptimization for a large library or application based on stack comparison. Our framework iterates over a given set of available methods, and generates statically for each method an optimized version. Then, the optimized version of the method is symbolically executed and for each on-stack-replacement point met, it stops the execution and performs then successively (steps corresponding to the step numbers shown on Figure 8):

- *step 1:* the extraction of an abstract stack for an optimized method on-stack-replacement point,
- *step 2:* the deoptimization of the abstract stack obtained from the previous step,
- *step 3:* the extraction of an abstract stack for the non optimized method at the same on-stack-replacement point than the other abstract stack available, and
- *step 4:* the comparison between the two abstract stacks to validate dynamic deoptimization at the stacks' on-stack-replacement point.

We present now in detail each step of the process. For each step, we show how the process works based on the pair of method example and optimizedExample3 shown in Figure 7 and Figure 1 from Section 2.

4.1 Generating optimized methods

The approach works on a pair of methods, a non optimized method and its optimized version. However, a JIT compiler usually generates optimized methods at runtime while running a program and no optimized methods are available statically.

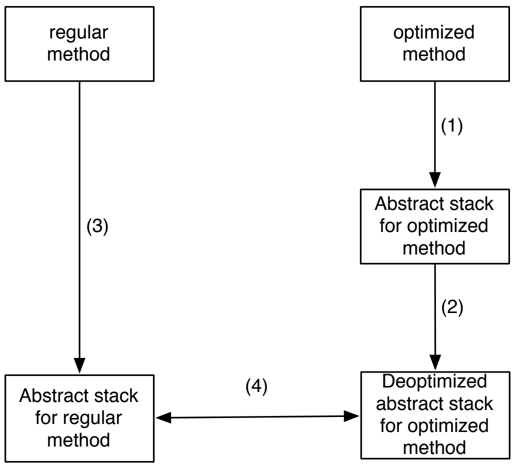

regular method
optimized method
(1)
(3)
Abstract stack for optimized method
(2)
Abstract stack for regular method
(4)
Deoptimized abstract stack for optimized method

Figure 8 – Four steps of our approach

Nowadays, many JIT compilers can be used as a library or directly from the programming language, allowing one to compile an optimized version of method statically if the JIT compiler is provided with a method and runtime type information. Virtual machines usually do not statically use the JIT compiler. However, several JIT compilers provide APIs to be used statically [RSB+14, TWSC10]. Using the JIT compiler statically is very convenient for tests and static analysis.

Even if the JIT compiler can be used statically to generate an optimized method, it needs runtime information that is usually extracted from the previous execution of the methods. We used two techniques to statically generate this information.

Recorded runtime profiles. The first technique consists in running methods of the large code base one wants to test. This can be done typically by running the test suites of the code base or by running an application using the code base. During this run, the JIT compiler saves the runtime information it generates for each method. The saved runtime information can be reused statically in our approach.

Saving runtime information is a work that has already been done by other teams [AWR05]. In their case the runtime information was reused to improve the JIT compiler performance across several runs.

This technique generates valid runtime information as the code is really run. The difference between the generated information and the real production runtime information depends on how well the code run to generate the runtime information mirrors the production application runtime. Recording the information directly from the application deployed in production may give accurate runtime information, but only a subset of the methods may be in practice run often enough to generate runtime information. Recording the information from a test suites may provide runtime information on all the methods if the test suite has a measured 100% code coverage, but the runtime information from the tests may be different from the production runtime information.

Generating fake runtime type information. Alternatively, one can run a type inferencer that generates deterministically approximate runtime information on a closed system.

We built a basic runtime information generator using type inference with two main properties:

- configurable: according to several settings we can force it to generate only metadata for inlinable call sites, non-inlinable call sites, dead branches, branches equally used.
- deterministic: on a closed world and with given configuration settings, the metadata generator will generate exactly the same metadata. This is very convenient to reproduce and debug compiler issues.

In this alternative, the generated runtime information may be different from the real runtime information but it can be generated without running any code or tests.

4.2 Generating the abstract stacks for the optimized method

Now that we have a pair of a method and its optimized version, we need runtime stacks to compare.

Running all methods is difficult. To access the runtime stack, one needs to execute the code. However, although running all the given methods is possible for a limited set of methods where the user can specify a set of receiver and arguments compatible with the methods, this is not possible to run any method from the complete code base without the right arguments and receiver. In addition, a method may trigger a system call or perform a primitive operation that can have an unpredictable behavior if run on an incorrect object.

Symbolic execution. As we can hardly run all the methods of our system with randomly generated arguments, we decided to symbolically execute the methods. The idea behind symbolic execution is to infer information on programs by executing them using abstract values rather than concrete ones, thus obtaining safe approximations of the behavior of the program. The symbolic execution can be implemented in a way that it is simple to:

- interrupt the execution,
- restart the execution,
- inspect the active abstract stack.

Abstract stack initialization. For our use case, the abstract stacks used for symbolic execution are always initialized with a *flagged stack frame*. This stack frame is used to mark the based stack frame so when the code execution returns to it, the symbolic execution knows it has finished to execute the method. In addition, the flagged stack frame holds the abstract receiver and the correct number of abstract arguments to be able to activate the method we want to run. All its other slots are filled with flags. The first instruction executed symbolically is always a method call to activate the method we want to analyze. The abstract values used for abstract arguments and the abstract receiver are values of a specific set of classes. When the symbolic executor attempts to run a method or a closure which is not inlined, if all values are known, for example they are all literals, it computes the result, else it builds a tree representing the code executed and answers this new abstract value. The nodes in the trees are either abstract or concrete values.

