One Compiler: Deoptimization to Optimized Code

Christian Wimmer Vojin Jovanovic Erik Eckstein * Thomas Würthinger
Oracle Labs
{christian.wimmer, vojin.jovanovic, thomas.wuerthinger}@oracle.com

Abstract

A multi-tier virtual machine (VM) deoptimizes and transfers last-tier execution to the first-tier execution when a speculative optimization is invalidated. The first-tier target of deoptimization is either an interpreter or code compiled by a baseline compiler. Because such a first-tier execution uses a fixed stack frame layout, this complicates all VM components that need to walk the stack. We propose to use the optimizing compiler also to compile deoptimization target code, i.e., the non-speculative first-tier code where execution continues after a deoptimization. Deoptimization entry points are described with the same scope descriptors used to describe the origin of the deoptimization, i.e., deoptimization is a two-way matching of two scope descriptors describing the same abstract frame at the same virtual program counter. We evaluate this deoptimization approach in a high-performance JavaScript VM. It strictly uses a one-compiler approach, i.e., all frames on the stack originate from the same compiler.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords Java; JavaScript; deoptimization; optimization; virtual machine; language implementation

1. Introduction

Speculative optimizations are a key ingredient to achieve high performance when executing managed languages such as Java or JavaScript. The optimizing compiler speculates based on profiling feedback collected by a lower execution tier, e.g., incorporates type profiles or assumes that certain hard-to-optimize conditions do not occur. If the speculation was too aggressive, *deoptimization* is used to revert back to the first execution tier. Such adaptive optimization and deoptimization was first implemented for the SELF VM [18], and is now used by most high-performance VMs such as the Java HotSpot VM or the V8 JavaScript VM.

Such a multi-tier optimization system increases the implementation and maintenance costs for a VM: In addition to the optimizing compiler, a separate first-tier execution system must be implemented [1, 3, 16]. The first tier is usually either an interpreter or a baseline compiler, i.e., a compiler that only assembles prepared patterns of machine code. Both interpreters and baseline compilers use a fixed layout for stack frames, i.e., each local variable of a source method has a known position in the stack frame. The deoptimization handler writes to these known locations without the need for method-specific metadata generated by the interpreter or baseline compiler. In contrast, the stack frame layout generated by optimizing compilers uses spill slots and registers allocated by the register allocator. The compiler generates metadata for every possible deoptimization origin point, which is used by the deoptimization handler to read values.

Implementing and optimizing an interpreter or baseline compiler is far from trivial [23], even though their complexity is lower than the complexity of an optimizing compiler. They need to be maintained and ported to new architectures. Deoptimization entry points must be placed manually by the implementer, which is tedious and error-prone. For example, the V8 baseline compiler[1] has more than a hundred manually placed deoptimization entry points, with more than half of them in architecture-specific source code, i.e., replicated in all nine architecture-specific baseline compiler implementations. Bugs that lead to semantic differences between baseline and optimized code are hard to identify and reduce the reliability of the whole VM. In addition, multiple execution tiers complicate many other parts of the VM because they have to deal with multiple layouts of stack frames: stack walking (e.g., computing the stack frame size), collection of root pointers for garbage collection (e.g., using an abstract interpretation of bytecodes), exception handling (e.g., interpreting the exception handler metadata of the bytecodes), and debugging tools.

We propose a VM architecture that uses only one compiler for all execution tiers. With only a single compiler, deoptimization targets are also optimized stack frames. Instead of relying on a fixed layout of the frame, the compiler needs to generate metadata for the deoptimization target code. We propose to use the same metadata format also used for deoptimization origins. The deoptimization handler reads and writes stack frames whose layout is described by the same compiler in the same metadata format. Deoptimization entry points are automatically placed by the compiler. This aligns with our goal of reducing the complexity of the VM and avoiding duplication of functionality.

This paper outlines the general approach of deoptimization to optimized code, and discusses which compiler optimizations are suitable for deoptimization target code. To the best of our knowledge, this is the first system that allows deoptimization to optimized code, with the notable exception of the Jikes RVM [12]. The deoptimization approach of the Jikes RVM, based on a generalization of on-stack-replacement, allows deoptimization to code generated by the optimizing compiler of Jikes. But because the approach requires the compilation to be performed at the time of deoptimization, we argue that it is infeasible to use anything other than baseline compiled code as the deoptimization target in the Jikes RVM.

* Work performed while being a member of Oracle Labs.

CC'17, February 5–6, 2017, Austin, TX, USA
ACM. 978-1-4503-5233-8/17/02...$15.00
http://dx.doi.org/10.1145/3033019.3033025

[1] https://github.com/v8/v8/tree/master/src/full-codegen

In summary, this paper contributes the following:

- We present a novel way to implement deoptimization, where the deoptimization target code is generated by the same optimizing compiler that also generates the deoptimization origin code. The approach is independent of the programming language and independent of the optimizing compiler.
- We show which compiler optimizations can still be performed when compiling the deoptimization target code.
- We use and evaluate the new deoptimization approach in a high-performance JavaScript VM that strictly adheres to the one-compiler design paradigm.

2. Deoptimization to Optimized Code

In the seminal first paper about deoptimization [18], deoptimization replaces stack frames of optimized code with frames of unoptimized code. Because of method inlining, one optimized frame can be replaced with many unoptimized frames. Replacing one frame with many frames usually increases the size needed for the frames on the stack. Therefore, *lazy deoptimization* is necessary: When a method is marked as deoptimized, the stacks of all threads are scanned for affected optimized frames. The return address that would return to the optimized code is patched to instead return to the *deoptimization handler*. At the time the deoptimization handler runs, the frame to be deoptimized is at the top of the stack and can be replaced with larger unoptimized frames.

The optimizing compiler creates metadata, called *scope descriptors*, for all possible deoptimization origin points. A scope descriptor specifies 1) the method, 2) the *virtual program counter*, i.e., the execution point in the method, 3) all values that are live at that point in the method (typically local variables and expression stack entries), and 4) a reference to the scope descriptor of the caller method, which forms a linked list of scope descriptors when the optimizing compiler performs method inlining. A value can either be a constant, a stack slot, or a register. For each scope descriptor of the inlining chain, the deoptimization handler creates a target stack frame and fills it with the values described in the scope descriptor and puts it on the stack. Unoptimized frames have a fixed layout, i.e., values are consecutive on the stack. Execution continues at the top of the newly written frames.