For example, if we want to symbolically execute the method example shown on Figure 1, the default abstract stack has a flagged stack frame holding the receiver and arguments of the example method, as shown in Figure 9. As optimizedExample3 should be called exactly as example, its initial stack frame is the same.

caller	flag
ip	flag
method	flag
receiver	abstract receiver (this)
arg 1	abstract argument (bool)
arg 2	abstract argument (array)

Figure 9 – Base stack frame to call example

Symbolic execution of an optimized method. Each node of the optimized method is executed according to a rule shown on Figure 10. Operations on the stack frame, such as pushing or popping value on stack or loading or storing into temporary variables work similarly to the regular interpreter, manipulating abstract values on an abstract stack instead of concrete values on a concrete stack. Two kinds of node need to be handled specifically.

On the one hand, control flow operations such as jumps and returns need to be specifically executed so that the symbolic execution executes all the different branches once, and loop bodies a fixed number of times. We added a field in the abstract stack frame to handle metadata about which branches has already been taken or not. We do not go further into detail about that as designing a good symbolic execution model is orthogonal to the goal of the paper.

On the other hand, the most complex node to execute is virtual method call. For our analysis, there is no need to symbolically execute methods non inlined in the optimized method because there are out of the optimization scope. Each method call is symbolically executed by adding an abstract node on stack which corresponds to the result of the method call with a given abstract receiver and abstract arguments. Therefore, the symbolic execution for the optimized method always uses a single stack frame in addition to the flagged one.

As the symbolic execution does not execute non inlined virtual calls, there are no issues with system calls and non inlined primitive operations on abstract values. For example, a primitive operation can answer the current date and time. Running this primitive several times will answer different results as time is progressing. The symbolic execution does not execute such calls.

The execution stops when the symbolic execution encounters an on-stack-replacement point. The current abstract stack used by the symbolic execution, corresponding to the run-time stack state for this on-stack-replacement point, can be used for analysis as explained in the next subsections. When the analysis is finished, the symbolic execution of the optimized method is restarted until it reaches the next on-stack-replacement point to do the next analysis or until it reaches the end of the method.

Example. With the example pair of methods example and optimizedExample3, the symbolic execution on the optimized method optimizedExample3 stops and provides an abstract stack for further analysis at the fifth on-stack-replacement points available in optimizedExample3, the three guards, the back jump of the loop and the method call println. For this example we consider the on-stack-replacement point on the guard that guarantees that the first value of the array has the class Printable. The symbolic execution provides the abstract stack shown on Figure 11.

Instruction	Symbolic execution of **non** optimized method	Symbolic execution of optimized method
Virtual call	If the call site was inlined, execute the method inlined, else push abstract method result on stack	Push abstract method result on stack
Primitive operation	Push abstract primitive result on stack	
Conditional jump	Take one branch, then when reaching return the other branch will be executed	
Unconditional jump	Follow it if forward jump, else resume the execution to the last untaken branch	
Load temporary	Push corresponding value got from the abstract frame on the abstract stack	
Load receiver	Push receiver for the abstract stack	
Load constant	Push constant on stack	
Load argument	Push corresponding value got from the abstract frame on stack	
Store into temporary	Store the value on top of stack into the stack field for the temporary	
Pop	Pop the abstract stack	
Return	Resume the execution to the last untaken jump or return the method value. If several returns are present, return multiple values.	

Figure 10 – Symbolic execution rules

Interrupt point guard(array*[i]*, Printable) - Abstract stacks		
Optimized stack		
1	caller	flag
	ip	flag
	method	flag
	receiver	abstract receiver
	arg 1	abstract arg (bool)
	arg 2	abstract arg (array)
2	caller	frame 1
	ip	0x001a56b8
	method	optimizedExample3
	temp 1	0
	arg 1	abstract result: (abstract arg (array) *[0]*)

Figure 11 – Abstract stack for the optimized method

4.3 Deoptimizing the optimized abstract stack

The JIT compiler is able to deoptimize the runtime stack at on-stack-replacement points. Therefore, it can deoptimize the abstract stack provided, using the mechanism provided for the concrete stack. This process provides a deoptimized stack. Each field of the deoptimized stack should be, in case of concrete values, equal to the matching field in the regular stack. The deoptimized stack is therefore ready to be compared with the regular stack as shown on Figure 12.