In [18] and all subsequent systems we are aware of, the deoptimization target is the first tier of an adaptive optimization system: a bytecode interpreter or code generated by a baseline compiler. Both have frames that fulfill the necessary requirement of the deoptimization handler: stack frames have a fixed layout that does not depend on the method that is executed, i.e., values are consecutive on the stack.

Our simpler system consists of only one compiler that can be configured for different optimization levels and execution scenarios. Since an optimizing compiler is required anyway to achieve excellent peak performance, it is desirable to use the same compiler also for the first execution tier by disabling expensive optimizations. This deoptimization target code uses the same stack frame layout as fully optimized code, i.e., values are no longer consecutive on the stack. Deoptimization needs to be adapted to this new target frame layout. We call that *deoptimization to optimized code*. The remainder of this section shows how deoptimization can be applied in this scenario, and which compiler optimizations are still allowed for deoptimization target code.

2.1 Matching of Scope Descriptors

Deoptimization entry points created by the optimizing compiler do not have a fixed stack frame layout, so deoptimization entry points also need metadata about the location of incoming values.

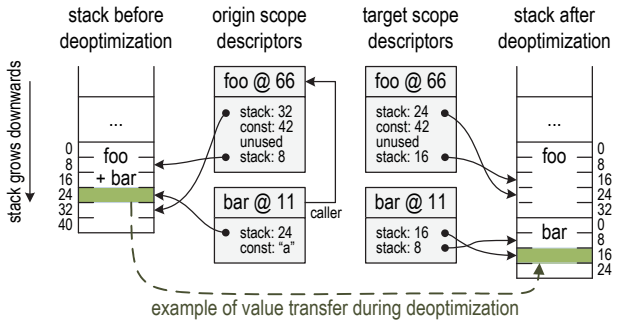


Figure 1: Deoptimization to optimized code using matching scope descriptors for the origin and target of deoptimization.

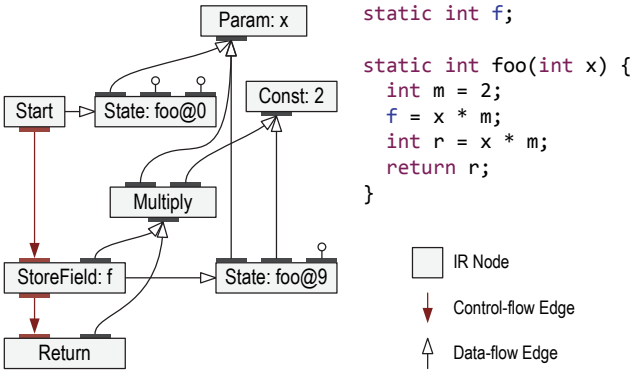
This metadata is stored using the scope descriptor format also used by deoptimization origin points. The optimizing compiler already contains all logic to create this information, which reduces the implementation and maintenance costs.

The deoptimization handler needs to perform a two-way matching of scope descriptors: The layout of the origin frame and the target frames are described using scope descriptors that have a matching abstract method position, i.e., the same method at the same virtual program counter with the same number of values. Iterating the values of both the origin and target frame simultaneously yields the origin and the target descriptor of a value. The deoptimization handler copies each value from the origin to the target location. If the target value is a constant, which can happen since the optimizing compiler performs constant folding also for deoptimization entry points, then no value needs to be copied. The deoptimization handler asserts that the origin value is the same constant, i.e., that the optimizing compiler performed the same constant folding also for the deoptimization origin.

Figure 1 illustrates the matching of scope descriptors. The deoptimization origin is method `foo` at a point where method `bar` is inlined. Two chained scope descriptors represent this state. The deoptimization handler replaces the deoptimized frame with two new target frames. Each of these frames has a scope descriptor without inlining. The virtual program counters and number of values in the scope descriptor are matching. In the method `bar`, two values are written to the stack frames, in the stack slots 16 and 8. Target stack slot 16 is filled from origin stack slot 24. These two stack slots are highlighted in the figure. Target stack slot 8 is filled with the string constant "a", which is the result of more aggressive constant folding in the optimized compilation due to, e.g., the method inlining of `bar` into `foo`. In the method `foo`, also two values are written to the stack frames even though there are four scope values. The second scope value, the constant 42, is the result of constant folding in the deoptimization target method. The value correctly matches the constant scope value in the deoptimization origin frame. The third scope value is a local variable that is unused at the deoptimization point, i.e., either not defined before the deoptimization point or not used after the deoptimization point. It is ignored by the deoptimization handler.

Every value that is live in a deoptimization target frame must be mentioned in the scope descriptor. Stack frame slots that are not written by the deoptimization handler, such as stack slot 24 in the target frame for method `bar`, must be unused at the deoptimization point. This reduces the optimization potential for deoptimization target code, but does not prohibit all optimizations (see Section 2.5).

We want one deoptimization target method to be usable for all possible deoptimization points of an origin method. In other




```
static int f;

static int foo(int x) {
  int m = 2;
  f = x * m;
  int r = x * m;
  return r;
}
```

Figure 2: Example compiler IR graph.

words, a deoptimization target method is a compilation of the whole method, and not just the continuation from a single deoptimization entry point. This avoids multiple compilations of the same deoptimization target method, and more importantly also allows ahead-of-time compilation of the deoptimization target methods before an actual deoptimization has happened. As a result, one deoptimization target method has many deoptimization entry points, i.e., many points described by a scope descriptor.

To avoid a complex matching of inlined methods, the scope descriptors for deoptimization entry points never have inlined frames. One deoptimization origin frame that has n methods inlined is always deoptimized to n deoptimized frames that have no methods inlined. Note that this does not preclude method inlining completely when compiling deoptimization entry points (see Section 2.5).

Escape analysis performed by the compiler removes allocations and replaces fields of objects with scalar values. During deoptimization, such objects need to be re-allocated. Scope descriptors for deoptimization origin frames contain all the necessary information for re-allocation (see for example [19]). To avoid a complex matching of escape analyzed objects, the scope descriptors for deoptimization entry points never have virtual objects, i.e., escape analysis must be disabled when compiling deoptimization target methods.

2.2 Deoptimization Entry Points in Optimized Code

Compiling deoptimization entry points using an optimized compiler requires small compiler extensions. We illustrate the details using the intermediate representation of the Graal[2] optimizing compiler [8]. The deoptimization approach itself is independent from the compiler, and we believe deoptimization entry points can be easily added to most optimizing compilers.