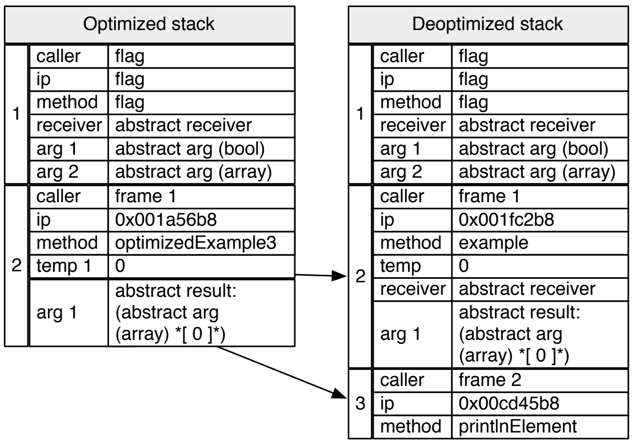


Figure 12 – Abstract stack deoptimization

4.4 Generating abstract stacks for the non optimized method

The abstract stack for the non optimized method is generated using symbolic execution in a similar way to the abstract stack to the optimized method, as shown on Figure 10. Two aspects are different: the symbolic execution needs to stop on the same on-stack-replacement point than the one where the other symbolic execution stopped and virtual calls needs to be executed if they were inlined in the optimized method.

Reaching the same on-stack-replacement point. The non optimized method may have more on-stack-replacement points than its optimized version, for example, if a guard was removed because it was implied by previous guards in the method. We need to guarantee the symbolic execution stops at the exact same on-stack-replacement point that the other symbolic execution. This is possible because the abstract stack should be in the same state than the deoptimized stack. One needs to check that the bottom frame has the same method and the same instruction pointer than the deoptimized stack, and that the number of frames in both stack are the same to avoid problems in case of recursive methods.

In addition, one needs to take specific care of loops. It is possible to detect the starting point of a loop, but to merge the abstract values, we need the abstract values already computed coming from the back edge already computed. The problem is that some abstract calls can iteratively grow the size of the expression tree represented by an abstract value after each iteration of the loop. Our approach was to iterate twice only over the loop, the first time to determine the abstract value that needs to be merged at the beginning of the loop, the second time to try dynamic deoptimization with the abstract values using the back jump merge point.

When the symbolic execution on the non optimized method reaches the same on-stack-replacement point than the other symbolic execution, the execution is stopped and the symbolic execution provides a second abstract stack. The static approach can then analyze the regular and optimized abstract stacks, as explained in Section 4.5.

Executing virtual calls. To execute the non optimized method, the static analyzer extracts from the optimized method the list of method calls that have been inlined and what methods have been inlined. We added this option to our JIT compiler by storing information in a collection each time the JIT compiler inlines a method call. The symbolic execution on the unoptimized method executes method calls only if the method call site is in the list of inlined method calls given by the optimizing compiler for the optimized method. In this case, it creates a new abstract stack frame with the method inlined (it cannot look up the method to run from the abstract receiver and argument, so it needs the method inlined from the optimizing compiler). If the method call is not in the list, the symbolic execution does not execute the method call and pushes instead the abstract result for the message send selector and its specified abstract receiver and arguments, as the symbolic execution executing the optimized method.

4.5 Comparing the abstract stacks

Due to the abstract nature of each value on the abstract stacks, our approach cannot just check for the equality of each corresponding values. A simple equality check is impossible because of:

- Control flow optimizations
- Primitive specialization

Comparing conditional assignment. Firstly, due to dead branch elimination, the abstract values of the deoptimized stack are included, and not equal, to their matching value in the unoptimized stack because a conditional assignment might have become unconditional when a branch was removed.

For example, in the method and its optimized version shown in Figure 13, the return value of the method computed by the symbolic execution is either 1 or 0 in the unoptimized method, but is 1 in the optimized method due to dead branch elimination. If this method is inlined in another method and its result is used in an on-stack-replacement point stack comparison, the approach validates the comparison because 1 is included in 0 or 1.

```
public deadBranchElim(boolean bool) {
    int result;
    if (bool) { result = 1; }
    else { result = 0; };
    return result; }

public deadBranchElimOptimized(boolean bool) {
    guard(bool, true);
    return 1; }
```

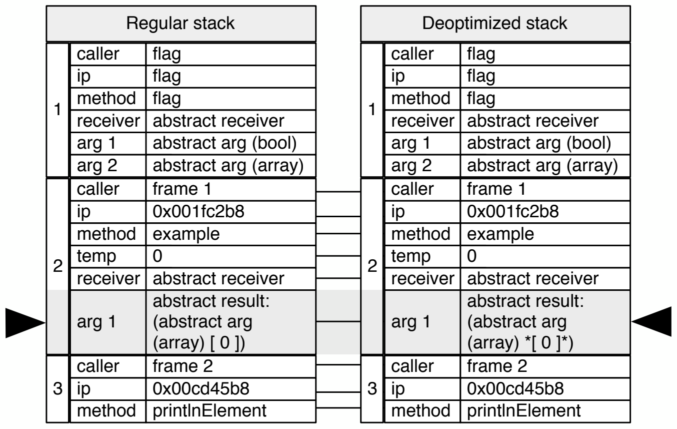
Figure 13 – Dead branch elimination

Comparing primitive operations. Secondly, an important part of compiler optimizations consists of optimizing primitive operations. A common optimization on primitive operations specializes array access to remove bounds checks [BGS00]. In this example, as well as in other optimizations on primitive operations such as number arithmetic specialization, the JIT compiler specializes the primitive operations, *i.e.,* another version of the primitive implementing a subset of the operation is generated instead of the primitive itself.

While comparing the two abstract stacks, these specializations may be problematic. On the one hand, one has the abstract result of a generic primitive operation and on the other hand one has the abstract result of the specialized primitive operation.