The intermediate representation of our compiler is structured as a directed graph in static single assignment (SSA) form [6]. Each IR node produces at most one value. To represent data flow, a node has *input* edges pointing to the nodes that produce its operands. To represent control flow, a node has *successor* edges pointing to its successors. In summary, the IR graph is a superposition of two directed graphs: the data-flow graph and the control-flow graph. This structure is illustrated in Figure 2. Note that the two kinds of edges point in opposite directions.

Nodes are not necessarily fixed to a specific point in the control flow. The control-flow graph provides a backbone around which most other nodes are *floating*. For example, the `Multiply` node in Figure 2 is floating. These floating nodes are only constrained by

[2] https://github.com/graalvm/graal-core

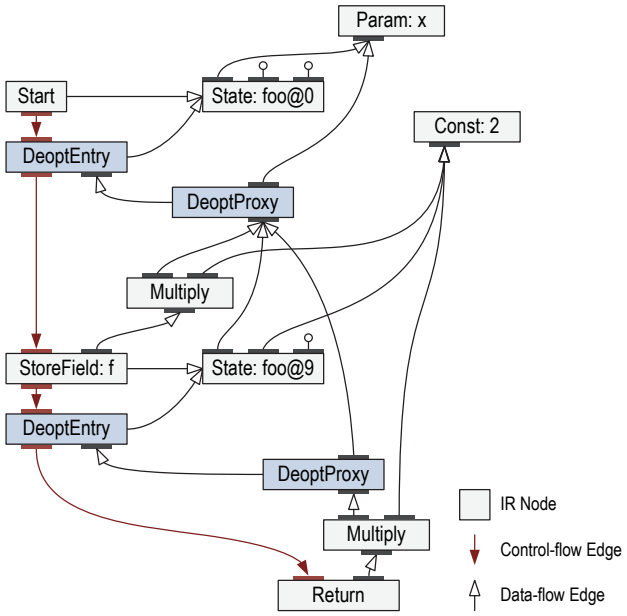


Figure 3: Example compiler IR graph with deoptimization entry points.

their data-flow edges, i.e., input values as well as additional dependencies such as memory dependencies and guarding dependencies. The dependencies maintain the program semantics but allow more freedom of movement for operations. The source code contains two multiplications, but since their input operands are equivalent only one `Multiply` node is created by the compiler.

Deoptimization requires knowing 1) where to continue execution, and 2) how to reconstruct the machine state (stack frame and registers) of the continuation point from the machine state of the optimized code. In order to know where we want to continue, we keep a reference to the method and the virtual program counter. For the VM state, we keep a mapping of the local variables and operand stack slots to their values in the IR. The information is maintained in a `State` node. The compiler maintains a `State` node with the new state after every instruction that cannot be re-executed, e.g., every memory store, every method invocation, and every control flow merge (see Section 2.3). In the example, the `State` node referenced by the `StoreField` node provides the new state to deoptimize to after changing memory. The example uses three local variables, so every `State` node references three input nodes. Local variables that are not yet assigned remain *uninitialized* and are ignored during deoptimization. Details about the deoptimization handling in the Graal IR have been published before [9]. Note that the entry points for deoptimization are compiler specific, and our deoptimization approach does not impose any assumption or constraints about deoptimization points.

Figure 3 shows the compiler IR when the same method is compiled with deoptimization entry points. A `DeoptEntry` node represents an entry point. It references a `State` node for the mapping of local variables. After IR graph building, every state-changing node is followed by a `DeoptEntry` node. Later compiler optimizations can remove the state-changing node if it is unnecessary. The `DeoptEntry` node must not be removed or moved by the compiler.

Every value that is alive across a `DeoptEntry` point must be mentioned in the `State` node, i.e., must be a local variable. Reusing the first `Multiply` node also for the second multiplication is not allowed because it would introduce a new value that is alive

across the second `DeoptEntry` point. To prevent such compiler optimizations, we wrap all local variable values in a `DeoptProxy` node. All subsequent usages use the wrapped nodes. No machine code is generated for a `DeoptProxy` node, but the compiler treats it as if it would compute a new value: the input of a `DeoptProxy` node and the `DeoptProxy` are not equivalent from the compiler's point of view. Therefore, the two `Multiply` nodes have different input values and are no longer equivalent. The `DeoptEntry` and `DeoptProxy` nodes are inserted during IR graph building, to avoid complicated rewrites of the graph.

2.3 Placement of Deoptimization Entry Points

The placement of deoptimization entry points depends on the deoptimization strategy chosen by the execution environment. Two different strategies are possible. Our deoptimization approach works with both strategies, since the placement of deoptimization entry points is the responsibility of the optimizing compiler.

1. Deoptimization restores the stack frame of execution exactly at the virtual program counter of the instruction that triggers deoptimization. For example, if deoptimization happens at a field load where the null check was speculatively eliminated but then the object is `null` during execution of the optimized code, deoptimization restores the state just before the field load. The deoptimization target code immediately throws the `NullPointerException`. `DeoptEntry` points need to be emitted for all instructions that might deoptimize. The Java HotSpot server compiler [21] (the last-tier compiler of the Java HotSpot VM) uses this approach.
2. Deoptimization restores the stack frame after the last state-changing instruction before the virtual program counter of the instruction that triggers deoptimization. For example, if deoptimization happens at the field load mentioned before, deoptimization restores some state before the field load. The deoptimization target code continues execution until it reaches the field load again, and then throws the `NullPointerException`. A limited amount of code is executed twice, but there is no behavior change. `DeoptEntry` points only need to be emitted after all instructions that are not re-executable, i.e., might change memory or produce outside-visible effects. These include memory writes, method invocations (because the callee can perform state-changing instructions), and control flow merges (because the information about which predecessor was executed is not preserved). The increased flexibility for deoptimization allows more aggressive compiler optimizations: deoptimization origins for, e.g., null checks or array bounds checks can be grouped or hoisted out of loops. The Graal compiler that we use for examples in this paper uses this approach: Figure 3 has a deoptimization entry point after the `StoreField` node, because the memory store cannot be re-executed.