Optimized primitive operation inclusion. This issue is solved by defining for each generic primitive operation the possible specialized versions. For example, the generic operation to access the fields of an array can be specialized in three operations, each of them expecting an in-bounds integer arguments, but expecting a different kind of array (array of raw bytes, array of raw words, array of pointers). While comparing the stack, if two results of abstract messages are not equal, the approach checks if the operation on the abstract stack for the optimized method is a specialized primitive, and if this is the case, it checks if the corresponding abstract message on the abstract stack for the regular method is the generic primitive operation for the specialized primitive.

Example. In the example, we compare the deoptimized and the regular abstract stack, as shown on Figure 14. To compare the stack, it compares the values hold by each stack field to its corresponding value in the other stack. When comparing the last field of the second stack frame, it detects that the tree of abstract values are different, as shown in Figure 15. On the one side, the operation performed is the generic array access, whereas on the other side a specialized operation, array access without bounds check, is performed. However, since one operation is included in the other, the comparison succeeds.



Regular stack		
1	caller	flag
	ip	flag
	method	flag
	receiver	abstract receiver
	arg 1	abstract arg (bool)
	arg 2	abstract arg (array)
2	caller	frame 1
	ip	0x001fc2b8
	method	example
	temp	0
	receiver	abstract receiver
	arg 1	abstract result: (abstract arg (array) [0])
3	caller	frame 2
	ip	0x00cd45b8
	method	printlnElement

Deoptimized stack		
1	caller	flag
	ip	flag
	method	flag
	receiver	abstract receiver
	arg 1	abstract arg (bool)
	arg 2	abstract arg (array)
2	caller	frame 1
	ip	0x001fc2b8
	method	example
	temp	0
	receiver	abstract receiver
	arg 1	abstract result: (abstract arg (array) *[0]*)
3	caller	frame 2
	ip	0x00cd45b8
	method	printlnElement

Figure 14 – Abstract stack comparison

Inclusion rules. Due to the abstract nature of the nodes, some nodes are included in the corresponding node on the other stack instead of just being equal.

We consider the comparison between a node N from the optimized method's abstract stack and its corresponding node N' on the non optimized method's stack. Three cases are possibles:

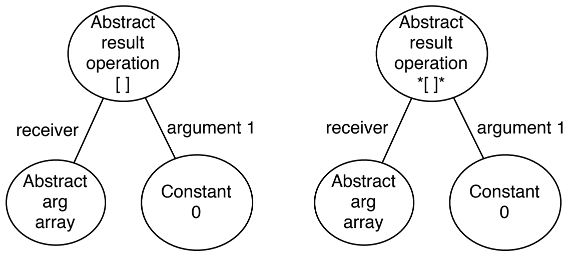


Figure 15 – Comparison of the stack field

- N' is a primitive operation, in which case N needs to be equal to N' or being one of the allowed corresponding optimized primitives operations.
- N' is a multiple values node (result of conditional assigment), in which case N needs to be either a multiple values node with its nodes equals to a subset of N' nodes or be directly equal to one of the nodes of N'
- N is any other node, N needs to be equal to N'

If the inclusion fails for a node, then the validation fails for the given method. If the inclusion succeeds for a given node, it is applied on the children of N and the corresponding children in N' until the whole tree is walked.

Complexity of the inclusion rule. The rule used looks quite simple. In fact, most of our deoptimization bugs are related to the optimization of closures, and a simple approach was able to catch enough bugs so that our virtual machine with the runtime optimizer could run macro benchmarks and small size applications such as a web server. The rule can be improved in two ways, firstly by being agnostic about specific optimizations such as strength reduction to support better such optimization and secondly by being more precise in the comparison of abstract stacks.

In our case, it seems that most of our bugs were related to closure inlining as well as memory representation changes related to closure remote temporary variables (once the closure is inlined, they become normal temporary variables). Other optimizations such as bounds check elimination were not as tricky for the deoptimizer. We didn't invest time there because it felt it would not help catching the problematic bugs. In the future, we will add for sure an optimization that removes object allocation, moving the object's field to temporaries if the object does not escape the optimized method, and we believe that the validation technique will need to be extended to validate the reconstruction of objects.

5 Validation

To validate our approach, we used a Smalltalk environment where we could easily build such an approach to test speculative inlining and its dynamic deoptimization. We run our tests on over 132000 methods to validate our approach.

5.1 Industrial context

We validate our idea with a runtime bytecode to bytecode compiler written in Smalltalk for a high performance Smalltalk virtual machine. The runtime optimizer performs speculative inlining on method and closure activations. In addition, it specializes Array access primitives to remove bounds checks in a similar way to the algorithm described in [BGS00] thanks to specialized bytecode operations.