2.4 Deoptimization Entry Points for Method Invocations

In addition to explicit deoptimization entry points, method calls are implicit entry points. If deoptimization happens while the method is currently executed, i.e., while the method invocation is on the stack, deoptimization needs to restore the state of the deoptimization target during the invocation. Both placement strategies mentioned above need these *during-invoke* deoptimization entry points.

The second approach for placement of deoptimization entry points, which we use for examples in this paper, requires additional entry points for every method invocation. If the deoptimization origin is after an invoke, the deoptimization entry point modeling the state during the invoke cannot be used since it does not contain the return value. A third deoptimization entry point is necessary in the exception handler of the invoke. It is used in case the callee throws

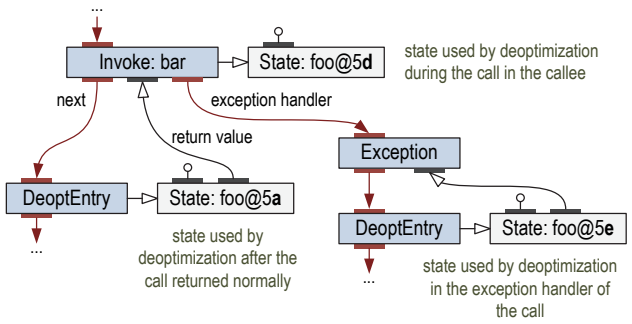


Figure 4: Deoptimization entry points for an invocation.

an exception and the deoptimization origin is in the exception handler. Figure 4 shows a method invocation in the method `foo` with the three deoptimization entry points for the invocation of method `bar`. Since the virtual program counter for all three entry points is the same, additional tags are necessary to distinguish the entry points. The tags `d`, `a`, and `e` are shown in Figure 4 in the `State` nodes.

1. The tag `d` denotes the entry point during the invoke. The state references all values that are live across the invocation, but no return value. When deoptimization happens, the stack frame for the methods `foo` and `bar` are on the stack.
2. The tag `a` denotes the entry point after the invoke. It has one more value than the `d` state: the return value of the method.
3. The tag `e` denotes the entry point in the exception handler of the invoke. It has one more value than the `d` state: the exception that was thrown by the callee. In our compiler IR, every IR node represents the value it produced. Since the `Invoke` node produces the normal method return value, we need a separate `Exception` node to model the exception value.

2.5 Compiler Optimizations for Deoptimization Targets

When compiling deoptimization target methods, the optimizing compiler can perform some but not all standard compiler optimizations.

Optimizations that are allowed in deoptimization target code:

- All local optimizations between deoptimization entry points. Such optimization do not affect the state and therefore the scope descriptors. Example optimizations are value numbering, common subexpression elimination, and strength reduction. For example, the multiplication by 2 in Figure 3 can be replaced with a left-shift by 1.
- All constant folding, even across deoptimization entry points. Constant values are allowed in scope descriptors. It does not matter if the constant is a literal constant in the source, or is the result of constant folding by the compiler. Constant values in the target scope descriptor are supported during deoptimization (see Section 2.1).
- Register allocation.
- Removing values from the state of deoptimization entry points, e.g., pruning of unused local variables. Such optimizations lead to *uninitialized* values in the scope descriptor, which are ignored during deoptimization.
- Dead code elimination, including deletion of unreachable deoptimization entry points, as long as the same or more dead code elimination is performed in the deoptimization origin code.

- Method inlining. No deoptimization entry points must be emitted in the inlined methods. A separate compilation is necessary where the inlined method is the root, this time with deoptimization entry points. This ensures that deoptimization target scope descriptors do not have inlined frames. We believe inlined frames would make matching of origin descriptors to target descriptors too complex.

  Method inlining removes the method invocation instruction and therefore the implicit during-invoke deoptimization entry point. Inlining must insert an explicit deoptimization entry point with the same state as the original invocation instruction. The new entry point must be after all instructions of the inlined method. The deoptimization handler restores a separate stack frame for the inlined method, i.e., the deoptimization entry point will be reached via a method return. Therefore, the explicit during-invoke deoptimization entry point must mimic the calling convention of the original method invocation, e.g., values must not use caller-saved registers.

Optimizations that are not allowed in deoptimization target code:

- Duplication of deoptimization entry points. The key for the lookup of a particular deoptimization entry point in the target method is the virtual program counter, plus the tags for states at method invocations. Duplication of a deoptimization entry point would lead to two entry points with the same virtual program counter, i.e., the lookup would no longer have a unique result. Many compiler optimization perform code duplication and are therefore not allowed, for example, loop peeling and loop unrolling.
- Optimizations that make new values alive across deoptimization entry points, for example, value numbering across deoptimization entry points. Such values would not be mentioned in scope descriptors, because they are not local variables of the original method. In the example in Figure 3, value numbering is not performed for the multiplication. Inserting proxy nodes as described in Section 2.2 is sufficient to prevent such optimizations. It is not necessary to disable optimizations such as global value numbering completely.
- Escape analysis. Virtual representations of escape analyzed object allocations are not allowed at deoptimization entry points. We believe that it would be too complex to match escape analyzed objects of the origin method with the matching escape analyzed objects of the deoptimization target method.
- Speculative optimizations, i.e., all optimizations that rely on deoptimization themselves. Deoptimization in deoptimization target code would replace a method with itself, leading to an infinite deoptimization loop.

2.6 Implementation Details

This section presents details of our implementation that are useful for implementers of our approach, but not an essential part or restriction.

For testing and debugging purposes, it is crucial to check in the compiler that all values alive across deoptimization entry points are referenced in scope descriptors. Errors would be difficult to identify: if a primitive integer value is not correctly written by the deoptimization handler, the application will continue without crashing but produce wrong results. We use an exhaustive assertion check for this in the back-end of the compiler. The register allocator already builds lifetime intervals for all values, i.e., during register allocation complete liveness information is already available. For each deoptimization entry point, we check that every value that is live, i.e., every value whose lifetime interval crosses the deoptimization entry point, is mentioned in the frame state (which is later converted to the scope descriptor). At this point during compilation, it is still possible to link the value back to the high-level compiler IR node that produced the value, which greatly simplifies locating the offending compiler optimization that introduced the compiler IR node.

Our lazy deoptimization is split into two stages: The first stage runs when frames are marked as deoptimized and the return address is patched to the deoptimization handler. We read origin frames at the first stage and build a heap-based representation of the target stack frames. All access to the scope descriptors happens at this stage. The second stage is the actual deoptimization handler, which only transfers the heap-based frame onto the stack. Because the scope descriptors are no longer needed by the deoptimization handler, the machine code and all metadata of deoptimization origin methods can be freed early, at the time frames are marked as deoptimized. We do not have to wait until execution has reached all deoptimized frames, which can take long when frames at the bottom of the stack are deoptimized.

3. Case Study: A High-Performance JavaScript Virtual Machine

We use and evaluate deoptimization to optimized code in a high-performance JavaScript VM. This case study allows us to present benchmark results using standard benchmarks, instead of synthetic micro-benchmarks that perform only deoptimization. It also shows that the deoptimization approach presented in this paper works in a complete and optimized VM implementation. However, we want to mention again that the basic approach presented in the previous section is independent of a specific compiler or VM.

Our JavaScript VM is written entirely in Java and adheres strictly to the one-compiler approach: the same compiler (Graal) is used to ahead-of-time compile the VM runtime (including the garbage collector) and to dynamically compile JavaScript code at run time. Our approach is similar to the Jikes RVM [2] and the Maxine VM [25], which are Java VMs written in Java. The garbage collector and the deoptimization handler are written in a low-level dialect of Java. Java can be used for low-level system programming without language extensions as long as the compiler exposes an unboxed pointer type and raw memory access [13]. The remaining parts of the JavaScript VM, such as the JavaScript parser, the JavaScript runtime, and the compiler are not using these low-level extensions. All reachable Java code is ahead-of-time compiled to a native executable. Java classes are loaded at this time, so there is no Java class file loading necessary at run time.

JavaScript code is executed using a mixed-mode approach: execution starts in an AST interpreter for fast startup and to collect profiling information; frequently executed JavaScript methods are dynamically compiled to achieve the best possible peak performance. To avoid implementing the JavaScript semantics twice (in the interpreter and the dynamic compiler), compilation is done by partial evaluation of the interpreter. The long-known theory of using partial evaluation for this purpose [14] has been shown to be feasible in practice in the last years in the Truffle project [26] for several languages [20] such as Ruby [7], R [24], and JavaScript. Truffle relies on speculative optimization of dynamically compiled code using the Graal compiler; and deoptimization to the interpreter in case speculations were too aggressive, e.g., when execution diverges from the type profiles collected by the interpreter. The GraalVM[3] executes Truffle languages on the Java HotSpot VM.

For our case study, we use the JavaScript implementation of the GraalVM project, but replace the Java HotSpot VM with a runtime

[3] http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/

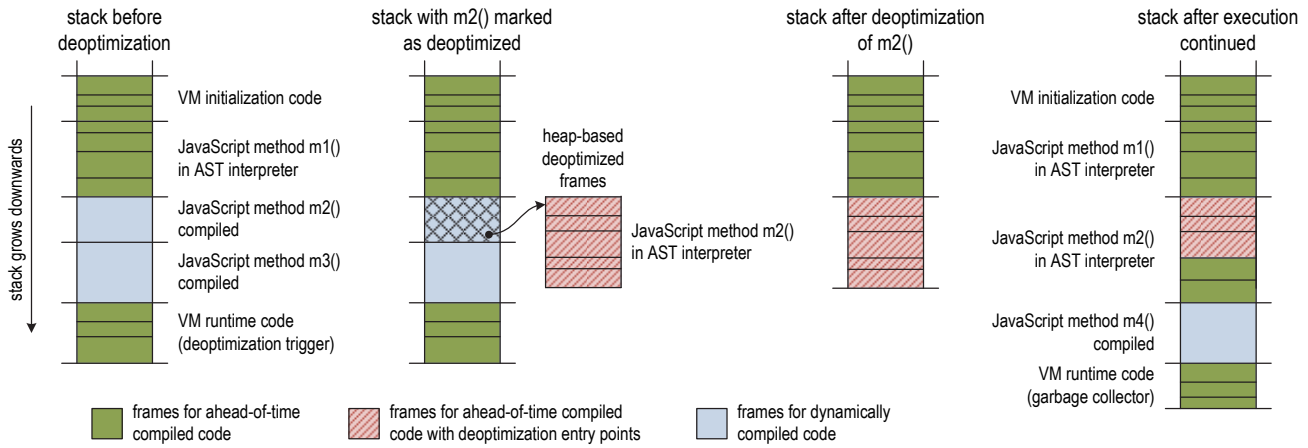


Figure 5: Frames before, during, and after deoptimization of our JavaScript VM.

system written in Java. We ahead-of-time compile the JavaScript AST interpreter, which is written in Java, together with the VM runtime, the garbage collector, and the Graal compiler. The Graal compiler is used to dynamically compile JavaScript methods at run time. When such a dynamically compiled JavaScript method is deoptimized, execution continues in the ahead-of-time compiled AST interpreter, i.e., the deoptimization target is optimized code that was ahead-of-time compiled. All possible deoptimization origin points that could be triggered at run time are known ahead of time, so we can also ahead-of-time compile all necessary deoptimization target methods.

When the dynamic compiler performs a partial evaluation, it inlines AST interpreter methods (written in Java) for all AST nodes of a particular JavaScript method. This means that the origin scope descriptors for a deoptimization usually have deep inlining. Deoptimization restores all the AST interpreter frames necessary to continue AST interpretation. All restored frames use the code compiled with deoptimization entry points. However, AST interpreter methods are usually short and invocation-heavy, i.e., they dispatch execution to child nodes, compute a result, and return the result to the parent node. All methods invoked by the AST interpreter after deoptimization are the regularly compiled methods, i.e., code without deoptimization entry points. The performance of deoptimization target code is not crucial for our VM, but it can still be relevant when using our deoptimization approach in a different setting, so we evaluate the performance of deoptimization target code in Section 4.4.

Figure 5 shows an extended example of stack frames and compilation in our JavaScript VM. The first stack on the left hand side has the VM initialization code, three JavaScript methods, and then some VM runtime code on the stack. Two of the three JavaScript methods have been executed frequently and were therefore compiled. A compiled JavaScript method consists of one stack frame. The first JavaScript method is running in the AST interpreter. An interpreted JavaScript method consists of many stack frames because the AST interpreter recursively calls the `execute` method of each AST node.