Why a bytecode to bytecode optimizer. In 2012, the production virtual machine had only an interpreter and a baseline Just-in-Time compiler, which translates bytecode to assembly code. Recently, a runtime optimizer was added to improve the virtual machine performance. It was designed as a bytecode to bytecode compiler for two reasons independent from the work presented in this paper. Firstly, the development team wanted to reduce the maintenance cost of a high performance virtual machine. Java's hotspot virtual machine was gifted with billions of dollars of investment, whereas the virtual machine we used has a limited number of engineers. This design reduced the maintenance cost because:

- The optimized methods generated can reuse the baseline JIT compiler as a backend to generate machine code with limited changes: it was only needed to add support for hot spot detection and for a few extra bytecodes used only in optimized methods.
- The bytecode to bytecode optimizer and deoptimizer can be implemented in the high-level language, running in the same runtime than the programmer's application, in a similar way to the Graal JIT compiler available for Java's hotspot. Implementing these complex artifacts in a higher level language made them simpler to maintain. In this context, if the optimizer generates machine code optimized methods, the virtual machine has to be extended to properly install and use these methods. In this design, as optimized methods are bytecoded methods, the virtual machine already knows how to execute them: it was only needed to add support for the few extra bytecodes used only in optimized methods and an extra primitive operation which answers the runtime information for a given method.

Secondly, as the normal Smalltalk workflow for the developer to save code is to take a snapshot of all the objects alive at a given time (including compiled methods) and to restart the runtime from this snapshot to resume his development from the last code save, saving optimized methods as regular bytecoded methods allows one to save the optimized state of the runtime by default across start-ups.

Why we use this VM. We use this Smalltalk virtual machine because our team is currently working with Cadence Design Systems on improving this virtual machine performance. We wanted to support the development with important test suites to have the fewest bugs possible when we release the runtime optimizer.

Smalltalk and its runtime bytecode to bytecode optimizer fits to our needs because:

- An infrastructure is already there to easily parse bytecode. Symbolic execution is easier to write over the bytecode than over assembly code.
- Smalltalk, as many high-level languages, provides a unit testing framework that we can reuse to write our tests.
- A bytecode to bytecode optimizer can be called from the language as any Smalltalk library.

- Smalltalk provides an efficient reified stack language side that can be introspected and edited as defined in [DS84], we are able to reuse this infrastructure for our abstract stack.

System characterization. This section aims to characterize the system we used for our validation.

Smalltalk is an object-oriented, dynamically-typed, reflective programming language. Smalltalk methods are usually short (6.9 lines on average including the method name and comments, but excluding empty lines) due to coding conventions. Every method call is a virtual call, which means a look up is theoretically performed for every call (even if the virtual machine optimizes most of these look-ups for obvious performance reasons). In a typical Smalltalk application, 90% of the virtual calls are always performed on a receiver with the same type. 9% of the virtual calls are performed on a receiver that can have one type among a limited set of 6 different types. 1% of the virtual calls are performed on a receiver that can have many different types [HCU91].

Smalltalk features closures which encapsulate their enclosing environment. These closures are optimized by the runtime optimizer. A specific aspect of Smalltalk closures is that a closure can return either to its outer frame or the outer frame of its enclosing environment. The latter is called a *non local return*. In case of a closure with control flow operations, the closure can potentially return to two different points (its outer frame or its enclosing environment outer frame) depending on the branch taken. Since closures can be passed as argument, as any other value, such enclosing outer frame may not be on the stack anymore when a non local return is performed, raising an exception. To perform non local returns, when a closure is created, the virtual machine keeps a pointer to its enclosing environment stack frame. For our optimizer, it means that this pointer has to be correctly restored when the stack is deoptimized. Closures are widely used in Smalltalk: in the Pharo Smalltalk Kernel, 19.3% of compiled methods holds the bytecode to create a closure.

Due to its reflective nature, the programmer can access stack frames at runtime as if they were objects [DS84] and edit the temporary variable values or exchange two objects identity by asking the virtual machine to switch all the pointers to one object to target other objects. The stack frame reification is the foundation for the debugger, the exception and the continuation implementations. Exchanging pointer identity is used to migrate instances when one adds at runtime an instance variable to a class and for database proxies. Although these two operations are important, they are uncommon. Therefore, their implementation can be slowed down a little if in exchange the common cases run faster. The current implementation requires the virtual machine to dynamically deoptimize some stack frames if one of these two operations is performed, allowing the optimizer to optimize methods ignoring these two cases.

5.2 Results: large data to validate

To validate our approach, we need a large set of methods to optimize. We used the kernel code (around 74000 methods) as well as the most common frameworks and libraries of the language. We run our test framework on all the method of the Pharo Smalltalk Kernel and base libraries provided by the environment. In addition, we loaded the code of the three main frameworks:

- *Seaside*: a framework for dynamic web application ($\sim$ 11000 methods) [DLR07].
- *Moose*: a framework for source code analysis ($\sim$ 32000 methods) [NDG05].

- *VMMaker*: a framework to build virtual machines ($\sim$ 15000 methods) [IKM+97].

All these 132000 methods were deoptimized at each on-stack-replacement point, for an average of 6.3 on-stack-replacement point per optimized method.

Validating this approach is difficult as it was built to support ongoing agile development and not to find bugs in a well-known and used compilers. In the next two paragraphs, we discussed how the approach supported our agile development process. If the reader is used to agile development, he may find the statements obvious but this is what we lived.

Supporting agile development. When we first set up our approach, in the agile philosophy of writing tests before the actual code, the runtime bytecode to bytecode optimizer was in early stage of development, only optimizing and deoptimizing successfully simple cases. Obviously, at that stage, most methods tested with our approach were failing. During the development process, 97% of our commits improved the results of the approach, increasing the number of methods that were successfully deoptimized at each interrupt point. On the other hand, 3% of commits reduced the number of methods that could be successfully tested, notifying the development team about the methods that failed due to the latest commit. Part of these commits consisted in important refactorings with results expected by the development team. A subset of this 3%, however, were real bugs that were introduced by the code committed. The approach identified such bugs to the development team allowing them to detect and fix the bugs early.