Assume that the VM runtime code on top of the stack triggers deoptimization of the JavaScript method `m2`, which is in the middle of the stack. The stack frame of method `m2` is marked as deoptimized. As mentioned in Section 2.6, we immediately construct a heap-based representation of the deoptimized frames (first stage of deoptimization). A pointer to this heap memory is installed into the deoptimized frame for later use by the deoptimization handler, i.e., the second stage of deoptimization. The pointer to the heap-based frame is the only data used from the stack frame until the second stage of deoptimization runs. Therefore, that frame is printed crossed-out in the second stack of Figure 5.

When execution returns to the deoptimized frame, the deoptimization handler transfers the heap-based representation of the deoptimized frames onto the stack. The third stack of Figure 5 shows the stack after the deoptimization handler has completed. Execution continues with the top frame of the deoptimized frames.

Deoptimized methods usually do not run for an extended period of time. The AST interpreter returns out of some deoptimized methods, and calls new AST interpreter methods. The newly called AST interpreter methods are regular ahead-of-time compiled methods without deoptimization entry points. Assume that the JavaScript method `m2` calls another JavaScript method `m4`, which then triggers a run of the garbage collector. The garbage collector is part of the VM runtime, i.e., also ahead-of-time compiled code. The fourth, right-hand-side stack of Figure 5 shows this final stack.

The final stack shows all three possible kinds of frames: 1) regular ahead-of-time compiled code, which is used both for the VM runtime and for the AST interpreter; 2) ahead-of-time compiled code with deoptimization entry points, which is used only for the few frames that are the result of deoptimization; and 3) dynamically compiled code, which is the result of the partial evaluation. All three kinds of frames are created by the same compiler, use the same stack frame layout, are described by the same metadata for the garbage collector, and can be debugged and inspected using the same tools.

4. Evaluation

This section evaluates our deoptimization approach using the case study, our high-performance JavaScript VM. All measurements were performed on a dual-socket Intel Xeon E5-2699 v3 with 18 physical cores (36 virtual cores) per socket running at 2.30 GHz, 256 GByte main memory running at 1600 MHz, Oracle Linux Server release 6.5 (kernel version 2.6.32-431.29.2.el6.x86_64), and Oracle JDK 1.8.0_111. All benchmarks were run on a server with a minimal setup and with no CPU consuming processes running and with frequency scaling disabled. Performance results are the mean of 5 executions (each execution in a new process) with a relative standard deviation below 5%.

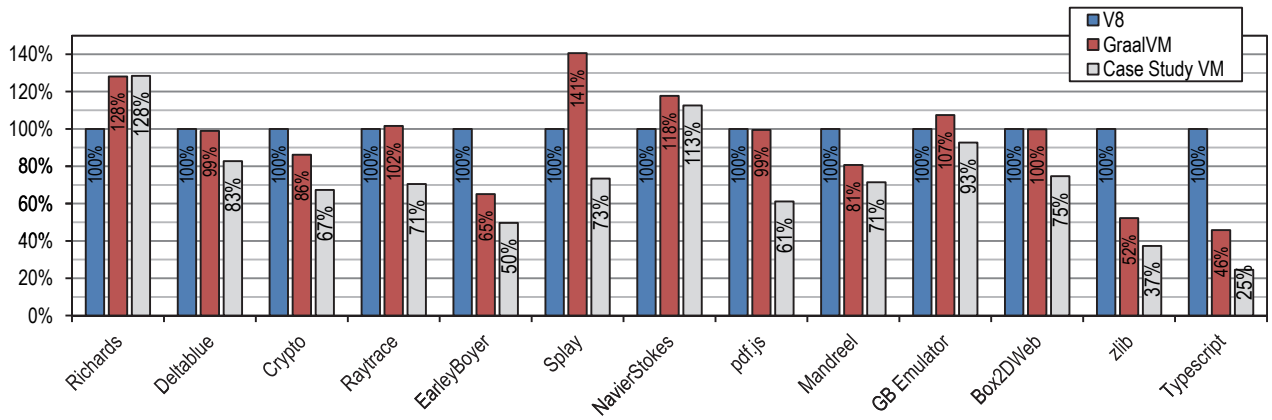


Figure 6: Peak performance for JavaScript Octane benchmarks (higher is better).

4.1 JavaScript Performance

Figure 6 shows the peak performance of our JavaScript VM. Note that peak performance is not a contribution of this paper, since it is independent of the deoptimization approach used in a VM. We include the numbers to 1) show that our case study JavaScript VM can compete with production-quality high-performance JavaScript VMs, i.e., that our deoptimization approach does not sacrifice peak performance, and 2) avoid the wrong interpretation of the numbers presented in Section 4.4 as peak performance numbers.

We use the standard Octane JavaScript benchmark suite to evaluate JavaScript performance. We exclude the *Latency* benchmarks because they only report a metric derived from the non-latency version of the according benchmark, the *CodeLoad* benchmark because it measures performance of the JavaScript source code parser, and the *Regexp* benchmark because it does not execute much JavaScript code. The excluded benchmarks are not related to JavaScript peak performance or deoptimization.

Figure 6 compares three different JavaScript VMs:

1. V8: a production-quality JavaScript VM written in C++ (version 5.3.332.6). It uses a baseline compiler as the first tier and the Crankshaft optimizing compiler as the second tier. Both compilers are designed and optimized to only execute JavaScript. Deoptimization target is baseline compiled code.
2. GraalVM: the Truffle JavaScript implementation running on the Java HotSpot VM. In the Java HotSpot VM, the deoptimization target is the Java bytecode interpreter, i.e., it does not use the deoptimization approach presented in this paper. Other than that, the execution and optimization of JavaScript code is similar to our case study: the Graal compiler performs a partial evaluation of the JavaScript AST interpreter.
3. Our case study VM as presented in Section 3: the Truffle JavaScript implementation running on our runtime and using the deoptimization approach presented in this paper.

The performance of the GraalVM is comparable to the V8 VM, most benchmarks are in a range of +/-30%. The performance of our case study VM is slightly worse than GraalVM because our runtime system, especially the garbage collector, is not as optimized as the Java HotSpot VM. This is especially true for the *Splay* benchmark, which stresses the garbage collector.

4.2 Size of Deoptimization Target Code

Figure 7 presents static numbers about deoptimization entry points and code size for our case study JavaScript VM. Deoptimization target code, i.e., code with deoptimization entry points, is necessary for all methods that can be dynamically compiled and therefore be the origin of a deoptimization. In our case study, this is the JavaScript AST interpreter that consists of 5,470 Java methods. When compiled with deoptimization entry points, they require 2.18 MByte of machine code. Compiled without deoptimization entry points, they require 2.43 MByte of machine code. Both versions are ahead-of-time compiled and included in our JavaScript executable. The size of the two versions is different because of the differences in allowed compiler optimizations, as explained in Section 2.5.

Methods requiring deoptimization entry points	5,470
Code size with deoptimization entry points [MByte]	2.18
Code size without deoptimization entry points [MByte]	2.43
Deoptimization entry points	41,988
Explicit deoptimization entry points	26,980
Implicit deoptimization entry points at invokes	15,008
Scope values of all entry points	118,912
Average number of values per entry point	2.8
Size of encoded deoptimization metadata [MByte]	0.86
Average metadata size per entry point [Byte]	21.5
Total number of compiled methods	27,843
Code size [MByte]	18.09

Figure 7: Deoptimization entry points in our JavaScript VM.

The 5,470 methods have a total of 26,980 explicit deoptimization entry points, and 15,008 implicit deoptimization entry points during method invocations. All entry points together have 118,912 values, which means on average one entry point has only 2.8 live values. We use a dense binary encoding to reduce the size of the deoptimization information, since it is read only infrequently. All entry points are encoded in 0.86 MByte, so on average one entry point is encoded in 21.5 Byte.

In total, the JavaScript executable contains 27,843 compiled methods with 18.09 MByte of machine code. Note that due to method inlining, the number of compiled methods is lower than the number of methods in the source code. The number of methods includes all parts of the JavaScript VM and runtime system, e.g., it includes the JavaScript parser and the garbage collector.

	Richards	Delta-blue	Crypto	Ray-trace	Earley-Boyer	Splay	Navier-Stokes	pdf.js	Mand-reel	GB Emu-lator	Box2D-Web	zlib	Type-script
Number of runtime compiled methods	46	96	135	161	151	47	24	575	212	629	319	32	1780
Code size [MByte]	0.12	0.42	2.15	0.76	1.52	0.90	0.26	14.38	3.10	2.17	1.92	0.72	18.61
Size of scope descriptor [MByte]	0.19	1.59	4.41	1.57	2.96	3.82	0.33	27.99	4.36	4.89	3.49	2.81	50.43
Number of deoptimizations	0	0	31	9	9	10	4	20	20	194	30	6	252
Time in deoptimization handler [ms]	0.00	0.00	0.10	0.03	0.04	0.03	0.01	0.07	0.05	0.20	0.05	0.01	0.44
Number of frames restored	0	0	603	200	217	186	60	311	243	821	223	42	2283
Average frames per deoptimization	0.0	0.0	19.5	22.2	24.1	18.6	15.0	15.6	12.2	4.2	7.4	7.0	9.1
Number of scope values	0	0	580	183	183	182	54	310	226	1075	246	39	2465
Average scope values per frame	0.0	0.0	1.0	0.9	0.8	1.0	0.9	1.0	0.9	1.3	1.1	0.9	1.1
Objects re-allocated	0	0	223	80	108	64	43	105	79	9361	87	11	566
Average objects per deoptimization	0.0	0.0	7.2	8.9	12.0	6.4	10.8	5.3	4.0	48.3	2.9	1.8	2.2

Figure 8: Performance of deoptimization and statistics about restored frames.

4.3 Performance of Deoptimization

Figure 8 shows dynamic numbers collected during the run of each Octane benchmark. The number of methods compiled at run time and the code size heavily depend on the size of the benchmark. The partial evaluator produces scope descriptors with deep inlining and escape analysis requires metadata to re-allocated objects during deoptimization. Therefore the size of the frame descriptors relative to the code size and relative to the number of compiled methods is higher than the ratios for deoptimization target code presented in the previous section.

The number of deoptimizations is generally low, with a few exceptions for complex benchmarks such as *GB Emulator* and *Typescript*. But even the benchmarks with more deoptimizations spend less than one millisecond in the deoptimization handler, which is insignificant compared to the seconds or minutes of benchmark execution time. This is the expected behavior: deoptimization is the safety net that allows speculative optimizations. But the interpreter collects enough profiling information before compilation to avoid using deoptimization.

In our case study, deoptimization restores the state of an AST interpreter, i.e., of call-intensive code. Therefore, the average number of frames restored per deoptimization, i.e., the number of scope descriptors processed during one deoptimization, is high. Escape analysis in compiled code is essential to replace the heap-based local variables of the interpreter with registers in compiled code, which leads to the high number of objects re-allocated during deoptimization. The average number of scope values per frame is low, which matches the observations of Figure 7.

4.4 Performance of Deoptimization Target Code

Since deoptimization happens rarely and execution usually stays in deoptimization target code only for a short amount of time, the performance of deoptimization target code does not matter in our case study. Still, we are interested and want to report the performance of deoptimization target code. For this, we use a synthetic experiment where we use our JavaScript VM like a pure Java benchmark by disabling the dynamic compiler. This means that the AST interpreter runs like a normal Java application. We then force the execution to use a) the deoptimization target code for every executed method (including the runtime system such as the garbage collector), or b) the deoptimization target code for just the AST interpreter. The configuration b) uses the deoptimization target code only for parts that could actually be deoptimized, i.e., for the methods listed in Figure 7 "requiring deoptimization entry points".

	Score Geomean	Relative
Performance of our system		
Fully optimized code (our ahead-of-time compiler)	152.5	100%
Everything using deoptimization target code	70.9	46%
AST interpreter using deoptimization target code	107.5	70%

Figure 9: Performance comparison of deoptimization target code relative to deoptimization deoptimization origin code, for Octane JavaScript benchmarks (higher score is better).

Figure 9 shows the performance difference for our system, i.e., deoptimization target code is generated by the same optimizing compiler hat produces the fully optimized code. The explicit deoptimization entry points and the limitations to compiler optimizations reduce performance, but only to 46% when running only deoptimization target code, and 70% when running the AST interpreter with deoptimization entry points. In cases where code size is more important than peak performance, it would therefore be feasible to use the deoptimization target code also for regular execution of the AST interpreter.

The compile time of deoptimization target code is similar to the compile time for fully optimized code: Handling of the deoptimization entry points adds some compile time, while disabling the disallowed optimization reduces compile time.