Most of the reappearing bugs were related to the deoptimization of closures. It is normal since closures with their non-local semantics are the most complex part of Smalltalk execution model. Typically, the closure encapsulating environment was not correctly set.

Raising the confidence level in the product. After 18 months of development, all the tested methods passed this validation. The approach led to an important confidence level in the optimizer, allowing to integrate the optimizer in the production VM with lower risks. This result is not specific to our approach but a general behavior of using a large test basis.

Now that the bytecode to bytecode optimizer is production-ready, it will be put into the production VM in summer 2016. End-users might find out bugs in the new optimizer based on their use cases. We will use our approach to guarantee that we do not break dynamic deoptimization while fixing end-users bugs.

6 Discussion and future work

6.1 Supporting other JIT compiler optimizations

Right now, our implementation can only compare the abstract stacks if the optimized methods have been generated using the following optimizations: speculative inlining, dead branch elimination and primitive specialization. We did not investigate in other optimizations as our optimizer does not perform them. In addition, we believe dynamic deoptimization is difficult to stabilize mostly because of speculative inlining. Mainstream JIT compilers can perform other optimizations such as the ones described in the next paragraphs, which may need to be disabled to use our implementation because the inclusion rule or the symbolic execution we implemented may not support it. It would be interesting to develop new techniques to validate dynamic deoptimization with other JIT compiler optimizations enabled. We discuss in this subsection three different optimizations that would be interesting to support.

Converting heap allocations to stack allocations. Thanks to escape analysis, a JIT compiler can allocate an object on the stack instead of on the heap if the object is alive only between two non-inlined method calls. This allows the state of the object to be optimized with data-flow optimizations and avoids the need of managing garbage collection of the object. This optimization complicates dynamic deoptimization as such objects need to be recreated during the deoptimization. It would be interesting to validate dynamic deoptimization with such optimizations.

We could not do it because the runtime compiler we used was not performing such optimizations, but it should work without additional work on our approach.

Exception optimizations. Another interesting optimization added in the past decade in the Java HotSpot virtual machine is the optimization of exceptions [OKN01]. This optimization allows some exceptions to be inlined to jumps, making them faster. This optimization is interesting because it complicates dynamic deoptimization. The stack to be recreated is more complex due to the exception signal and handling. The symbolic execution used in our approach mirrors the behavior of an interpreter, so it is possible to improve to support exceptions. However, it would complicate our approach as implementing the support for exceptions in the symbolic execution model is not an easy task.

Arithmetic manipulation. Some arithmetic optimizations such as strength reduction can change operations: strength reduction can change an integer multiplication to an integer addition if the correct conditions are met. With this kind of optimizations, it is difficult to compare the values on the abstract stacks, as the regular stack may have an abstract value corresponding to the result of different arithmetic operations than the optimized stack.

We did not investigate in this direction because the runtime compiler we used did not implement such optimizations. We believe our approach could still work using a mathematical model to compute equivalence between arithmetic expressions, but clearly further work in that direction would be needed.

6.2 Symbolic execution of a lower code representation

Symbolic execution of machine code. Our validation relies on a bytecode symbolic execution, and works fine to validate a runtime optimizing bytecode to bytecode compiler. Other optimizing JIT compilers such as Java's HotSpot, however, optimize the methods from bytecode to machine code and not to bytecode. For our approach, we need that the original and optimized version of the method share the same intermediate representation. In the case of Java's HotSpot, one could compare the machine code generated by the baseline JIT compiler and the optimizing JIT compiler. To validate dynamic deoptimization with these machine code versions of the method and our approach, one would need to build an abstract processor for one of the main machine code used, such as x86_64. This abstract processor would be able to symbolically execute machine code routine with abstract values, to interrupt its execution and to let the user access the abstract stack for stack deoptimization and stack comparison.

Such an abstract processor could be restricted by several constraints. Firstly, it needs only to be able to symbolically execute only the instructions generated by the JIT compiler, which are a subset of the available machine instructions. Secondly, as in our approach, it could ignore calls escaping the scope of the optimizations, using an abstract value instead. These constraints may reduce the complexity of an abstract processor, however, such an abstract processor is extremely difficult to implement. Current state-of-the-art machine code symbolic executors usually symbolically execute only part of the program, while the rest of the execution is done with concrete values. For example, Cadar *et al.,* [CDE08] concretize floating-point data to avoid their symbolic execution, others [GKS05, MMP$^+$12] concretize pointers to avoid symbolic execution of memory access, and lastly the S2E platform uses selective symbolic execution to automatically minimize the amount of code that has to be executed symbolically given a target analysis [CKC11]. Hence, the symbolic executor required for our validation technique seems to require several improvements over the state-of-the-art executors, implying that the application of our approach on machine code would require a massive amount of work to be production-ready.

Symbolic execution of a low level intermediate representation. As executing symbolically machine code is a tedious task, one may prefer to execute symbolically a low-level intermediate representation used by the JIT compiler. This symbolic execution may be simpler to implement and to maintain. We did not investigate in this direction as we wanted our approach to be completely independent from the optimizer and its intermediate representation, so we could run the test suites with another optimizer instead of the one we test. Such an approach has other flaws: the different optimizing paths of the JIT compiler have to share the same low-level intermediate representation to work, which is not necessarily the case, and dynamic deoptimization have to be performed on abstract stacks using abstract nodes for the low-level intermediate representation, which most probably will not work out of the box.

6.3 Constraint solver

As described in subsection 4.5, to compare the two abstract stacks we only apply an inclusion rule. We considered implementing a more complex rule, based on the different constraints met during symbolic execution such as branch's conditions and guards, and add a constraint solver to find out more precisely if the value in the optimized method's abstract stack is correct. We didn't implement it because we wanted the validator to remain simple to maintain.

Complexity and maintenance. Our solution already relies on a symbolic executor, which already needs to be maintained. To implement it, we reused a common code base from the production interpreter. Most of its maintenance work will be done while maintaining the interpreter. Although most behavior is inherited from the interpreter, which is stable, when we ran the first validation we found a few bugs due to the symbolic executor and not to the validated methods. These bugs were quickly fixed because we tried to keep the symbolic executor simple, and we had very quickly only bugs coming from the optimizer. If we wanted to implement a constraint solver, we would need to spend time to implement and maintain a complex artifact, and we would need for each bug detected, to ensure that the bug truly come from the validated method and not from the constraint solver. We didn't want to invest time in that direction, even though it may be interesting research-wise.

Efficiency of a more complex rule. Most bugs we had came from the dynamic deoptimization of closures. Once the validator successfully passed, the runtime was usable with the runtime optimizer: users didn't report additional crashes. It has yet to be proven that, in practice, given the large data we used as input for the validator, a more complex rule to compare the abstract stack would find additional bugs. This is future work.

7 Related Work

Some related works about static analysis of compilers have been discussed in the Section 3. In this section we discuss other related work.

C compiler validation using symbolic execution. In our approach, we needed static information about the runtime stack. We got this information from the symbolic execution. C compiler validation may also benefit from information about the runtime state of the program. To extract runtime information from the program, one solution is to run the optimized code with controlled test inputs, extract information from the runtime and use it in static analysis. This approach let a research team find 147 confirmed, unique bug reports for GCC and LLVM alone by profiling program test executions and pruning its unexecuted code [LAS14].

Bounded model checking. Some validation tools, such as LLBMC [MFS12], use symbolic execution to discover information about the runtime stack and use it to prove the program cannot reach a runtime state leading to a bug, such as an out of bounds access array access.

JIT compiler optimizations based on symbolic execution. Symbolic execution has been used by compiler teams for other purpose than validating or verifying the compiler. Their goal was to implement compiler optimization using symbolic execution. A team has used symbolic execution for efficient type inference and applied it to JavaScript optimization [LV10]. Another team used it in the context of their tracing JIT compiler [DLR14].

Other teams have also used symbolic execution to improve their JIT compilers. However, none of this work is related to JIT compiler validation.

JIT compiler validation. Most of the research work related to compiler validation is done on C compilers. However, a few papers exist on JIT compiler validation.

For example, Chris Hawblitzel *et al.,* worked on the reduction of false alarms in their JIT compiler validator [HLP+13], an alarm being triggered by the compiler validator when it finds a bug. They reduced the number of false alarms to 2.2% of the alarms, whereas other works report a false alarm rate of 10 to 40% [STL11, TGM11] or are specialized on specific optimizations to decrease their alarm rate to only 3% [Nec00].

Papers described in this section are about JIT compiler validation, but they do not discuss about the validation of dynamic deoptimization.

Validation of dynamic deoptimization in industrial virtual machines. Some industrial virtual machine, such as Google V8's Javascript engine [Goo], have validation techniques for dynamic deoptimization. In the example of V8, an option *–deopt-every-n-times* is available and forces the virtual machine to deoptimize the code every *n* times it reaches a deoptimization point. This approach stresses the dynamic deoptimizer and can help discover many bugs.

This approach often leads to virtual machine crashes when a bug is detected, which is hard to debug, whereas with symbolic execution we have a runtime error that we can explore.

8 Conclusion

In this paper, we looked for a solution to validate the correctness of dynamic deoptimization in JIT compilers. This problem applies in any programming language featuring dynamic deoptimization in their execution environment, which is common for language with late binding and running on top of a virtual machine. We proposed a static approach to guarantee that the virtual machine can correctly deoptimize dynamically the runtime stack when needed. This approach relies on symbolic execution to access statically the runtime stack of methods. The solution was validated by an implementation which is now used to test our production code base.

Acknowledgements

We thank Stefan Marr and Xavier Leroy for their reviews of an earlier versions of this article.

References

[AWR05] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving Virtual Machine Performance Using a Cross-run Profile Repository. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, 2005. doi:10.1145/1094811.1094835.

[BGS00] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Programming Language Design and Implementation*, PLDI '00, 2000. doi:10.1145/349299.349342.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Operating Systems Design and Implementation*, OSDI'08, 2008.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011. doi:10.1145/1950365.1950396.