5. Related Work

Deoptimization was introduced to support debugging in the SELF VM [18]. In the interactive environment of SELF, the programmer could set breakpoints and replace methods with new versions at any time. Deoptimization allowed implementing these features without impacting the peak performance, because the optimizing compiler did not need to create code or restrict optimizations. Instead, execution reverted to the first-tier execution using deoptimization when the user started debugging a method. Debugging is still an important application of deoptimization, for example, the Java HotSpot VM uses deoptimization for debugging the same way as SELF did. However, debugging is by far not the only application of deoptimization. It is useful for all speculative compiler optimizations, i.e., optimizations where the compiler omits code for a complicated but infrequent situation. Many feedback directed adaptive optimizations have been proposed and implemented [4].

Most current implementations of deoptimization follow the original design of the SELF VM. The Jikes RVM [2] is a notable exception [12]. It implements deoptimization as a generalized form of on-stack-replacement [17]. Deoptimization target code can be produced by any compiler, but it is bound to a single actual deoptimization. Values are not coming from a scope descriptor, but emitted as constants in on-the-fly generated bytecodes. Using the optimizing compiler for deoptimization targets is therefore unattractive due to the long compilation time that is added to the deoptimization time.

Simple interpreters can be written quickly and without architecture specific code. Production-quality interpreters are heavily optimized though, making their implementation complex and difficult to port. A non-comprehensive list of optimizations include dispatching techniques [5, 11], caching of values in registers [10], and combination of instructions to super-instructions [22]. The interpreter of the Java HotSpot VM is even generated during startup of the VM [15], i.e., the C++ code is actually an architecture-specific assembler generator[4]. To ease porting, the Java HotSpot VM has a second, slower interpreter written in pure C++ code[5].

Baseline compilers eliminate the dispatching overhead of interpreters, but several of the interpreter optimizations techniques still apply to baseline compilers. In many cases, they are also highly architecture-specific. For example, the baseline compiler of the V8 JavaScript VM has more architecture-specific code for each of the supported architectures than architecture-independent code[6], even when not counting the assembler code performing the actual instruction encoding (which can be shared with the optimizing compiler). Our deoptimization to optimized code eliminates the need for an interpreter or baseline compiler can therefore reduce the complexity of a VM significantly.

Inserting deoptimization entry points in a baseline compiler is often done manually and therefore error-prone. For example, the developers of the V8 baseline compiler marked more than a hundred places in the baseline compiler as entry points by calling one of the `PrepareForBailout*` functions (we added the number of calls in the architecture-independent file `full-codegen.cc` and one of the architecture-specific files). Missing deoptimization entry points, and even worse manually placed deoptimization entry points at slightly wrong positions, lead to hard-to-debug errors. In our system, deoptimization entry points are automatically placed by the compiler at all necessary positions.

6. Conclusions

Deoptimization to optimized code uses the same compiler for the origin as well as the target of deoptimization: for the speculatively optimized code that is optimized as aggressively as possible, and for the deoptimization-target optimized code where execution continues after a deoptimization. Deoptimization uses a two-way matching of the scope descriptors that describe live values, so that the location of origin and target values can be freely decided by the compiler. The scope descriptors have a matching state, i.e., the same virtual program counter and the same live values. This restricts some optimizations that the compiler can perform for deoptimization target code, but many important optimizations such as method inlining, constant folding, and register allocation are still possible.

Our evaluation using a high-performance JavaScript VM shows that the approach works in practice and that deoptimization target code is fast. However, the main advantage of deoptimization to optimized code is a simplification of the VM architecture: there is no need for a separate interpreter or baseline compiler, and all parts of the VM that need to walk the stack are simple because all stack frames are produced by the same optimizing compiler.

Acknowledgments

We thank the members of the Virtual Machine Research Group at Oracle Labs for their support and contributions. We especially thank Paul Wögerer, Peter Kessler, and Gilles Duboscq for feedback on this paper.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] O. Agesen and D. Detlefs. Mixed-mode bytecode execution. Technical report, Sun Microsystems, Inc., 2000.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[3] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65. ACM Press, 2000. doi: 10.1145/353171.353175.

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005. doi: 10.1109/JPROC.2004.840305.

[5] J. R. Bell. Threaded code. *Communications ACM*, 16(6):370–372, 1973. doi: 10.1145/362248.362270.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[7] B. Daloze, S. Marr, D. Bonetta, and H. Mössenböck. Efficient and thread-safe objects for dynamically-typed languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 642–659. ACM Press, 2016. doi: 10.1145/2983990.2984001.

[8] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[9] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*, 2013. doi: 10.1145/2542142.2542143.

[10] M. A. Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 315–327. ACM Press, 1995. doi: 10.1145/207110.207165.

[11] M. A. Ertl and D. Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 403–412. Springer-Verlag, 2001.

[12] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 241–252. IEEE Computer Society, 2003. doi: 10.1109/CGO.2003.1191549.

[4] http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/tip/src/cpu/x86/vm/templateTable_x86.cpp

[5] http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/tip/src/share/vm/interpreter/bytecodeInterpreter.cpp

[6] https://github.com/v8/v8/tree/master/src/full-codegen

[13] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 81–90. ACM Press, 2009.

[14] Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):721–728, 1971.

[15] R. Griesemer. Generation of virtual machine code at startup. In *Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, 1999.

[16] U. Hölzle and D. Ungar. A third-generation SELF implementation: Reconciling responsiveness with performance. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 229–243. ACM Press, 1994. doi: 10.1145/191080.191116.

[17] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336. ACM Press, 1994.

[18] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[19] T. Kotzmann and H. Mössenböck. Run-time support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60. IEEE Computer Society, 2007.

[20] S. Marr, B. Daloze, and H. Mössenböck. Cross-language compiler benchmarking: Are we fast yet? In *Proceedings of the Dynamic Languages Symposium*, pages 120–131. ACM Press, 2016. doi: 10.1145/2989225.2989232.

[21] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12. USENIX, 2001.

[22] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300. ACM Press, 1998. doi: 10.1145/277650.277743.

[23] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4), 2008.

[24] L. Stadler, A. Welc, C. Humer, and M. Jordan. Optimizing R language execution via aggressive speculation. In *Proceedings of the Dynamic Languages Symposium*, pages 84–95. ACM Press, 2016. doi: 10.1145/2989225.2989236.

[25] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4): 30:1–30:24, 2013.

[26] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of Onward!* ACM Press, 2013.