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A Flexible Environment for Building Dynamic Web Applications. *IEEE Softw.*, 24(5), September 2007.

[DLR14] Stefano Dissegna, Francesco Logozzo, and Francesco Ranzato. Tracing Compilation by Abstract Interpretation. In *Principles of Programming Languages*, POPL '14, 2014. doi:10.1145/2535838.2535866.

[DOD12] Vinicius H. S. Durelli, Jeff Offutt, and Marcio E. Delamaro. Toward Harnessing High-level Language Virtual Machines for Further Speeding Up Weak Mutation Testings. In *International Conference on Software Testing, Verification and Validation*, ICST '12, 2012. doi:10.1109/ICST.2012.158.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 system. In *Principles of Programming Languages*, POPL '84, 1984. URL: http://webpages.charter.net/allanms/popl84.pdf, doi:10.1145/800017.800542.

[ECM01] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, dec 2001. URL: http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf;http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[ER08] Eric Eide and John Regehr. Volatiles Are Miscompiled, and What to Do About It. In *Embedded Software*, EMSOFT '08, 2008. doi:10.1145/1450058.1450093.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Programming Language Design and Implementation*, PLDI '05, 2005. doi:10.1145/1065010.1065036.

[Goo] Google. V8 source code repository: https://github.com/v8/v8.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *European Conference on Object-Oriented Programming*, ECOOP '91, London, UK, UK, 1991.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Programming Language Design and Implementation*, PLDI '92, 1992. doi:10.1145/143095.143114.

[HLP$^+$13] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will You Still Compile Me Tomorrow? Static Cross-version Compiler Validation. In *Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013. doi:10.1145/2491411.2491442.

[IKM$^+$97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, 1997. doi:10.1145/263698.263754.

[LAS14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. In *Programming Language Design and Implementation*, PLDI '14, 2014. doi:10.1145/2594291.2594334.

[Ler06] Xavier Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Principles of Programming Languages*, POPL '06, 2006. doi:10.1145/1111037.1111042.

[LV10] Francesco Logozzo and Herman Venter. RATA: Rapid Atomic Type Analysis by Abstract Interpretation. Application to Javascript Optimization. In *European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, 2010. doi:10.1007/978-3-642-11970-5_5.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded Model Checking of C and C++; Programs Using a Compiler IR. In *Verified Software: Theories, Tools, Experiments*, VSTTE'12, 2012. doi:10.1007/978-3-642-27705-4_12.

[MMP$^+$12] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012. doi:10.1145/2150976.2151012.

[Myr10] Magnus O. Myreen. Verified Just-in-time Compiler on x86. In *Principles of Programming Languages*, POPL '10, 2010. doi:10.1145/1706299.1706313.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The Story of Moose: An Agile Reengineering Environment. In *European Software Engineering Conference Held Jointly with Foundations of Software Engineering*, ESEC/FSE-13, 2005. doi:10.1145/1095430.1081707.

[Nec00] George C. Necula. Translation Validation for an Optimizing Compiler. In *Programming Language Design and Implementation*, PLDI '00, 2000. doi:10.1145/349299.349314.

[OKN01] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. A Study of Exception Handling and Its Dynamic Optimization in Java. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01. ACM, 2001. doi:10.1145/504282.504289.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98. Springer-Verlag, 1998.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java hotspotTM Server Compiler. In *Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01. USENIX Association, 2001.

[RSB$^+$14] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. Surgical Precision JIT Compilers. In *Programming Language Design and Implementation*, PLDI '14, 2014. doi:10.1145/2594291.2594316.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based Translation Validator for LLVM. In *Computer Aided Verification*, CAV'11, Berlin, Heidelberg, 2011.

[TGM11] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating Value-graph Translation Validation for LLVM. In *Programming Language Design and Implementation*, PLDI '11, New York, NY, USA, 2011. doi:10.1145/1993498.1993533.

[TWSC10] Ben L. Titzer, Thomas Würthinger, Doug Simon, and Marcelo Cintra. Improving Compiler-runtime Separation with XIR. In *Virtual Execution Environments*, VEE '10, 2010. doi:10.1145/1735997.1736005.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Programming Language Design and Implementation*, PLDI '11, 2011. doi:10.1145/1993498.1993532.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Principles of Programming Languages*, POPL '12, 2012. doi:10.1145/2103656.2103709.

About the authors

Clément Béra is a phd student interested in virtual machines for object-oriented languages. Contact him at clement.bera@inria.fr, or visit http://clementbera.wordpress.com/.

Eliot Miranda is a long-time Smalltalk systems programmer and virtual machine implementor. Contact him at eliot.miranda@gmail.com, or visit http://www.mirandabanda.org/.

Marcus Denker is a researcher at Inria Lille. His research interests include reflective systems and implementation aspects of reflective programming languages. Contact him at marcus.denker@inria.fr, or visit http://marcusdenker.de.

Stéphane Ducasse is a research director at Inria and manages the RMOD team. His fields of expertise are language design, reflective programming, software maintenance, metamodelling, program analysis and visualisation. He is a contributor to Pharo http://www.pharo.org and the Moose analysis platform http://www.moosetechnology.org. He wrote several books on web development, maintenance and Pharo. Contact him at stephane.ducasse@inria.fr, or visit http://stephane.ducasse.free.fr/.